Od Autora

Niniejsza książka jest ***Poradnikiem programisty-eksperta***, to jest każdego, kto chce nabrać biegłości w pisaniu programów i mieć możliwość uzyskiwania odpowiedzi na pytania powstające w procesie programowania.

Zainteresowanych informuję, iż mimo pozornych podobieństw do moich wcześniejszych monografii, ***Java XP*** jest całkowicie **zmienioną**, **rozszerzoną** i **poprawioną** ich wersją, uwzględniającą doświadczenia jakich nabyłem podczas ponad **5**-letniego nauczania i praktykowania Javy.

Właściwym początkiem książki jest rozdział ***Podstawy języka***. W odróżnieniu od poprzednich prac, w których unikałem wczesnego wprowadzania pojęć związanych z obiektowością, przedstawiam w nim Javę jako język obiektowy już od samego początku.

W celu umożliwienia niecierpliwym rychłego przystąpienia do pisania programów, bezpośrednio po ***Podstawach języka*** umieściłem rozdział ***Szybki start***. Ma on ułatwić oswojenie się ze składnią i semantyką deklaracji, wyrażeń i instrukcji oraz zapoznać z aromatem praktycznego programowania.

Pozostała część książki jest szczegółowym opisem języka i jego zastosowań. Liczne przykłady programów oraz obszerne ***Dodatki***, wśród których szczególnie polecam ***Java 2 SDK*** i ***Kawa 5 Enterprise***, a także ***Pytania i odpowiedzi*** oraz ***Quiz dla ekspertów***, powinny znacznie ułatwić biegłe posługiwanie się Javą.

Na dołączonej do książki dyskietce umieściłem teksty źródłowe programów oraz wymagane przez nie pliki graficzne i dźwiękowe. Tamże można znaleźć ***projekty wzorcowe***, ułatwiające pisanie własnych programów oraz eksperymentowanie z przytoczonymi w książce.

prof. Jan Bielecki

*Niektórych może dziwić, dlaczego po latach publikowania w Wydawnictwie **Helion**, z którym współpraca układała mi się nad wyraz dobrze, wróciłem „do korzeni", to jest do **PLJ**. Sprawa jest prostsza niż się wydaje: otóż Helion zawarł umowę z amerykańskim Wydawcą Strategicznym, który w obawie przed konkurencją ze strony polskich autorów, postawił **Warunek**: żadnych książek o językach programowania pisanych przez Polaków. Fakt ten jest smutny, bo potwierdza, że jak tak dalej pójdzie, to potężni partnerzy rodzimych firm pozwolą nam, we własnym kraju, tylko sprzątać ulice.*

Szkielety

Współczesne programy, przewidziane do wykonania w środowiskach okienkowych, są zazwyczaj obarczone narzutem w postaci co najmniej kilkudziesięciu wierszy kodu, którego zrozumienie nie jest łatwe, a samodzielne napisanie, zwłaszcza przez początkującego programistę, niemal niemożliwe.

Sposobem na wybrnięcie z tego kłopotu jest przygotowanie ***szkieletu aplikacji***, to jest takiego kompletnego programu, którego fragmenty mogą być modyfikowane albo zastępowane innymi.

Podczas nauki programowania w Javie, szczególnie atrakcyjne jest zwłaszcza zastępowanie wycinków nazywanych ***pod-aplikacjami***, wycinkami zaczerpniętymi z książki. Dzięki niemu można szybko tworzyć całkiem złożone aplikacje, posługując się łatwą do ogarnięcia abstrakcją jaką jest pod-aplikacja.

Ta metodyka postępowania legła u podstaw moich ostatnich książek. Jej walory dydaktyczne zostały wysoko ocenione przez Recenzentów, Radę Wydziału i Senat Politechniki Warszawskiej, skutkując wnioskiem o

Uhonorowanie
prof. Jana Bieleckiego
Nagrodą Ministra Edukacji Narodowej

Za wybitny wkład
w nauczanie i upowszechnianie
nowoczesnych technik programistycznych
opartych na języku Java

A oto uzasadnienie Wniosku:

Prof. Jan Bielecki jest znanym i cenionym autorem ponad 200 publikacji, w tym 112 książek o tematyce komputerowej. Ze względu na walory dydaktyczne i merytoryczne, książki prof. Jana Bieleckiego cieszą się uznaniem zarówno w kraju i zagranicą, a ich łączny nakład przekroczył już 800 tys. egz.

W toku całej, już 35-letniej, działalności prof. Jana Bieleckiego jako nauczyciela akademickiego, jego zainteresowania koncentrowały się na problematyce oprogramowania komputerów, zwłaszcza programowania systemowego, języków programowania i metodologii tworzenia oprogramowania. Nie trzeba szerzej uzasadniać poglądu, że na przestrzeni ostatniego półwiecza dziedzina ta przeżywała niezwykle intensywny rozwój. Nie było roku, by nie powstawały wciąż nowe rozwiąza-

nia i propozycje, zapowiadające radykalny przełom w technice programowania. Oczywiście, ogromna ich większość szybko ginęła w niepamięci, a jedynie nieliczne okazywały się rzeczywiście przełomowe i prowadziły do powstania nowych paradygmatów programowania.

W takiej sytuacji trzeba było głębokiej znajomości przedmiotu, profesjonalizmu i jednocześnie intelektualnej odwagi, żeby bez wahania i trafnie ocenić doniosłość nowopowstałego języka programowania oraz ważną rolę, jaką odegra w przyszłości. Takimi właśnie walorami wykazał się prof. Jan Bielecki wprowadzając, jako pierwszy w Polsce, język programowania Java do dydaktyki specjalności informatycznych na polskich wyższych uczelniach i propagując go w licznych książkach i publikacjach.

Początkowo język Java był istotnie sceptycznie przyjęty przez środowisko informatyczne. Mimo to, praktycznie od chwili jego powstania, prof. Jan Bielecki twierdził i dawał temu wyraz w swoich publikacjach, że Java jest „skazana na sukces". Jako profesjonalista dostrzegł, że język ten wspiera w wyjątkowo spójny i konsekwentny sposób techniki programistyczne oparte na programowaniu obiektowym, współbieżnym, zdarzeniowym i rozproszonym, a jako doświadczony nauczyciel akademicki, że ma ponadto nieocenione walory dydaktyczne. Twierdził, że dzięki temu może wkrótce wyprzeć inne języki programowania systemowego, stając się nowoczesnym narzędziem współczesnego programisty.

Rzeczywistość potwierdziła tę tezę. Język Java bardzo się upowszechnił, obecnie w jego rozwoju biorą czynny udział wiodący giganci informatyczni, napisano o nim ponad 1800 książek, a nawet firma Microsoft, od początku walcząca z Javą, opracowała podobny do Javy język programowania C#.

Już w 1996 roku, a więc rok po pojawieniu się propozycji języka Java, prof. Jan Bielecki, wydał dwie książki: *Java po C++* oraz *Java od podstaw* (Intersoftland, 1996). W ślad za nimi wkrótce pojawiły się następne publikacje, upowszechniające kolejne, coraz bardziej rozbudowane wersje szybko rozwijającego się języka: *Java 2* (Intersoftland, 1997), *Java 3* (Helion, 1997), *Java 3 RMI* (Helion 1997), *Java w szkole* (Helion, 1999) oraz *Java na uczelni* (Helion, 1999). Książki te były poszukiwanymi pozycjami wydawniczymi, a jako wartościowe podręczniki akademickie były nagradzane przez Rektora i Ministra Edukacji Narodowej.

W roku 2000 prof. Jan Bielecki wydał obszerną, 2-tomową monografię *Java 4 Swing* (Helion, 2000, tom I i II, łącznie 1300 stron), którą można uznać za ukoronowanie jego dotychczasowej bogatej działalności wydawniczej. W tej monografii prof. Jan Bielecki posłużył się wartościową metodę dydaktyczną, nowatorską w tej dziedzinie, gdyż nie stosowaną dotąd w nauczaniu programowania. Jest ona oparta na tzw. „szkieletach aplikacji".

Dzięki opracowanemu przez Autora zestawowi „szkieletów" początkujący programista może przystąpić do stosowania nowoczesnych technik programistycznych (jak programowanie obiektowe, zdarzeniowe i współbieżne) bez konieczności koncentrowania się na formalnych wymaganiach systemów kompilacyjnych. Stwarza to możliwości pisania nowoczesnych programów osobom praktycznie bez uprzedniego przygotowania programistycznego. Jednocześnie, obszerna monografia ma charakter kompetentnego, referencyjnego opracowania, z którego mogą korzystać także programiści zawodowi.

Już poprzednio, prof. Jan Bielecki był odznaczony Złotym Krzyżem Zasługi oraz nagrodzony licznymi nagrodami Ministra Edukacji Narodowej i Rektora Politechniki Warszawskiej. Warto także dodać, że w roku 2000 otrzymał on na wystawie Infosystem 2000 w Poznaniu „Krzemowego Oskara", najwyższą polską nagrodą informatyczną, przyznawaną w tajnym głosowaniu internetowym. Stanowi ona wyraz uznania szerokiego środowiska polskich informatyków i sympatyków informatyki dla roli jaką prof. Jan Bielecki odegrał w rozwoju tej dziedziny.

Podstawy języka

Język programowania jest narzędziem do pisania ***programów***. W Javie program składa się z jednej albo z większej liczby definicji ***klas*** zgrupowanych w ***pakiety*** i umieszczonych w plikach źródłowych.

Po skompilowaniu plików źródłowych powstaje zestaw plików z rozszerzeniem **.class** zawierający program w *B*-kodzie. Wykonanie *B*-kodu powierza się ***Maszynie Wirtualnej*** (zazwyczaj emulowanej przez program **java**).

Na końcu dowolnego wiersza programu można umieścić ***komentarz***. Początkiem komentarza jest para znaków **//** (*skośnik*, *skośnik*), a końcem ostatni znak wiersza. Komentarzem jest także dowolny napis, który zaczyna się od znaków **/*** (*skośnik*, *gwiazdka*) i kończy najbliższą parą znaków ***/** (*gwiazdka*, *skośnik*). Teksty zawarte w komentarzach są uznawane za **niebyłe**.

Definicje klas

Definicja klasy jest **wzorem**, na podstawie którego tworzy się ***zmienne*** i ***obiekty***.

Ponieważ wykonanie programu zapisanego w Javie polega w głównej mierze na przetwarzaniu obiektów, uprawnione staje się twierdzenie, że Java jest językiem do ***programowania obiektowego***.

Definicje klas umieszcza się w plikach z rozszerzeniem **.java**. Jeśli klasę zadeklarowano jako **publiczną**

```
public
class Point2D {

    // ...
}
```

to część główna nazwy pliku musi być **identyczna** z nazwą klasy (skąd wynika, że w pliku **Point2D.java** nie może wystąpić definicja klasy publicznej innej niż **Point2D**).

Określenie pakietu

Zaleca się, aby zestaw definicji klas umieszczonych w pliku był poprzedzony ***określeniem pakietu***, na przykład

```
package jbPack;  // określenie pakietu

public           // klasa jbPack.Point2D
class Point2D {

    // ...
}
```

Wówczas wszystkie klasy zdefiniowane w tym pliku należą do tego samego pakietu.

Pełną nazwą klasy jest nazwa występująca po słowie kluczowym **class**, kwalifikowana nazwą pakietu (np. **jbPack.Point2D**). Kwalifikację za pomocą pakietu można pominąć wówczas, gdy do klasy odwołuje się w obrębie jej pakietu.

Polecenia importu

Po określeniu pakietu, a przed definicjami klas można umieścić ***polecenia importu***.

Jeśli w pewnym pakiecie (np. **jbUtils**) użyto polecenia importu z pełną nazwą klasy, na przykład

```
import jbPack.Point2D;
```

to w tym pakiecie odwołanie do klasy **jbPack.Point2D** można uprościć do jej identyfikatora (**Point2D**).

Jeśli w pewnym pakiecie użyto polecenia importu z nazwą klasy zastąpioną znakiem * (*gwiazdka*), na przykład

```
import jbPack.*;
```

to w tym pakiecie odwołanie do **każdej** klasy pakietu **jbPack** (m.in. do klasy **jbPack.Point2D**) można uprościć do jej identyfikatora (**Point2D**).

Wraz z każdym kompilatorem Javy muszą być dostarczone pakiety **java.lang** i **java.io**. Ponieważ pierwszy z nich zawiera klasy wykorzystywane w większości programów (m.in. **String** i **Math**), przed pierwszą definicją klasy zawartą w pliku źródłowym umieszcza się niejawnie polecenie importu

```
import java.lang.*;
```

Dzięki temu jawne importowanie klas pakietu **java.lang** jest zbyteczne (chociaż dozwolone).

Zmienne

Zmienną jest obszar pamięci operacyjnej, w którym przechowuje się ***dane***.

Od ***typu*** zmiennej zależy ***rozmiar*** obszaru, ***zakres*** wartości danych jakie można przypisywać zmiennej oraz ***reprezentacja*** danych.

W szczególności zmienna typu **int** (od **integer** – całkowity) zajmuje **4** bajty pamięci, za pomocą instrukcji przypisania jak

```
age = 24;
```

umożliwia przechowywanie danych z zakresu

```
-2,147,483,648 .. +2,147,483,647
```

i reprezentuje je w ***zapisie-uzupełnieniowym-do-2***.

W *zapisie-uzupełnieniowym-do-2*, wagi poszczególnych pozycji bitowych są kolejnymi potęgami liczby dwójki (**1**, **2**, **4**, **8**, itd.), ale waga pozycji najbardziej znaczącej jest **ujemna**. Zapis *uzupełnieniowy-do-2* ma tę właściwość, że jeśli **zaneguje** się wszystkie bity reprezentacji danej, po czym do wyniku doda **1**, to otrzyma się liczbę o takiej samej wartości bezwzględnej, ale z przeciwnym znakiem.

Typy zmiennych

Biorąc za podstawę typ zmiennej, można wyodrębnić

zmienne typów ***pierwotnych***
- numeryczne
  - całkowite
  - rzeczywiste
  - znakowe
- orzecznikowe

zmienne typów ***prostych***
- pierwotne
- odnośnikowe

zmienne typów ***złożonych***
- tablice
- obiekty

Zmienne typów prostych mogą być elementami tablic i obiektów, mogą być występować poza tablicami i obiektami oraz mogą być zmiennymi ***lokalnymi***, tworzonymi i niszczonymi podczas wykonywania funkcji.

Ponieważ w ostatnim przypadku nie ma zastosowania inicjowanie domyślne, należy zagwarantować, aby pierwsze odwołanie do zmiennej było poprzedzone jej ***zainicjowaniem***.

W szczególności, po opracowaniu deklaracji

```
double pi;
```

zmienna lokalna **pi** ma wartości **nieokreśloną**, ale po opracowaniu deklaracji z ***inicjatorem***

```
double pi = 3.14;
```

jest zainicjowana daną o wartości **3.14**.

Zmienne całkowite

Zmienne ***całkowite*** mogą przyjmować tylko wartości całkowite.

Poza **4**-bajtowymi zmiennymi całkowitymi typu **int** istnieją również **2**-bajtowe zmienne całkowite typu **short**, **1**-bajtowe zmienne całkowite typu **byte** oraz **8**-bajtowe zmienne całkowite typu **long**. Jak łatwo się domyślić, różnica między nimi dotyczy tylko zakresu wartości (np. dla typu **byte** jest nim zakres **–128 .. 127**).

W szczególności, po opracowaniu deklaracji

```
int count = 12;
```

zmienna **count** jest zainicjowana daną o wartości **12**.

Zmienne rzeczywiste

Zmienne ***rzeczywiste*** mogą przyjmować nie tylko wartości całkowite, ale również ***ułamkowe*** (np. **12.4**).

Zmienne rzeczywiste typu **double** są **8**-bajtowe, a zmienne rzeczywiste typu **float** są **4**-bajtowe.

Deklaracje zmiennych rzeczywistych mają postać

```
double width;
double height = 12.4;
```

Zmienne znakowe

Zmienne ***znakowe*** mogą przyjmować tylko takie wartości, które są kodami znaków w ***Unikodzie***.

Zmienne znakowe są typu **char**. Ponieważ są **2**-bajtowe, więc umożliwiają reprezentowanie nie tylko kodów znaków ***ASCII***, ale również kodów znaków **narodowych** (np. **ą** ma w *Unikodzie* wartość **261**).

W szczególności, po opracowaniu deklaracji

```
char chr = '2';
```

zmienna **chr** jest zainicjowana kodem znaku **'2'** (wartością **50**).

Zmienne typu **char** są **numeryczne**. Nic zatem nie stoi na przeszkodzie wykonywania na nich operacji arytmetycznych (np. **chr + 3** ma wartość **53**).

Zmienne orzecznikowe

Zmienne ***orzecznikowe*** są **1**-bajtowe i mogą przyjmować tylko wartości **true** (*prawda*) i **false** (*fałsz*), orzekające o prawdziwości albo fałszu ***orzeczeń*** (np. orzeczenie *"liczba **12** ma wartość ujemną"* ma wartość **false**).

W szczególności, po opracowaniu deklaracji

```
boolean isNegative = 12 < 0;
```

zmienna **isNegative** jest zainicjowana wartością **false**.

Zmienne odnośnikowe

Zmienne ***odnośnikowe***, w skrócie ***odnośniki***, mogą przyjmować tylko wartości ***identyfikujące*** zmienne tablicowe i obiektowe.

Istota ***odnośników*** i ***odniesień*** wynika z następujących wyjaśnień

1. Każda klasa definiuje nowy **typ** danych. W szczególności, jeśli klasa nazywa się **String** (*łańcuch*) to definiuje typ **String**, a jeśli nazywa się **Point2D** (*punkt*), to definiuje typ **Point2D**.

2. W celu utworzenia obiektu klasy, należy dokonać jego ***fabrykacji***. Do fabrykowania obiektów służy ***fabrykator***, w którym podaje się nazwę klasy. W szczególności opracowanie fabrykatora **new Point2D(10, 20)** powoduje sfabrykowanie obiektu klasy **Point2D** (reprezentującego punkt) i dostarczenie w miejscu fabrykacji **odniesienia** do właśnie sfabrykowanego obiektu. W podobny sposób fabrykuje się tablice (np. fabrykator **new int[5]** tworzy tablicę o **5** elementach typu **int**).

Tak jak na przykład zmiennym typu **int** można przypisywać dane całkowite, tak zmiennym odnośnikowym można przypisywać **odniesienia** do obiektów oraz ***odniesienie puste***, reprezentowane przez słowo kluczowe **null** (co oznacza, że odnośnik nie identyfikuje żadnego obiektu).

W szczególności, po opracowaniu deklaracji

```
Point2D point = new Point2D(10, 20);
```

zmienna **point** jest zainicjowana odniesieniem do obiektu reprezentującego punkt o współrzędnych **(10, 20)**.

Należy zwrócić uwagę na subtelność terminologiczną. O ile o obiekcie mówi się, że jest na przykład ***klasy*** **Point2D**, o tyle o analogicznym odnośniku mówi się, że jest ***typu*** **Point2D**.

Stałe

Stałą jest zmienna, którą można zainicjować, ale której **nie można** modyfikować.

Deklarację stałej należy poprzedzić specyfikatorem **final**. Zainicjowanie stałej może być co najwyżej jednokrotne i odbywa się zazwyczaj w miejscu jej zadeklarowania.

```
final double Pi = 3.141592;
```

Z racji nie-modyfikowalności, stałe są nazywane zmiennymi ***ustalonymi*** albo ***finalnymi***.

Literały

Literałem jest napis, który, mimo iż nie jest identyfikatorem, jest nazwą stałej typu prostego.

W szczególności literał **12** jest nazwą stałej typu **int**, literał **12.4** jest nazwą stałej typu **double**, literał **'a'** jest nazwą stałej typu **char**, literał **true** jest nazwą stałej typu **boolean**, a literał **"Ewa"** jest nazwą stałej typu **String**, zainicjowanej odniesieniem do obiektu reprezentującego łańcuch składający się ze znaków: **E**, **w**, **a**.

Tablice

Tablica jest zmienną typu **złożonego**, której ***elementami*** są zmienne typu **prostego**. Wymaga się, aby wszystkie elementy tablicy były takiego samego typu (np. **int** albo **String**).

W celu utworzenia tablicy należy ją sfabrykować. W fabrykatorze należy określić typ oraz liczbę elementów tablicy. W miejscu opracowania fabrykatora jest dostarczane odniesienie do tablicy. Odniesienie to przypisuje się zazwyczaj odnośnikowi typu tablicowego.

W szczególności

```
int[] values;
```

deklaruje odnośnik, któremu można przypisywać odniesienia do dowolnych tablic o elementach typu **int**, na przykład

```
values = new int[5];
```

natomiast

```
String[] strings;
```

deklaruje odnośnik, któremu można przypisywać odniesienia do dowolnych tablic, których elementami są odnośniki do obiektów klasy **String**, na przykład

```
int[] strings = new String[3];
```

Nazwy elementów

Z każdym elementem tablicy jest związany ***indeks***. Z pierwszym elementem jest związany indeks **0**, a z następnymi indeksy **1**, **2**, **3**, itd., aż do indeksu o **1** mniejszego od liczby elementów tablicy.

Jeśli w odwołaniu do elementu tablicy użyje się indeksu spoza dozwolonego zakresu, to powstanie sytuacja wyjątkowa **ArrayIndexOutOfBoundsException**. Brak jej jawnego obsłużenia (por. ***Wyjątki***) spowoduje przymusowe zakończenie wykonywania programu.

Odwołania do elementów tablicy mają postać

name[***index***]

gdzie ***name*** jest nazwą odnośnika do tablicy, a ***index*** jest wyrażeniem całkowitym. Każdy z takich napisów jest nazwą elementu tablicy.

W szczególności, następujący zestaw instrukcji tworzy **3**-elementową tablicę, a następnie przypisuje jej elementom odniesienia do obiektów reprezentujących imiona.

```
String[] names;
names = new String[3];
names[0] = "Ewa";
names[1] = "Iza";
names[2] = "Jan";
```

Jeśli nazwą odnośnika do tablicy jest ***name***, to wyrażenie ***name*****.length** dostarcza liczbę elementów tablicy. W szczególności wyrażenie **names.length** ma wartość **3**.

dla dociekliwych

Przytoczony powyżej zestaw instrukcji można zastąpić instrukcją

```
String[] names = new String[] { "Ewa", "Iza", "Jan" };
```

a także instrukcją

```
String[] names = { "Ewa", "Iza", "Jan" };
```

Podobne uproszczenia są w Javie dość liczne i znacznie skracają zapis programów.

Funkcje

Funkcja jest zamkniętym ***opisem czynności***.

Definicja funkcji nie może wystąpić poza klasą, ani w ciele innej funkcji. Zaleca się aby funkcje były krótkie i odwoływały się do innych funkcje. W przeciwnym razie program jest mało czytelny i trudny do uruchomienia.

Definicje funkcji

Definicja funkcji

1. Określa, czy funkcja dostarcza ***rezultat*** oraz jakiego jest on typu (tu **int**)

```
int fun(double par)
{
    // ...
}
```

albo określa, że jest bez-rezultatowa (tu **void**)

```
void fun(String par)
{
    // ...
}
```

2. Wyszczególnia deklaracje parametrów (tu **x** i **y**)

```
int fun(int x, double y)
{
    // ...
}
```

3. W przypadku funkcji rezultatowej określa wartość rezultatu (w **return**)

```
int sum(int x, int y)
{
    int result = x + y;
    return result;
}
```

W tej samej klasie może wystąpić więcej niż jedna funkcja o takiej samej nazwie. Zestaw takich ***przeciążonych*** funkcji uznaje się za poprawny tylko wówczas, gdy ich dowolne pary różnią się ***sygnaturami*** (mają odmienne zestawy typów parametrów).

Parametry i argumenty

W miejscu wywołania funkcji dostarcza się ***argumenty*** (chyba, że funkcja jest bezparametrowa i ich nie oczekuje).

Następnie każdy parametr funkcji kojarzy się z odpowiadającym mu argumentem. Skojarzenie polega na potraktowaniu parametru jako zmiennej lokalnej funkcji i zainicjowaniu jej wartością argumentu.

Po dokonaniu skojarzenia, operacje wykonywane za pośrednictwem parametru **mogą**, ale **nie muszą**, wpływać na wartość argumentu.

Jeśli parametr jest typu **pierwotnego**, to operacje na parametrze **nie mogą** zmodyfikować argumentu. Jeśli jest typu **tablicowego** albo **obiektowego**, to **mogą** spowodować zmodyfikowanie obiektu identyfikowanego przez argument.

W szczególności, jeśli w zasięgu deklaracji

```
int[] pair = { 10, 20 };

int fun(int[] vec, int value)
{
    vec[0] = 100;
    value  = 200;
    return pair.length;
}
```

zostanie wykonana instrukcja

```
int len = fun(pair, pair[1]);
```

to **len** otrzyma wartość **2**, element **pair[0]** otrzyma wartość **100**, ale element **pair[1]** pozostanie niezmieniony.

Analogicznie, jeśli w zasięgu deklaracji

```
String name = "Ewa";
```

```
String fun(String par)
{
    par = "Iza";
    return par;
}
```

zostanie wykonana instrukcja

```
String name2 = fun(name);
```

to **name2** otrzyma wartość **"Iza"**, ale **name** pozostanie niezmienione.

Obiekty

Obiektem jest zmienna typu **złożonego**, której ***elementami*** są zmienne typu **prostego**. W odróżnieniu od tablicy, której wszystkie elementy muszą być takiego samego typu, poszczególne elementy obiektu mogą być **różnych** typów.

Definiowanie klas

Prosta definicja klasy, opisującej obiekty reprezentujące punkty na płaszczyźnie ma postać

```
class Point2D {

    int x, y;              // pola

    Point2D(int x, int y)  // konstruktor
    {
        this.x = x;
        this.y = y;
    }
}
```

W podanej definicji występują deklaracje pól **x** i **y** oraz ***konstruktor***.

Konstruktor jest funkcją o nazwie identycznej z nazwą klasy, zdefiniowaną **bez określenia** czy jest funkcją rezultatową czy bez-rezultatową. Jego zadaniem jest użycie dostarczonych mu argumentów do zainicjowania elementów obiektu. W ciele klasy można zdefiniować więcej niż jeden konstruktor, ale jeśli nie dostarczy się ani jednego, to klasa zostanie niejawnie wyposażona w bezparametrowy konstruktor domyślny (którego ciało składa się z instrukcji **super();**).

W tym rozdziale będą rozpatrywane tylko klasy ***zewnętrzne***, to jest takie, których definicje nie są zawarte w innych klasach. Nic jednak nie stoi na przeszkodzie definiowania klas ***zagnieżdżonych***: ***wewnętrznych*** i ***zanurzonych***, a także klas ***anonimowych*** i ***lokalnych***, opisanych w dalszej części książki.

Fabrykowanie obiektów

Do utworzenia obiektu służy fabrykator. Fabrykator obiektu klasy ***Name*** ma postać

```
new Name(arg, arg, ... , arg)
```

w której każde ***arg*** jest argumentem konstruktora, na przykład

```
new Point2D(10, 20)
```

Opracowanie fabrykatora składa się z

1. Utworzenia obiektu o elementach opisanych przez pola klasy.
2. Utworzenia zmiennej **this** zainicjowanej odniesieniem do obiektu.
3. Wstępnego zainicjowania elementów obiektu danymi opisanymi przez inicjatory zawarte w deklaracjach pól (domyślnie: **0**, **false**, **null**).
4. Wywołania konstruktora w celu ostatecznego zainicjowania elementów obiektu.
5. Dostarczenia w miejscu fabrykacji odniesienia do obiektu.
6. Zniszczenia zmiennej **this**.

Ponieważ w ciele konstruktora, nazwą elementu obiektu opisanego przez pole ***field*** (np. **x** albo **y**) jest **this.field** (odpowiednio **this.x** i **this.y**), więc fabrykator

```
new Point2D(10, 20)
```

niejawnie wywołujący konstruktor

```
Point2D(int x, int y)
{
    this.x = x;
    this.y = y;
}
```

doprowadzi do zainicjowania elementów obiektu danymi całkowitymi o wartościach **10** i **20**.

Jeśli deklaracja pola opisującego element obiektu nie zawiera inicjatora, to niejawnie uzupełnia się ją inicjatorem dostosowanym do typu pola (**0** dla pól numerycznych, **false** dla orzecznikowych i **null** dla odnośnikowych). Dlatego, nawet gdy konstruktor nie zainicjuje pewnych elementów (w skrajnym przypadku, ponieważ jego ciało jest **puste**), cały obiekt jest jednoznacznie zainicjowany.

Nazywanie elementów

W miejscu użycia fabrykatora jest dostarczane odniesienie do sfabrykowanego obiektu. Jeśli zostanie przypisane odnośnikowi (zazwyczaj jest on typu identycznego z nazwą klasy), to umożliwi to nazywanie i odwoływanie się do elementów obiektu.

Zasada nazywania elementów jest następująca

> Jeśli odniesienie do obiektu zostanie przypisane odnośnikowi ***ref***, a element obiektu jest opisany przez pole ***field***, to nazwą tego elementu jest ***ref.field***.

W szczególności, jeśli na podstawie definicji klasy

```
class Point2D {

    int x, y;

    Point2D(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

sfabrykowano dwa obiekty

```
Point2D p1 = new Point2D(10, 20),
        p2 = new Point2D(30, 40);
```

to **p1.x** jest nazwą elementu o wartości **10**, a **p2.y** jest nazwą elementu o wartości **40**.

Hermetyzowanie obiektów

Umożliwienie niekontrolowanego dostępu do elementów obiektu (np. poprzez operacje na **p1.x** i **p2.y**) może doprowadzić do sytuacji, kiedy ***stan obiektu*** (zbiór aktualnych wartości jego elementów) jest trudny do określenia. Dlatego podczas definiowania klasy z reguły stosuje się ***hermetyzację***.

Hermetyzację składników klasy (jej pól i funkcji) wyraża się za pomocą specyfikatorów **private**, **public** i **protected**. Zazwyczaj do pól stosuje się hermetyzację **private** i **protected**, a do funkcji hermetyzację **public**.

```
class Point2D {

    private int x, y;

    public Point2D(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

Obowiązują następujące zasady

1. Jeśli składnik zostanie zadeklarowany ze specyfikatorem **private**, to jego identyfikator może być używany **tylko** w ciele jego klasy.

2. Jeśli składnik zostanie zadeklarowany ze specyfikatorem **public**, to jego identyfikator może być używany **wszędzie**.

3. Jeśli składnik zostanie zadeklarowany ze specyfikatorem **protected**, to jego identyfikator może być używany tylko w ciele jego klasy oraz w ciele jej ***klasy pochodnej*** (por. dalej).

Do składnika zadeklarowanego bez specyfikatora hermetyzacji można się odwoływać z ciała dowolnej klasy należącej do tego samego **pakietu** co klasa składnika.

Kontrolowanie dostępu

Po wprowadzeniu hermetyzacji, dostęp do elementów obiektu może być poważnie ograniczony, a nawet niemożliwy. W szczególności, jeśli odnośnikowi **p** przypisano odniesienie do obiektu z elementem opisanym przez prywatne pole **x**, to poza klasą tego obiektu **nie wolno** się odwoływać do elementu opisanego przez pole **x** (używając na przykład jego nazwy **p.x**).

W takim przypadku jednym ze sposobów uzyskania dostępu do elementów obiektu jest zdefiniowanie w jego klasie publicznych ***funkcji dostępujących*** (***pobieraczy*** i ***ustawiaczy***).

```
class Point2D {

    private int x, y;

    public Point2D(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public int getX()  // pobieracz
    {
        return this.x;
    }

    public int getY()  // pobieracz
    {
        return this.y;
    }

    public void setXY(int x, int y)  // ustawiacz
    {
        this.x = x;
        this.y = y;
    }
}
```

Wydawanie poleceń

Jeśli odnośnik ***ref*** zainicjowano odniesieniem do obiektu klasy, a w klasie jest widoczna funkcja ***fun*** zadeklarowana bez specyfikatora **static**, to napis

```
ref.fun(arg, arg, ... , arg)
```

jest ***poleceniem*** opisanym przez funkcję ***fun***, wydanym obiektowi identyfikowanemu przez odniesienie przypisane ***ref***.

Tuż przed wydaniem polecenia jest tworzony odnośnik **this** i inicjowany odniesieniem przypisanym ***ref***. Dzięki temu, **niezależnie** od tego jakiemu obiektowi jest wydawane polecenie ***fun***, w ciele tej funkcji, nazwą elementu obiektu opisanego przez pole ***field*** jest **this.***field*.

Po wykonaniu polecenia odnośnik **this** jest **niszczony**.

A zatem, jeśli sfabrykowano obiekt klasy z funkcjami **getX** i **setXY** (por. powyżej), a odniesienie do niego przypisano odnośnikowi **p**

```
Point2D p = new Point2D(10, 20);
```

to mimo iż elementy opisane przez pola **x** i **y** są niedostępne poprzez nazwy **p.x** i **p.y**, polecenie zawarte w instrukcji

```
int x = p.getX();
```

dostarczy aktualną wartość elementu opisanego przez pole **x**, a polecenie zawarte w instrukcji

```
p.setXY(30, 40);
```

zmieni wartości elementów obiektu na **30** i **40**.

Składniki statyczne

Składnik klasy zadeklarowany ze specyfikatorem **static** jest jej składnikiem ***statycznym***.

Każde statyczne pole klasy jest opisem unikalnego ***elementu statycznego***, występującego **poza** wszystkimi obiektami klasy i utworzonego w chwili pierwszego ***aktywnego użycia*** klasy (tj. tuż przed sfabrykowaniem dowolnego obiektu klasy albo tuż przed odwołaniem się do dowolnego jej elementu statycznego).

Składnikami statycznymi klasy mogą być zarówno pola jak i funkcje. Ponieważ podczas wykonania funkcji statycznej nie istnieje odnośnik **this**, więc funkcja statyczna może się odwoływać tylko do składników statycznych.

Zasada nazywania elementów statycznych jest następująca

> Jeśli element statyczny klasy ***Name*** opisuje jej pole statyczne ***field***, to nazwą tego elementu jest ***Name.field***.

Nazwę składnika statycznego można także kwalifikować nazwą odnośnika. W takim przypadku (którego nie zaleca się stosować) nazwa odnośnika jest niejawnie zastępowana nazwą jego klasy.

W szczególności, w zasięgu definicji

```
class Point2D {

    private int x, y;
    public static Point last;

    public Point2D(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public static void setLast(int x, int y)
    {
        Point2D.last = new Point2D(x, y);
    }
}
```

można w następujący sposób użyć zmiennej **Point2D.last**.

```
Point2D p;
    // tu jeszcze nie istnieje zmienna Point2D.last
p = new Point2D(10, 20);
    // tu już istnieje zmienna Point2D.last
Point2D.setLast(p.getX(), p.getY());
```

Jak wynika z podanego przykładu, samo zadeklarowanie odnośnika do obiektów klasy **nie stanowi** aktywnego jej użycia.

Pomijanie kwalifikacji

W celu uproszczenia zapisu programu, w pewnych sytuacjach zezwala się na pomijanie kwalifikatorów **this** i **Name**. W związku z klasą **Point2D** umożliwia to skrócenie odwołania **this.x** do **x** oraz odwołania **Point2D.last** do **last**.

Obowiązują tu następujące zasady

1. Jeśli w miejscu wystąpienia napisu **this.*id*** jest widoczna deklaracja **nie-statycznego** składnika ***id*** (bo nie przesłoniła jej na przykład deklaracja parametru funkcji), to **this.*id*** można uprościć do ***id***.

2. Jeśli w miejscu wystąpienia napisu ***Name.id*** jest widoczna deklaracja **statycznego** składnika ***id***, to ***Name.id*** można uprościć do ***id***.

Inne spojrzenie na pomijanie kwalifikatorów jest następujące

1. Jeśli w miejscu wystąpienia **nie-kwalifikowanego** odwołania do identyfikatora ***id*** jest widoczna deklaracja **nie-statycznego** składnika ***id***, a odwołanie występuje w funkcji nie-statycznej, to odwołanie ***id*** uznaje się za skrót od **this.*id***.

2. Jeśli w miejscu wystąpienia **nie-kwalifikowanego** odwołania do identyfikatora ***id*** jest widoczna deklaracja **statycznego** składnika ***id***, to odwołanie ***id*** uznaje się za skrót od ***Name.id***.

W szczególności następująca definicja, z której wyeliminowano **wszystkie** zbędne kwalifikacje

```
class Point2D {

    private int x, y;
    public static Point2D last;

    public Point2D(int x, int y)
    {
        setPoint(x, y);
    }

    public void setPoint(int x2, int y)
    {
        x = x2;
        this.y = y;
        last = new Point2D(x, y);
    }

    public static void setLast(int x, int y)
    {
        last = new Point2D(x, y);
    }
}
```

jest równoważna następującej definicji z pełnym zestawem kwalifikatorów

```
class Point2D {

    private int x, y;
    public static Point2D last;

    public Point2D(int x, int y)
    {
        this.setPoint(x, y);
    }

    public void setPoint(int x2, int y)
    {
        x = x2;  // wolno pominąć this!
```

```
            this.y = y;
            Point2D.last = new Point2D(x, y);
        }

        public static void setLast(int x, int y)
        {
            Point2D.last = new Point2D(x, y);
        }
    }
```

Dziedziczenie

Każda klasa, z wyjątkiem klasy **Object**, od której wywodzą się wszystkie pozostałe klasy, powstała przez ***odziedziczenie*** składników jej klasy bazowej.

Istota dziedziczenia (wyrażanego za pomocą słowa kluczowego **extends**) polega na tym, że jeśli dysponuje się pewną klasą, ale chce się posługiwać klasą z ***dodatkowymi*** albo ***przedefiniowanymi*** składnikami (ostatnie dotyczy tylko funkcji), to należy utworzyć klasę pochodną klasy bazowej i w niej zdefiniować dodatkowe składniki.

```
class Point3D           // klasa pochodna
    extends Point2D {   // klasa bazowa

    // ...
}
```

Wynika stąd, że jeśli obiekty klasy bazowej składały się z pewnej liczby **elementów**, to obiekty klasy pochodnej będą się składały z elementów takich samych typów oraz **dodatkowo**, z elementów opisanych przez **nie-statyczne** pola klasy pochodnej.

W szczególności, w zasięgu definicji klasy bazowej

```
class Point2D {

    private int x, y;

    public Point2D(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

można zdefiniować klasę pochodną

```
class Point3D
    extends Point2D {

    private int z;
```

```
    public Point3D(int x, int y, int z)
    {
        super(x, y);

        this.z = z;
    }
}
```

której obiekty składają się z **3** elementów typu **int** opisanych przez pola **x**, **y**, **z**.

Posługując się słowem kluczowym **super**, z konstruktora klasy **Point3D** wywołano konstruktor klasy **Point2D**, dostarczając mu argumenty **x** i **y**, na podstawie których nastąpi zainicjowanie odziedziczonego przez klasę **Point3D** pod-obiektu klasy **Point2D**.

dla dociekliwych

W starszych językach programowania (np. w C++) występuje możliwość dziedziczenia składników więcej niż jednej klasy. W Javie jest to możliwe tylko wówczas, gdy dotyczy ***interfejsu***: klasy, której składnikami są wyłącznie deklaracje statycznych zmiennych finalnych oraz deklaracje funkcji nie-statycznych (deklaracje, a nie definicje!).

Interfejs definiuje się za pomocą słowa kluczowego **interface**, a jego dziedziczenie wyraża za pomocą słowa kluczowego **implements**. Interfejs jest domyślnie klasą **abstrakcyjną**, jego pola są domyślnie **publiczne**, **statyczne** i **finalne**, a jego funkcje są domyślnie **publiczne** i **abstrakcyjne**.

```
interface Movable {

    double Pi = 3.141592;

    Point2D move(int dx, int dy);
}

interface Printable {

   void print(Graphics gDC);
}

public
class  Sprite
    extends BasicSprite
    implements Movable, Printable {

    // ...
}
```

Implementowanie interfejsu stanowi **zobowiązanie** klasy dziedziczącej do dostarczenia **definicji** wszystkich funkcji zadeklarowanych w interfejsie. Jeśli się tego nie uczyni, to klasa pochodna stanie się klasą **abstrakcyjną** (nie będzie można fabrykować jej obiektów).

Polimorfizm

Polimorfizm umożliwia zapisywanie poleceń ***polimorficznych***, to jest takich, których analiza dokonana **przed** podjęciem wykonywania programu nie umożliwia rozstrzygnięcia, do jakiej klasy będzie należeć funkcja opisująca polecenie.

Przedefiniowanie funkcji

Jeśli w klasie pochodnej zdefiniuje się funkcję, która ma taką sama **sygnaturę** jak nie-prywatna i nie-finalna funkcja jej klasy bazowej, to o funkcji klasy pochodnej mówi się, że ***przedefiniowuje*** funkcję klasy bazowej.

```
class Point2D {

    // ...

        // funkcja przedefiniowywana
    public boolean closerThan(double radius)
    {
        return Math.sqrt(x * x + y * y) < radius;
    }
}

class Point3D
    extends Point2D {

    // ...

        // funkcja przedefiniowująca
    public boolean closerThan(double radius)
    {
        return Math.sqrt(x * x + y * y + z * z) < radius;
    }
}
```

Polecenia polimorficzne

Polecenie polimorficzne

```
ref.fun(arg, arg, ... , arg)
```

realizuje się w taki sposób, że o wyborze realizującej je funkcji ***fun*** nie decyduje typ odnośnika ***ref***, ale typ przypisanego mu ***odniesienia***.

Podczas kompilowania polecenia wymaga się jedynie, aby w klasie odnośnika była widoczna funkcja ***fun***. Podczas wykonania polecenia przyjmuje się, że jego opisem jest funkcja ***fun*** widoczna w klasie odniesienia przypisanego odnośnikowi.

W szczególności, jeśli w zasięgu definicji klas

```
class Point2D {

    private int x, y;

    public Point2D(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

        // funkcja przedefiniowywana
    public double fromOrigin()
    {
        return Math.sqrt(x * x + y * y);
    }
}

class Point3D
    extends Point2D {

    private int z;

    public Point3D(int x, int y, int z)
    {
        super(x, y);

        this.z = z;
    }

        // funkcja przedefiniowująca
    public double fromOrigin()
    {
        return Math.sqrt(x * x + y * y + z * z);
    }
}
```

wykona się instrukcje

```
Point2D point = new Point3D(2, 6, 3);
double distance = point.fromOrigin();
```

to mimo iż odnośnik **point** jest typu **Point2D**, polecenie **point.fromOrigin**() będzie zrealizowane za pomocą funkcji z klasy **Point3D**. Spowoduje to, że zmiennej **distance** zostanie przypisana wartość **7.0**.

dla dociekliwych

Jeśli wywołanie funkcji nie-statycznej ma postać **fun()**, to jest w istocie skrótem od **this.fun()**, a o wyborze klasy, w której zdefiniowano funkcję o takiej nazwie, decyduje odniesienie przypisane **this**.

W szczególności, w zasięgu definicji

```
class Point2D {

    public void caller()
    {
        fun();  // this.fun();
    }

    public void fun()
    {
        // ...
    }
}

class Point3D
    extends Point2D {

    public void fun()
    {
        // ...
    }
}
```

wykonanie instrukcji

```
new Point2D().caller();
```

spowoduje wywołanie funkcji **Point2D.fun**, a wykonanie instrukcji

```
new Point3D().caller();
```

spowoduje wywołanie funkcji **Point3D.fun**.

Zarządzanie pamięcią

Miejsce dla **nie-statycznych** zmiennych programu rezerwuje się na ***stosie*** albo na ***stercie***, natomiast miejsce dla zmiennych statycznych rezerwuje się w obszarze dodatkowym.

Różnica między stosem i stertą polega na tym, że zmienne ***dokładane*** do stosu można z niego ***wyjmować*** tylko w kolejności **odwrotnej** do dokładania, podczas gdy zmienne dokładane do sterty można z niej wyjmować w **dowolnej** kolejności.

Na stosie są umieszczane zmienne lokalne funkcji, a na stercie tablice i obiekty.

Zmienna lokalna funkcji zaczyna istnieć po opracowaniu jej deklaracji i zostaje zniszczona w chwili, gdy sterowanie opuści blok, w którym została zadeklarowana.

Tablice i obiekty zaczynają istnieć tuż po ich sfabrykowaniu i można je zniszczyć nie wcześniej niż w chwili, gdy ani jeden odnośnik nie zawiera odniesienia to tej tablicy albo obiektu.

Element statyczny zaczyna istnieć w chwili aktywnego użycia jego klasy i jest niszczony tuż przed zakończeniem wykonywania programu.

Zarządca nieużytków

Niszczenie zmiennych odbywa się poza kontrolą programu i jest powierzane ***Zarządcy nieużytków***. W chwili zakończenia wykonywania programu niszczy się wszystkie jeszcze nie zniszczone zmienne.

W szczególności, podczas wykonywania funkcji

```
public void fun()
{
    Point2D p;
    {
        int[] vec = { 10, 20, 30 };
        p = new Point2D(vec[0], vec[1]);
    }
    p = null;
    // ...
}
```

odnośnik **vec** zostanie zniszczony tuż przed tym, gdy zakończy się wykonywanie bloku wewnętrznego, tablica będzie mogła być zniszczona, nie wcześniej niż po zniszczeniu odnośnika **vec**, a obiekt będzie mógł być zniszczony nie wcześniej niż po przypisaniu odnośnikowi **p** odniesienia pustego.

Strategia niszczenia

Strategia niszczenia zmiennych i odzyskiwania nieużytków zależy od *Maszyny Wirtualnej*. W *Maszynie Wirtualnej* ***HotSpot*** dostarczanej wraz z pakietem *Java 2 SDK*, zastosowano niszczenie ***pokoleniowe***. Polega ono na tym, że w chwili gdy zabraknie pamięci operacyjnej, niszczy się ***zbędne*** obiekty, a w każdym z pozostałych zwiększa się ***licznik przetrwań***.

Podczas kolejnych odzyskiwań nieużytków, niszczy się tylko obiekty o **najniższych** licznikach przetrwań. Dzięki takiej strategii, odzyskiwanie nieużytków jest znacznie **szybsze**, ponieważ nie musi dotyczyć **wszystkich** zbędnych obiektów.

Zamiast domyślnego trybu pokoleniowego można użyć trybu ***przyrostowego***. W takim przypadku, obiekty dzieli się na **grupy**, a nieużytki odzyskuje tylko z niezbędnej liczby grup. Może to zwiększyć wrażliwość programu na zdarzenia zewnętrzne.

Wyjątki

Wyjątkiem jest **obiekt** klasy pochodnej od **Throwable** (zazwyczaj dziedziczącej po klasie **Exception** albo **Error)**. Jest wysyłany z miejsca zajścia ***sytuacji wyjątkowej***, takiej jak dzielenie całkowite przez **0**, wydanie polecenia poprzez odniesienie **puste**, itp.

O tym, czy funkcja może wysłać wyjątek, można się przekonać, zapoznając z jej ***nagłówkiem***, w którego frazie **throws** można wyszczególnić listę nazw klas wysyłanych wyjątków.

Wyszczególnione **muszą** być tylko wyjątki ***sprawdzane***. Wyjątki ***nie-sprawdzane***, opisane przez klasy **RuntimeException** i **Error** oraz ich pochodne (w szczególności wyjątek klasy **IllegalArgumentException**) **nie muszą** być wyszczególnione, ale informacja o możliwości wysłania wyjątku powinna być podana w słownym opisie funkcji.

```
public double sqrt(double value)
     throws IllegalArgumentException
{
     // ...
     return Math.sqrt(value);
}
```

Wywołanie funkcji, która może wysłać wyjątek **sprawdzany** musi być zawarte w bloku instrukcji ***try*** (por. dalej) albo nagłówek funkcji z której jest wywoływana, musi zawierać frazę *throws* z nazwą klasy tego wyjątku.

W szczególności, ponieważ podczas wykonywania następującego polecenia **readLine**, może zostać wysłany wyjątek klasy **IOException**, więc **niepoprawną** funkcję

```
public String read(BufferedReader br)
{
     return br.readLine();
}
```

należy zapisać na przykład jako

```
public String read(BufferedReader br)
{
     try {
          return br.readLine();
     }
     catch(IOException e) {
          return "";
     }
}
```

albo jako

```
public String read(BufferedReader br)
     throws IOException
```

```
{
    return br.readLine();
}
```

Dwie ostatnie funkcje **nie są** równoważne. Ilustrują one jedynie formalnie poprawne sposoby wywoływania funkcji wysyłających wyjątki sprawdzane.

Przygotowanie obsługi

Przygotowanie obsługi wyjątku polega na umieszczeniu wywołania funkcji z której wyjątek **może** być wysłany, w ciele instrukcji ***try*** zawierającej frazę *catch* z nazwą klasy wyjątku (albo z nazwą jej klasy bazowej).

```
try {
 // ...
 double root = sqrt(144);
 // ...
}
catch(IllegalArgumentException e) {
    // ...
}
```

Instrukcja ***try*** może zawierać więcej niż jedną frazę *catch*. Ma to zastosowanie wówczas, gdy podczas wykonywania instrukcji frazy **try**, może być wysłany więcej niż jeden wyjątek.

Instrukcja ***try*** może się kończyć frazą *finally*. Jej instrukcje są wykonywane wówczas po zakończeniu wykonywania instrukcji frazy *try*. Odbywa się to **niezależnie** od tego, czy podczas wykonywania frazy *try* został, czy nie został, wysłany wyjątek.

```
try {
    root = sqrt(a / b);
    // ...
}
catch(ArithmeticException e) {
    // ...
}
catch(IllegalArgumentException e) {
    // ...
}
finally {
    // ...
}
```

Odebranie wyjątku

Jeśli **przygotowano** obsługę wyjątku, a wyjątek zostanie **wysłany**, to w ramach ***odebrania*** wyjątku zaniecha się dalszego wykonywania instrukcji frazy *try* i **w zamian** wykona instrukcje frazy *catch* obsługującej wyjątek, a następnie wykona instrukcje frazy *finally*.

Jeśli **nie przygotowano** obsługi wyjątku, a wyjątek zostanie wysłany, to zaniecha się dalszego wykonywania instrukcji frazy *try*, a po wykonaniu instrukcji frazy *finally* **niejawnie** wyśle ten wyjątek z miejsca wywołania funkcji, w której nie odebrano wyjątku.

Jeśli ***propagacja*** wyjątku spowoduje wyjście sterowania poza program, to dalsze wykonywanie programu zostanie **zaniechane**.

Wysłanie wyjątku

Do **jawnego** wysłania wyjątku można użyć instrukcji **throw** ***ref*****;** w której ***ref*** jest odnośnikiem do obiektu wyjątku.

Wejście-wyjście

Operacje wejścia-wyjścia wykonuje się za pośrednictwem ***strumieni***. Po skojarzeniu strumienia z plikiem, polecenia wydawane strumieniowi są w istocie wykonywane na pliku.

Predefiniowane strumienie **System.in** i **System.out** umożliwiają wprowadzanie danych z ***klawiatury*** i wyprowadzanie ich na ***ekran***.

Wprowadzanie z klawiatury

Do wprowadzenia ***znaku*** z klawiatury służy polecenie **read**

```
System.in.read()
```

W miejscu jego wykonania jest dostarczany *Unikod* znaku (jako dana typu **int**).

```
int chr = System.in.read();
```

W celu wprowadzenia ***wierszy***, należy utworzyć strumień reprezentujący klawiaturę

```
BufferedReader br;
br = new BufferedReader(
          new InputStreamReader(System.in)
);
```

a następnie wprowadzać wiersze wydając polecenia **readLine**

```
String line = br.readLine();
```

W niektórych *Systemach*, wprowadzenie z klawiatury znaku *Ctrl-Z* (albo *Ctrl-Y*) powoduje, że **read** dostarcza **-1**, a **readLine** dostarcza **null**.

Wyprowadzanie na ekran

Wyprowadzanie znaków na ekran odbywa się za pomocą poleceń opisanych przez funkcje **print** i **println**. Pierwsza z nich wyprowadza argument jako łańcuch znaków. Druga wyprowadza dodatkowo znak końca wiersza (**'\n'**).

```
System.out.println(
    "sqrt(144) = " + Math.sqrt(144)  // sqrt(144) = 12
);
```

Wprowadzanie z pliku

W celu wprowadzenia ***wierszy***, należy utworzyć strumień reprezentujący plik

```
FileReader fr =                 // strumień
    new FileReader(fileName);   // nazwa pliku
BufferedReader br =             // strumień buforowany
    new BufferedReader(fr);
```

a następnie wprowadzać wiersze wydając polecenia **readLine**

```
String line = br.readLine();
```

Po zakończeniu wprowadzania, **zaleca się** strumień ***zamknąć***, wydając polecenie **close**

```
br.close();
```

Polecenie **readLine**, wydane w pozycji przed końcem pliku dostarcza **null**.

Wyprowadzanie do pliku

W celu wyprowadzenia łańcucha ***znaków***, należy utworzyć strumień reprezentujący plik

```
FileWriter fw =                 // strumień
    new FileWriter(fileName);   // nazwa pliku
BufferedWriter bw =             // strumień buforowany
    new BufferedWriter(fw);
```

a następnie wyprowadzać łańcuchy znaków wydając polecenia **write**

```
bw.write(string);           // łańcuch
```

Po zakończeniu wyprowadzania, **należy** strumień ***zamknąć***, wydając polecenie **close**

```
bw.close();
```

Brak zamknięcia może spowodować, że niektóre, ostatnio wyprowadzone znaki, nie zostaną ***wymiecione*** z bufora strumienia do pliku.

Aplikacje

Aplikacją jest program ***samodzielny***, a więc taki, którego wykonanie nie wymaga na przykład ***przeglądarki***.

Każda aplikacja musi zawierać ***funkcję główną***. Funkcja główna musi być składnikiem klasy **publicznej** i mieć postać

```
public static void main(String[] args)
{
    // ...
}
```

A więc musi być funkcja **publiczną**, **statyczną** i **bez-rezultatową** o parametrze typu **String[]**.

Jeśli funkcja główna zostanie zawarta na przykład w klasie **Program** pakietu **jbPack**, a *Maszyna Wirtualna* jest emulowana przez program **java.exe**, to wydanie (np. w systemie *MS-DOS*) polecenia

```
java jbPack.Program Ewa Iza Jan
```

spowoduje przypisanie elementom tablicy identyfikowanej przez ***args***, odnośników do obiektów klasy **String**, reprezentujących takie same łańcuchy, jakie reprezentują literały **"Ewa"**, **"Iza"** i **"Jan"**.

W szczególności wykonanie następującego programu z argumentami jak powyżej, spowoduje wyprowadzenie napisu **Iza**.

```
package jbPack;

class Program {

    public static void main(String[] args)
    {
        System.out.println(
            args[1]
        );
    }
}
```

Szybki start

Prosty program składa się z ***określenia pakietu***, z ***poleceń importu*** oraz z ***klasy***, w której występuje ***funkcja główna***.

Określenie pakietu związuje klasę (np. **Program**) z wybranym pakietem (np. **jbPack**), polecenia importu wyszczególniają nazwy klas (np. **Graphics**), na które można się powoływać bez specyfikowania ich przynależności do pakietów (np. **Graphics** zamiast **java.awt.Graphics**), a funkcja główna (zawsze **main**) jest opisem czynności wykonywanych przez program.

W dowolnym miejscu programu można umieścić ***komentarz*** zaczynający się od pary ***ukośników*** (**//**). W takim wypadku, napis ***od-ukośników-do-końca-wiersza*** jest uznawany za **niebyły**.

Jeśli pewna klasa jest zadeklarowana jako ***publiczna*** (**public**) i ma nazwę ***Name***, to musi być umieszczona w pliku ***Name*.java**.

Następujący program jest ***szkieletem***, który można wypełnić konkretnymi deklaracjami i instrukcjami.

Szkielet zawiera kilka przydatnych poleceń importu. Z przytoczonych dalej programów będą usuwane te, które okażą się zbyteczne.

```
package jbPack;         // określenie pakietu

import javax.swing.*;   // polecenia importu
import java.util.*;
import java.io.*;

public
class Program {

        // funkcja główna
    public static void main(String[] args)
    {
        new Program();
    }

    // miejsce na deklaracje

    public Program()
    {
        // miejsce na instrukcje
    }
}
```

dla dociekliwych

Funkcja główna ma jeden parametr typu **String[]**, o zwyczajowej nazwie **args**. Za jego pośrednictwem można uzyskać dostęp do ***argumentów programu***.

Jeśli skompilowany program odpalono za pomocą polecenia

```
java jbPack.Program arg0 arg1 ... argN
```

w którym ***argi*** jest ***i***-tym argumentem, to parametr **args** zostanie zainicjowany w taki sposób, że wyrażenie **args.length** dostarczy ***liczbę argumentów***, a **args[*i*]** dostarczy odniesienie do obiektu łańcuchowego reprezentującego argument ***i***.

Sposób odpalenia programu oraz przekazania mu argumentów, zależy od użytego *Środowiska Projektowego*. W środowiskach graficznych argumenty podaje się w oknie dialogowym, a program odpala przez kliknięcie ikony paska narzędziowego.

W szczególności, jeśli następująca aplikacja zostanie wywołana za pomocą polecenia

```
java jbPack.Program Ewa Iza
```

to wyprowadzi na konsolę napis

```
Ewa
Iza
```

Program ma **2** argumenty, z których **Ewa** jest argumentem **0**, a **Iza** jest argumentem **1**.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Program(args);
    }

    public Program(String[] args)
    {
        int count = args.length;
        for (int i = 0; i < count ; i++) {
            System.out.println(args[i]);
        }
    }
}
```

Otoczenie

Komunikacja programu z ***otoczeniem*** odbywa się zazwyczaj za pomocą ***dialogów***, ale do wyprowadzenia większej liczby wyników można wykorzystać ***konsolę*** (w środowiskach graficznych implementowaną jako **okno** albo ***panel*** wchodzący w skład okna).

W celu wyprowadzenia na konsolę wartości wyrażenia ***exp*** należy wykonać instrukcję

```
System.out.println(exp);
```

Każdy z argumentów funkcji **println** (np. **127**) jest wówczas wyprowadzany w osobnym wierszu.

```
System.out.println("Hello " + "Jan");  // Hello Jan
System.out.println(127);               // 127
System.out.println();                  // pusty wiersz
```

Następująca aplikacja wyświetla dialog, w którym pyta o imię użytkownika, a po otrzymaniu odpowiedzi (np. **Jan**), wyprowadza pozdrowienie (**Hello Jan**).

```
package jbPack;

import javax.swing.*;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

        // deklaracja zmiennej
    private String name;

    public Program()
    {
            // zapytanie o imię
        name = JOptionPane.showInputDialog("Enter your name");

        if (name == null)    // jeśli nie podano imienia
            System.exit(0);  // zakończenie wykonywania
        else                 // w przeciwnym razie
                // wyświetlenie pozdrowienia
            System.out.println("Hello " + name);
            // zakończenie wykonywania programu
        System.exit(0);
    }
}
```

Zmienne

Wykonanie programu polega na przetwarzaniu ***zmiennych***.

Zmienna jest **obszarem pamięci** do przechowywania ***danych***. W szczególności zmienna typu **int** przechowuje dane ***całkowite*** (np. o wartości **–127**), zmienna typu **double** przechowuje dane ***rzeczywiste*** (np. o wartości **12.4**), a zmienna typu **String** przechowuje dane, które stanowią odniesienia do ***obiektów*** reprezentujących ***łańcuchy*** znaków (np. o wartości **"Jan"**).

W celu przechowania danej, należy zmienną ***zainicjować*** albo ***przypisać*** jej daną przechowywaną w innej zmiennej.

W Javie **nie ma** różnicy między inicjowaniem i przypisywaniem. Mimo to, w celu nawiązania do innych języków programowania, określenie inicjowanie będzie używane w kontekście deklarowania zmiennych i wywoływania funkcji.

```
int value = 10;  // zainicjowanie zmiennej
value = 12;      // przypisanie danej

String name = "Jan";
System.out.println(name);  // Jan
```

dla dociekliwych

W zmiennej można przechować tylko taką daną, której typ jest ***zgodny*** z typem zmiennej. W szczególności w zmiennej typu **double** nie można przechować danej typu **int** i odwrotnie.

Ponieważ rygorystyczne egzekwowanie tego wymagania znacznie utrudniałoby pisanie programów, w pewnych sytuacjach dopuszcza się **niejawne** wykonywanie ***konwersji*** zmiennej, w zmienną innego typu.

W szczególności, w zasięgu deklaracji

```
int fixed;    // zmienna typu całkowitego
double real;  // zmienna typu rzeczywistego
```

wykonanie instrukcji

```
fixed = 12;
```

nie wymaga niejawnych zabiegów, bo **12** jest nazwą zmiennej typu **int**, ale wykonanie (w zasadzie niepoprawnej) instrukcji

```
real = fixed;
```

zostaje niejawnie zastąpione wykonaniem instrukcji

```
real = (double)fixed;
```

w której występuje konwersja zmiennej typu **int** na zmienną typu **double** (zainicjowaną taką samą wartością jaką ma zmienna **fixed**).

Konwersja z typu **double** do typu **int** nie może być wykonana **niejawnie**, ponieważ mogłoby to spowodować utratę wartości.

Stałe

Zmienne zadeklarowane ze specyfikatorem **final** (zmienne ***finalne***) są ***nie-modyfikowalne*** i jako takie będą nazywane ***stałymi***.

Zdefiniowanie stałej może nastąpić podczas deklarowania albo przypisania. Taką czynność można wykonać co najwyżej **jednokrotnie**.

```
final double Pi;
Pi = 3.14;  // dobrze
Pi = 3.14;  // błąd
```

Jeśli zmienną finalną jest zmienna **lokalna**, to nie przestaje ona istnieć po zakończeniu wykonywania bloku, w którym ją zadeklarowano. Może to skutkować jednoczesnym istnieniem **wielu** zmiennych finalnych opisanych przez tę samą deklarację.

Literały

Literałem jest napis reprezentujący **stałą,** z którego postaci wynika jej wartość. Typowymi literałami są ***liczby*** (np. **12**) i ***łańcuchy-znakowe*** (np. **"Ewa"**).

Uwaga: Znaki specjalne, w tym znak ***nowego wiersza***, ***ukośnik*** i ***cudzysłów***, zapisuje się za pomocą **pary** znaków, z których pierwszy jest ***ukośnikiem*** (np. **\n**, ****, **\"**).

```
String name = "J\"B";
System.out.println(name);  // J"B
```

Typy

Każda zmienna jest określonego ***typu***. Typy dzielą się na ***pierwotne*** i ***odnośnikowe***. Typami pierwotnymi są typy ***całkowite*** (np. **int**), ***znakowe*** (**char**), ***rzeczywiste*** (np. **double**) i ***orzecznikowe*** (**boolean**), a typami odnośnikowymi pozostałe typy języka (np. **String**).

Typy całkowite i rzeczywiste oraz typ znakowy są typami ***numerycznymi***. Rozmiary zmiennych typów pierwotnych są **niezależne** od platformy sprzętowej.

Typy całkowite

Zmienna typu ***całkowitego*** (**int**, **long**, **byte**, **short**) służy do przechowywania danych całkowitych. Zmienne typu **int** są **4**-bajtowe, a zmienne typu **long** są **8**-bajtowe. Bardzo rzadko używa się **1**-bajtowych zmiennych typu **byte** oraz **2**-bajtowych zmiennych typu **short**.

Zaleca się posługiwanie zmiennymi typu **int**, zapewniającymi możliwość reprezentowania wartości całkowitych z przedziału

-2,147,483,648 .. +2,147,483,647

```
int value = -12;
System.out.println(value);  // -12
```

Typ znakowy

Zmienna typu ***znakowego*** (**char**) służy do przechowywania ***kodów znaków***. Zmienne typu **char** są **2**-bajtowe i umożliwiają m.in. reprezentowanie znaków wszystkich krajów europejskich.

Do reprezentowania kodów znaków stosuje się ***Unikod***.

```
char star = '*';
System.out.println(star);   // *
```

Typy rzeczywiste

Zmienna typu ***rzeczywistego*** (**double**, **float**) służy do przechowywania danych nie-całkowitych. Zmienne typu **double** są **8**-bajtowe, a zmienne typu **float** są **4**-bajtowe.

Zaleca się posługiwanie zmiennymi typu **double**, zapewniającymi możliwość reprezentowania wartości z przedziału

-1.8×10^{308} .. 1.8×10^{308}

Operacje na zmiennych typów rzeczywistych **nigdy** nie prowadzą do zaistnienia sytuacji wyjątkowych. W szczególności dzielenie rzeczywiste przez **0.0** powoduje dostarczenie wartości **Double.NaN** (**N**ot-**a**-**N**umber).

```
double value = -12.4567e2;
System.out.println(value);  // -1245.67
```

Dane rzeczywiste (np. **4.35**) są reprezentowane w sposób **przybliżony**. Powoduje to, że na przykład wyrażenie **(int)(4.35 * 100)** ma wartość **434**, a nie **435**. Wynika to stąd, że na przykład w komputerze *Pentium* wyrażenie **4.35 * 100** ma wartość

434.99999999999994, a nie **435.0**. Dlatego przed przekształceniem liczby rzeczywistej w całkowitą, do czego służy operator **(int)**, warto dodać do niej **0.5**. Wówczas **(int)(4.35 * 100 + 0.5)** będzie miało oczekiwaną wartość **435**.

Typy orzecznikowe

Zmienna typu ***orzecznikowego*** (**boolean**) służy do przechowywania danych o wartościach **true** (*prawda*) i **false** (*fałsz*). Zmienne typu **boolean** są **1**-bajtowe.

```
int value = 20;
boolean isGreater = value > 0;
System.out.println(isGreater);  // true
```

Typy odnośnikowe

Zmienna typu ***odnośnikowego*** (np. **String**) jest ***odnośnikiem***, któremu można przypisywać ***odniesienia*** do obiektów. Jeśli odnośnikowi przypisze się odniesienie ***puste*** (wyrażone za pomocą literału **null**), to odnośnik nie odnosi się do żadnego obiektu. Zmienne typów odnośnikowych są **4**-bajtowe.

W szczególności, po wykonaniu instrukcji

```
String name;
name = "Ewa";
```

odnośnik **name** odnosi się do obiektu reprezentującego łańcuch znaków **"Ewa"**, a po wykonaniu instrukcji

```
name = null;
```

nie odnosi się do żadnego obiektu.

Typy łańcuchowe

Zmienna typu ***łańcuchowego*** (**String**, **StringBuffer**) jest odnośnikiem do obiektu reprezentującego łańcuch znaków.

W odróżnieniu od obiektów klasy **StringBuffer**, które są „zwykłymi" zmiennymi, obiekty klasy **String** są **stałymi**. Każdy taki obiekt jest fabrykowany w miejscu wystąpienia stałego wyrażenia łańcuchowego (np. **„Ewa"** albo **„E" + „wa"**) oraz w miejscu użycia fabrykatora (np. **new String(„Ewa")**).

Sklejanie łańcuchów

Posługując się operatorem **+** (*plus*), można za pośrednictwem odnośników do łańcuchów wykonywać ***sklejanie łańcuchów***. Odbywa się to **niejawnie**, za pomocą poleceń **append** klasy **StringBuffer**, a w miejscu sklejenia jest dostarczane odniesienie do obiektu klasy **String**.

Operacja sklejania może dotyczyć zarówno **numeryków** jak i **łańcuchów**. Rezultatem sklejenia, w którym jeden argument jest łańcuchem, a drugi numerykiem, jest łańcuch składający się z wszystkich znaków argumentu-łańcucha oraz ze znaków liczby stanowiącej wartość argumentu-numeryku.

W szczególności wyrażenie

```
"Ewa" + 2 + 4
```

jest niejawnie przekształcane w

```
new StringBuffer().
    append("Ewa").append(2).append(4).
    toString()
```

co powoduje dostarczenie odniesienia do łańcucha **"Ewa24"**.

Jeśli numerykiem jest zmienna **num**, na przykład o wartości **-12.4**, to wyrażenie **"" + num** jest odnośnikiem do obiektu reprezentującego łańcuch **"-12.4"**. Ta właściwość sklejania jest często stosowana do przekształcania numeryków w łańcuchy.

Porównywanie łańcuchów

Do porównywania łańcuchów nie zawsze nadają się operatory **==** (równe) i **!=** (nie-równe), ponieważ orzekają one tylko o tym, czy łańcuchy znajdują się w tym samym miejscu pamięci (podczas gdy pamięć przydzielana łańcuchom klasy **String**, zapisanym za pomocą stałych wyrażeń łańcuchowych jest **rozłączna** z pamięcią przydzielaną pozostałym łańcuchom). Dlatego bezpiecznym sposobem porównania łańcuchów jest użycie polecenia **equals**.

```
"Ewa" == "E" + "w" + "a"                    // true
"Ewa" == new String("Ewa")                  // false
new String("Ewa") == new String("Ewa")      // false
new String("Ewa").equals(new String("Ewa")) // true
```

Funkcje łańcuchowe

Jeśli przyjąć, że ***str*** jest odnośnikiem do obiektu łańcuchowego, to temu obiektowi można wydawać m.in. następujące polecenia

***str*.charAt**(pos)
Polecenie dostarczenia kodu znaku, który w łańcuchu występuje na pozycji ***pos*** (jeśli w łańcuchu nie ma takiej pozycji, to jest wysyłany wyjątek klasy **IndexOutOfBoundsException**).

```
np. int chr = "Kaja".charAt(2);           // j
```

***str*.indexOf**(chr)
Polecenie dostarczenia indeksu pierwszego znaku o kodzie ***chr*** występującego w łańcuchu (jeśli w łańcuchu nie ma takiego znaku, to jest dostarczane **-1**).
np. int pos = "Kaja".indexOf('a'); // 1

***str*.indexOf**(substr)
Polecenie dostarczenia indeksu pierwszego znaku, od którego w łańcuchu zaczyna się podłańcuch ***substr*** (jeśli w łańcuchu nie ma takiego podłańcucha, to jest dostarczane **-1**).
np. int pos = "Kaja".indexOf("ja"); // 2

***str*.substring**(int from, int to)
str.substring(int from)
Polecenie dostarczenia podłańcucha, począwszy od znaku o indeksie ***from*** aż do znaku o indeksie ***to*-1** włącznie; domyślnie: do ostatniego znaku (jeśli w łańcuchu nie ma pozycji od ***from*** do ***to*-1** włącznie, to jest wysyłany wyjątek klasy **IndexOutOfBoundsException**).
np. String sub = "Kaja".substring(1, 3); // "aj"

***str*.trim**()
Polecenie dostarczenia łańcucha pozbawionego początkowych i końcowych znaków o kodach **0x20** i mniejszych (w szczególności usunięcie początkowych i końcowych ***spacji***, ***tabulacji*** i ***znaków sterujących***).
np. String string = " Ewa ".trim(); // "Ewa"

Deklaracje

Każda zmienna musi być ***zadeklarowana***. Deklaracja zawiera opis ***typu*** zmiennej (np. **int**) oraz jej ***identyfikator*** (np. **value**). Podczas deklarowania można ***zainicjować*** zmienną daną początkową (np. **0**).

Zmienne deklarowane **poza** funkcjami są domyślnie inicjowane danymi o wartościach **0** (numeryczne), **false** (orzecznikowe) i **null** (odnośnikowe).

```
public void fun(int par)
{
    int value1 = 10, value2;
    System.out.println(value1);  // 10
    System.out.println(value2);  // błąd
    // ...
}
```

Operacje

Każde wyrażenie zawierające ***operatory*** (np. **a + b**) jest zapisem ***czynności*** określonych przez ***operacje***.

Najczęściej stosowaną operacją jest **przypisanie**. Każdej zmiennej zadeklarowanej w ciele funkcji należy przed pierwszym użyciem, **jawnie** przypisać daną początkową.

```
public void fun(int par)
{
    int value = 10;
    System.out.println(value);   // 10
    value = 20;
    System.out.println(value);   // 20

    String name;
    name = "Ewa";
    System.out.println(name);   // Ewa

    double value;
    System.out.println(value);  // błąd
    // ...
}
```

Przypisania

`width = height`	dostarczenie wartości **width** po skopiowaniu do niej **height**
`age += 2`	dostarczenie wartości **age** po zwiększeniu **age** o **2**
`age *= 3`	dostarczenie wartości **age** po pomnożeniu **age** przez **3**

Zwiększenia i zmniejszenia

`width++`	dostarczenie wartości **width**, a następnie zwiększenie **width** o **1**
`--height`	dostarczenie wartości **height**, ale dopiero po zmniejszeniu **height** o **1**
`width--`	dostarczenie wartości **width**, a następnie zmniejszenie **width** o **1**
`++height`	dostarczenie wartości **height**, ale dopiero po zwiększeniu **height** o **1**

Dodawania i odejmowania

`width + 2`	dostarczenie sumy **width** i **2** (ale bez zmiany **width**)
`height - 1`	dostarczenie różnicy **height** i **1** (ale bez zmiany **height**)

Mnożenia, dzielenia, reszty

`width * height`	dostarczenie iloczynu **width** i **height**
`age / 2`	dostarczenie ilorazu **age** i **2** (dla całkowitych, bez ułamka!)
`number % 3`	dostarczenie reszty z dzielenia **number** przez **3** (znak reszty jest taki jak znak dzielnej!)

Porównania numeryków

`age == limit`	dostarczenie **true** tylko wówczas, gdy **age** jest równe **limit**
`age != limit`	dostarczenie **true** tylko wówczas, gdy **age** nie jest równe **limit**
`age > limit`	dostarczenie **true** tylko wówczas, gdy **age** jest większe niż **limit**
`age < limit`	dostarczenie **true** tylko wówczas, gdy **age** jest mniejsze niż **limit**
`age >= limit`	dostarczenie **true** tylko wówczas, gdy **age** jest większe niż **limit** albo mu równe
`age <= limit`	dostarczenie **true** tylko wówczas, gdy **age** jest mniejsze niż **limit** albo mu równe

Porównania odnośników

`ref1 == ref2`	dostarczenie **true** tylko wówczas, gdy **ref1** i **ref2** identyfikuje ten sam obiekt
`ref != null`	dostarczenie **true** tylko wówczas, gdy **ref** identyfikuje tablicę albo obiekt

Porównania łańcuchów

`s1.`**`equals`**`(s2)`	dostarczenie **true** tylko wówczas, gdy łańcuchy identyfikowane przez **s1** i **s2** są takie same
`s1.`**`compareTo`**`(s2)`	dostarczenie wartości: *ujemnej*, **0**, *dodatniej*, jeśli łańcuch identyfikowany przez **s1** jest odpowiednio: *mniejszy*, *równy*, *większy* niż łańcuch identyfikowany przez **s2** (porównanie łańcuchów zastępuje się porównaniem kodów ich pierwszych znaków nie-identycznych, np. łańcuch **"abceee"** jest mniejszy niż **"abda"**).
`s1 == s2`	dostarczenie **true** tylko wówczas, gdy **s1** oraz **s2** identyfikuje **ten sam** łańcuch.
`s1 != s2`	dostarczenie **true** tylko wówczas, gdy **s1** identyfikuje **inny** łańcuch niż **s2** (co nie wyklucza, że łańcuchy mogą być równe!)

Negacje, koniunkcje, alternatywy

`!isMarried`	dostarczenie **true** tylko wówczas, gdy **isMarried** ma wartość **false**
`isBig & isRed`	dostarczenie **true** tylko wówczas, gdy **isBig** oraz **isRed** ma wartość **true**
`isBig | isRed`	dostarczenie **true** tylko wówczas, gdy **isBig** lub **isRed** ma wartość **true**

Operacje bitowe

`~bits`	dostarczenie negacji bitów (`10` → `01`)
`bitsA & bitsB`	dostarczenie koniunkcji bitów (`1100 & 1010` → `1000`)
`bitsA | bitsB`	dostarczenie alternatywy bitów (`1100 | 1010` → `1110`)
`bitsA ^ bitsB`	dostarczenie sumy *modulo*-**2** bitów (`1100 ^ 1010` → `0110`)
`bits << n`	dostarczenie bitów przesuniętych w lewo o **n** pozycji z wstawianiem zer na najniższej pozycji (`10...1` → `0...10`)
`bits >> n`	dostarczenie bitów przesuniętych w prawo o **n** pozycji z powielaniem bitu znaku (`10...01` → `110...0`)
`bits >>> n`	dostarczenie bitów przesuniętych w prawo o **n** pozycji z zerowaniem bitu znaku (`1...01` → `01...0`)

Wybory

`a ? b : c`	jeśli orzeczenie **a** jest prawdziwe, dostarczenie **b**, w przeciwnym razie dostarczenie **c**

Rozpoznania

`ref instanceof` ***`Name`***

dostarczenie orzeczenia o wartości: "*odniesienie przypisane odnośnikowi* ***ref*** *identyfikuje obiekt klasy* ***Name*** *albo obiekt jej klasy pochodnej* ".

Konwersje

`(int)12.8`	dostarczenie wartości **12** (przekształcenie do typu **int**)
`"" + -12`	dostarczenie wartości **"-12"** (przekształcenie do typu **String**)
`"xy" + 2 + 3`	dostarczenie wartości **"xy23"** (przekształcenie do typu **String**)
`"xy" + (2 + 3)`	dostarczenie wartości **"xy5"** (przekształcenie do typu **String**)

Uproszczenia

Każdą operację przypisania o postaci

`a` `=` ***`a`*** `@` ***`b`***

w której `@` symbolizuje jeden z następujących operatorów

```
+  -  *  /  %  <<  >>  >>>  &  |  ^
```

a wyrażenie **a** nie ma skutków ubocznych (co ma na przykład miejsce w **vec[i++]**, gdzie występuje modyfikacja zmiennej **i**), można uprościć do

```
a @= b
```

(np. **v += 2; v &= 1; v <<= 3; v >>>= 5**).

Polecenia

Obiektom identyfikowanym przez odnośniki można wydawać ***polecenia*** (np. **name.charAt(*pos*)**). Skutkiem wydania polecenia jest wykonanie czynności opisanych przez funkcję definiującą polecenie, w tym dostarczenie przez nią danej.

```
String name = "Ewa";
    // dostarczenie znaku na pozycji 1
char chr = name.charAt(1);
System.out.println(chr);   // w

String name = "       Ewa       ";
    // dostarczenie łańcucha pozbawionego
    // początkowych i końcowych spacji
name = name.trim();
System.out.println(name);  // Ewa
```

dla dociekliwych

Nic nie stoi na przeszkodzie ***łączenia poleceń***. W szczególności polecenie

```
name.trim().charAt(1)
```

składa się z poleceń **trim** i **charAt**.

```
System.out.println(
    "  Ewa ".trim().charAt(0)  // E
);
```

Funkcje

Funkcją jest wyodrębniony ***opis czynności***. Czynności opisane w ciele funkcji wykonuje się w miejscu jej ***wywołania***.

Jeśli funkcję zdefiniowano z jednym albo z większą liczbą ***parametrów***, to tuż przed jej wywołaniem wyznacza się wartości odpowiadających im argumentów, po czym parametry traktuje jak zmienne lokalne funkcji, ***zainicjowane*** wartościami argumentów.

Parametry pierwotne

W ciele funkcji, każde odwołanie do parametru typu pierwotnego dotyczy tylko tego parametru i **nie powoduje** zmiany wartości skojarzonego z nim argumentu.

```
public Program()
{
    int number = 10;
    add(number, 2);
    System.out.println(number);    // 10 (sic!)
}

public void add(int variable, int value)
{
    variable += value;
    System.out.println(variable);  // 12
}
```

Parametry odnośnikowe

W ciele funkcji, każde przypisanie danej parametrowi odnośnikowemu nie powoduje zmiany wartości obiektu, do którego odnosi się argument, ale każde inne odwołanie się do parametru, w szczególności użycie go w celu wydania polecenia obiektowi identyfikowanemu przez argument, **może** spowodować zmianę wartości tego **obiektu**.

```
public Program()
{
    String name = "Jan";
    add(name, "B");
    System.out.println(name);      // Jan (sic!)

    StringBuffer name2 = new StringBuffer("Jan");
    add(name2, "B");
    System.out.println(name2);     // JanB
}

public void add(String variable, String value)
{
    variable += value;
    System.out.println(variable);  // JanB
}
```

```
public void add(StringBuffer variable, String value)
{
    variable.append(value);
    System.out.println(variable);  // JanB
}
```

Biblioteki

Wiele przydatnych czynności jest opisanych przez ***funkcje biblioteczne***. W celu wywołania funkcji bibliotecznej związanej z **klasą**, a nie z obiektem, należy podać nazwę klasy oraz nazwę funkcji (np. **String.valueOf(12)**). Spowoduje to wykonanie czynności opisanych przez funkcję i zazwyczaj dostarczenie danej.

W szczególności wykonanie instrukcji

```
String name = JOptionPane.showInputDialog("Enter name");
```

spowoduje wyświetlenie dialogu z napisem **Enter name** zawierającego ***klatkę edycyjną*** do wprowadzenia tekstu.

Jeśli do klatki wpisze się na przykład napis **Ewa** i naciśnie przycisk ***OK***, to zmiennej **name** zostanie przypisane odniesienie do obiektu reprezentującego łańcuch **Ewa**.

W przypadku naciśnięcia przycisku **Cancel** albo kliknięcia ikony zamknięcia dialogu, funkcja **showInputDialog** dostarczy odniesienie **puste**.

```
String name =
    JOptionPane.showInputDialog("Enter name");

if (name == null) {
    System.out.println(
        "Sorry, I do not know your name!"
    );
    System.exit(0);
} else
    System.out.println("Hello " + name);
```

Instrukcje

Instrukcja jest opisem czynności, jakie mają być wykonane na zmiennych. Wiele użytecznych programów można zapisać posługując się tylko kilkoma podstawowymi instrukcjami (w tym instrukcjami **if** i **while**).

Sposób rozmieszczenia instrukcji w wierszach programu jest **dowolny**. Wymaga się jedynie, aby poszczególne ***leksemy***, w tym ***identyfikatory*** (np. **age**), ***literały*** (np. **16**) i ***operatory*** (np. +=), znajdowały się w całości w tym samym wierszu.

Każdą instrukcję kończy **średnik** albo **klamra** zamykająca. Postawienie średnika po nawiasie okrągłym kończącym frazę **if**, **while**, **for** oraz **switch**

```
if (a > b);
    b = a;
```

albo

```
int i;
for (i = 0; i < 3 ; i++);
    System.out.println(i);
```

jest zazwyczaj **błędem**, który może być bardzo trudny do wykrycia.

Instrukcje wyrażeniowe

Instrukcje wyrażeniowe służą do przypisywania danych, wywoływania funkcji oraz do wykonywania operacji zwiększenia i zmniejszenia.

```
value = 2;
value2 = value3 = value * 5 + 2;
name = JOptionPane.showInputDialog("Enter your name");
initial = name.charAt(0);
System.out.println("Hello Jan");
count++;
```

Instrukcje warunkowe

Instrukcje warunkowe służą do podejmowania decyzji na podstawie zawartych w nich orzeczeń.

```
if (speed > 100)       // jeśli szybkość jest większa niż 100
    speed = 40;        // to zmniejsz ją do 40
```

oraz

```
if (speed > limit)     // jeśli speed > limit
    speed = limit;
else                   // w przeciwnym razie
    speed -= 10;
```

Instrukcje grupujące

Instrukcje grupujące służą do przekształcenia sekwencji instrukcji w **jedną** instrukcję. Są przydatne w miejscach, gdzie chce się wykonać więcej niż jedną instrukcję, ale składnia języka zezwala na użycie tylko **jednej** instrukcji.

Wnętrze instrukcji grupującej będzie nazywane ***blokiem***.

```
{ value1 = 10; value2 = 20; }
```

oraz

```
if (speed > limit) {
     System.out.println("Limit exceeded");
     speed = limit;
} else
     speed -= 10;
```

Instrukcje iteracyjne

Instrukcje iteracyjne umożliwiają wielokrotne wykonanie wybranej instrukcji. Jeśli instrukcji do wykonania jest więcej, to należy się posłużyć instrukcją grupującą.

```
     // dla i = 0, dopóki i < 10
     // wyprowadź wartość i
     // po czym zwiększ i o 1
for (int i = 0; i < 10 ; i++) {
     System.out.println(i);
}
```

oraz

```
     // dopóki i < 10
     // wyprowadź wartość i
     // po czym zwiększ i o 1
i = 0;
while (i < 10) {
     System.out.println(i);
     i++;
}
```

Instrukcje wyjątków

Instrukcje wyjątków służą do rozpoznawania i obsługiwania ***sytuacji wyjątkowych*** (takich jak dzielenie przez **0**) oraz do wysyłania wyjątków.

Sytuacje wyjątkowe dzielą się na ***sprawdzane*** i ***nie-sprawdzane***. Jeśli podczas wykonywania funkcji może zajść sytuacja wyjątkowa **sprawdzana**, a nie przewidziano jej obsługi, to kompilator poda stosowny komunikat. Dlatego w kontekście korzystania z funkcji, pamiętanie o sytuacjach sprawdzanych nie jest konieczne.

Sytuacjami wyjątkowymi nie-sprawdzanymi są sytuacje opisane przez klasy **RuntimeException** i **Error** oraz przez ich klasy ***pochodne***. Wszystkie **pozostałe** sytuacje wyjątkowe, w tym te, które są powiązane z wykonywaniem operacji wejścia-wyjścia, są sprawdzane.

Sytuacje sprawdzane

Jeśli sytuacja wyjątkowa ***Name*** należy do kategorii ***sprawdzanych*** (np. **IOException**), to instrukcję, w której może wystąpić należy umieścić w bloku frazy *try* instrukcji wyjątków, a w bloku jednej z fraz *catch* umieścić ***obsługę*** wyjątku.

Jeśli obsługa wyjątku jest pusta, to jest w istocie **zignorowaniem** sytuacji wyjątkowej. A ponieważ zignorowanie wyjątku nie zawsze jest sensowne, więc w bloku pustej frazy *catch* często umieszcza się wywołanie funkcji **printStackTrace**, wyprowadzającej na konsolę informację o kontekście powstania sytuacji wyjątkowej.

```
try {
    Thread.sleep(secs * 1000);
}
catch (InterruptedException e) {
    e.printStackTrace();
    System.exit(0);
}
```

Innym rozwiązaniem jest umieszczenie w nagłówku funkcji wywołującej, frazy *throws* wyszczególniającej ***listę-nazw-wyjątków***.

```
public void dream(int secs)
    throws InterruptedException
{
    Thread.sleep(secs * 1000);
}
```

Sytuacje nie-sprawdzane

Jeśli sytuacja wyjątkowa należy do kategorii ***nie-sprawdzanych*** (m.in. **ArithmeticException**, **NumberFormatException**, **IndexOutOfBoundsException**), ale nie przewidziano jej ***obsługi***, to nastąpi **przymusowe** zakończenie wykonywania programu.

W szczególności, jeśli funkcji **Integer.parseInt**, używanej do przekształcenia łańcucha o postaci liczby (np. **"-127"**) w liczbę (**-127**)

```
String string = "-127";
int number = Integer.parseInt(string);
```

opisanej w dokumentacji jako funkcja, która **może** wysłać wyjątek typu **NumberFormatException**, dostarczy się argumentu, który nie ma postaci liczby (np. **"$12"**), to w przypadku nie przewidzenia obsługi wyjątku, nastąpi **przymusowe** zakończenie wykonywania programu.

Przymusowemu zakończeniu można **zapobiec**, posługując się instrukcją *try* z obsługą wyjątku klasy **NumberFormatException**. W takim wypadku wyjątek zostanie **odebrany** i **obsłużony**.

```
try {
    String string = "$12";
    int number = Integer.parseInt(string);
    System.out.println(number * number);
}
catch (NumberFormatException e) {
    System.out.println("Wrong argument");
```

```
    System.exit(0);
}
```

Następująca aplikacja sumuje swoje argumenty, zakładając, że mają postać **liczb całkowitych** bez znaku albo ze znakiem - (*minus*). Ponieważ nie przewidziano obsługi zdarzenia **NumberFormatException**, więc jeśli taki wyjątek zostanie wysłany, nastąpi przymusowe zakończenie wykonywania programu.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Program(args);
    }

    public Program(String[] args)
    {
        int count = args.length,
            sum = 0;
        for (int i = 0; i < count ; i++) {
            int number = Integer.parseInt(args[i]);
            sum += number;
        }

        System.out.println(
            "The sum of arguments is " + sum
        );
    }
}
```

Wyjątki

Jeśli podczas wykonywania bloku frazy *try* instrukcji wyjątków wystąpi sytuacja wyjątkowa, to **zaniecha się** dalszego wykonywania bloku i **w zamian** wykona blok tej frazy *catch*, którą przystosowano do obsłużenia wyjątku.

Następująca aplikacja sumuje swoje argumenty liczbowe, **pomijając** te, które nie mają postaci **liczb całkowitych** bez znaku albo ze znakiem - (*minus*)

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Program(args);
    }
```

```
    private boolean numberFound;

    public Program(String[] args)
    {
        int count = args.length,
            sum = 0;
        for (int i = 0; i < count ; i++) {
            try {
                int number = Integer.parseInt(args[i]);
                sum += number;
                numberFound = true;
            }
            catch (NumberFormatException e) {
                System.out.println(
                    "The argument No. " + i +
                    " does not look" +
                    " like a number to me!"
                );
            }
        }

        if (numberFound)
            System.out.println(
                "The sum of arguments is " + sum
            );
        else
            System.out.println("No number found");
    }
}
```

Analizator

Dane na których program wykonuje swoje czynności mają niekiedy postać dość złożonych łańcuchów. Aby móc je podzielić na ***słowa***, używa się ***analizatora***.

Słowem jest **spójny** ciąg znaków różnych od spacji. Po każdej stronie słowa może wystąpić dowolna liczba spacji.

Jeśli przyjąć, że analizowany łańcuch jest reprezentowany przez ***line***, to w celu wyodrębnienia w nim poszczególnych słów można zastosować schemat

```
StringTokenizer st =
    new StringTokenizer(line);

while (st.hasMoreTokens()) {
    String item = st.nextToken();
    // operacje na słowie item
}
```

Następująca aplikacja wyznacza średnią arytmetyczną liczb całkowitych bez znaku, zawartych w podanym łańcuchu. Napisy, które nie są liczbami bez znaku (takie jak **Ewa**, **+12**, **-3**) są ignorowane.

```
package jbPack;

import java.util.*;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    private String Numbers = " 10 Iza 20 Ewa 30 ";

    public Program()
    {
        StringTokenizer st =
            new StringTokenizer(Numbers);

        int sum = 0,
            count = 0;

        while (st.hasMoreTokens()) {
            String token = st.nextToken();
            try {
                int value = Integer.parseInt(token);
                if (value >= 0) {
                    sum += value;
                    count++;
                }
            }
            catch (NumberFormatException e) {}
        }

        try {
            System.out.println(
                "Average = " + (sum / count)  // 20
            );
        }
        catch (ArithmeticException e) {
            System.out.println(
               "String does not contain any numbers"
            );
        }
    }
}
```

Środowiska

Najprostszymi ***środowiskami wykonawczymi*** są: okno ***MS-DOS*** i środowisko ***Kawa***. Każde z nich wymaga zainstalowania pakietu ***Java 2 SDK*** (por. Dodatek ***Java 2 SDK***), który można ściągnąć z *Internetu*.

MS-DOS

Przed przystąpieniem do posługiwania się środowiskiem *MS-DOS* należy się upewnić, że parametr **path** zawiera ścieżkę do katalogu z programami **javac**, **java** i **appletviewer** (np. **d:\jdk1.3\bin**). Można to uzyskać, wydając na przykład polecenie

```
path = %path%;d:\jdk1.3\bin
```

Jeśli program źródłowy znajduje się w pliku **Program.java**, to w celu skompilowania go należy wydać polecenie

```
javac -d . Program.java
```

Spowoduje to utworzenie pliku **Program.class** zawierającego *B*-kod programu i umieszczenie go w podkatalogu odpowiadającym nazwie jego pakietu.

Po wykonaniu tej czynności, wykonanie polecenia

```
java jbPack.Program
```

spowoduje ***odpalenie*** programu.

dla dociekliwych

Jeśli program korzysta z ***klasy szkieletowej***, takiej jak **Console.java**, umieszczonej w tym samym katalogu co plik **Program.java**, to w celu przeprowadzenia kompilacji należy wydać polecenie

```
javac -d . Program.java Console.java
```

Kawa

Przed przystąpieniem do posługiwania się środowiskiem *Kawa*, należy zainstalować pakiet ***Java 2 SDK*** i środowisko ***Kawa***, a następnie zainstalować ***Szkielety*** (por. Dodatki: ***JDK 2 SDK***, ***Kawa 5 Enterprise*** i ***Stosowanie szkieletów***).

Po wywołaniu *Kawy*, a przed użyciem nowego szkieletu, należy wyczyścić panel ***projektowy*** i ***edycyjny***.

Po wyczyszczeniu obu paneli należy wydać polecenie ***Project / Open*** i otworzyć ***projekt wzorcowy*** (np. **d:\jbKawa**\jbApplication**jbApplication.kpx**), który znajduje się w pod-katalogu *base***jbKawa**.

W otwartym projekcie, znajdzie się plik **Program.java**. Jego zawartość można **zmodyfikować** albo **zastąpić** programem zaczerpniętym z książki (ich kopie znajdują się na dyskietce w pliku **Programs.doc**).

Katalog z projektem wzorcowym jest ***katalogiem projektowym***. Jeśli pewien program niniejszej książki odwołuje się do ***zasobu*** dostarczonego w katalogu **jbFiles** (np. pliku ***.gif** albo ***.wav**), to przed ***odpaleniem programu*** (ikona ***Run Java***) należy **skopiować** zasób do katalogu projektowego.

Programy

W celu uświadomienia faktu, że

jeśli dla pewnego zestawu testów
program działa zgodnie z oczekiwaniami,
to z tego NIE WYNIKA, że jest poprawny

przedstawiono kolejno ulepszane wersje programu, który w zamierzeniu

1. Wyświetla dialog wejściowy do podania zestawu danych,
   którymi są **liczby** (np. -10 20 +30) i **nie-liczby** (np. Ewa $12).

2. Wyświetla dialog wyjściowy, w którym podaje różnicę
   między liczbą największą i najmniejszą.

3. Reaguje na wszystkie warunki brzegowe,
   m.in. na same nie-liczby, same spacje, itp.

4. Kończy się w chwili zamknięcia dialogu wyjściowego.

Przyjęto z definicji, że dana nie-całkowita (np. **12.4**) jest nie-liczbą.

Rozwiązanie 1

Dla danych **10 30** program powinien wyprowadzić liczbę 20.
Ale wyprowadza 30.
Dlaczego? Bo przyjmuje, że liczbą minimalną jest **0** (początkowa wartość zmiennej **min**).

Dla danych **10 30 Ewa** program powinien wyprowadzić liczbę 20.

Ale kończy się komunikatem:
`java.lang.NumberFormatException: Ewa`
Dlaczego? Bo podczas próby przekształcenia łańcucha **"Ewa"** na liczbę jest wysyłany wyjątek.

Dla danych **-10 0** program poprawnie wyprowadza liczbę 10.
Ale, podobnie jak w poprzednich przypadkach, nie kończy się.
Dlaczego? Bo jeśli program wyświetla dialog, to należy go zakończyć **jawnie** (funkcja **System.exit**).

```
package jbPack;

import javax.swing.*;
import java.util.*;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    public Program()
    {
         String data = JOptionPane.showInputDialog("Enter data");

         StringTokenizer tokens = new StringTokenizer(data);

         int min = 0,
             max = 0;

         while (tokens.hasMoreTokens()) {
              int number = Integer.parseInt(tokens.nextToken());
              if (number > max)
                  max = number;
              if (number < min)
                  min = number;
         }

         JOptionPane.showMessageDialog(
             null, "Result = " + (max - min)
         );
    }
}
```

Rozwiązanie 2

Po usunięciu błędów podanych powyżej

Dla danych **10 Ewa** program poprawnie wyprowadza liczbę **0**,
ale dla danych **Ewa 10** kończy się komunikatem:

```
java.lang.NumberFormatException: Ewa
```

Dlaczego? Bo podczas inicjowania zmiennej **min** jest wysyłany wyjątek.

```
package jbPack;

import javax.swing.*;
import java.util.*;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    public Program()
    {
        String data = JOptionPane.showInputDialog("Enter data");

        StringTokenizer tokens = new StringTokenizer(data);

        int min = Integer.parseInt(tokens.nextToken()),
            max = min;

        while (tokens.hasMoreTokens()) {
            try {
                int number = Integer.parseInt(tokens.nextToken());
                if (number > max)
                    max = number;
                if (number < min)
                    min = number;
            }
            catch (NumberFormatException e) {
            }
        }

        JOptionPane.showMessageDialog(
            null, "Result = " + (max - min)
        );

        System.exit(0);
    }
}
```

Rozwiązanie 3

Po usunięciu błędów podanych powyżej

Dla danych **Ewa Iza** program powinien wyprowadzić informację o braku liczb.
Ale wyprowadza liczbę 0.
Dlaczego? Bo przed wyprowadzeniem wyniku nie upewniono się, że zmienna **numberEntered** ma wartość **true**.

Po zamknięciu dialogu wejściowego w inny sposób jak za pomocą przycisku **OK.** (np. za pomocą klawisza *Esc*) program kończy się komunikatem:
`java.lang.NullPointerException`
Dlaczego? Bo po zamknięciu dialogu nie sprawdzono, czy dostarczono w nim dane.

```
package jbPack;

import javax.swing.*;
import java.util.*;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    public Program()
    {
        String data = JOptionPane.showInputDialog("Enter data");

        StringTokenizer tokens = new StringTokenizer(data);

        boolean numberEntered = false;
        int min = 0, max = 0;

        while (tokens.hasMoreTokens()) {
            try {
                int number = Integer.parseInt(tokens.nextToken());

                if (!numberEntered) {
                    numberEntered = true;
                    min = max = number;
                }

                if (number > max)
                    max = number;
                if (number < min)
                    min = number;
            }
            catch (NumberFormatException e) {}
        }

        JOptionPane.showMessageDialog(
            null, "Result = " + (max - min)
        );

        System.exit(0);
    }
}
```

Rozwiązanie 4

Po usunięciu błędów podanych powyżej

Dla danych **+30 10** program powinien wyprowadzić liczbę 20.
Ale wyprowadza liczbę 0.
Dlaczego? Bo funkcja **Integer.parseInt** nie uznaje napisu **+30** za liczbę.

```
package jbPack;

import javax.swing.*;
import java.util.*;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    public Program()
    {
        String data = JOptionPane.showInputDialog("Enter data");

        if (data == null) {
            JOptionPane.showMessageDialog(
                null, "Dialog closed by Cancel"
            );
            System.exit(0);
        }

        if (data.trim().equals("")) {
            JOptionPane.showMessageDialog(
                null, "No data entered"
            );
            System.exit(0);
        }

        StringTokenizer tokens = new StringTokenizer(data);

        boolean numberEntered = false;
        int min = 0, max = 0;

        while (tokens.hasMoreTokens()) {
            try {
                int number = Integer.parseInt(tokens.nextToken());

                if (!numberEntered) {
                    numberEntered = true;
                    min = max = number;
                }

                max = Math.max(number, max);
```

```
                min = Math.min(number, min);
            }
            catch (NumberFormatException e) {}
        }

        if (numberEntered)
            JOptionPane.showMessageDialog(
                null, "Result = " + (max - min)
            );
        else
            JOptionPane.showMessageDialog(
                null, "No number entered"
            );

        System.exit(0);
    }
}
```

Rozwiązanie poprawne 1

Z analizatorem

```
package jbPack;

import javax.swing.*;
import java.util.*;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    public Program()
    {
        String data = JOptionPane.showInputDialog("Enter data");

        if (data == null) {
            JOptionPane.showMessageDialog(
                null, "Dialog closed by Cancel"
            );
            System.exit(0);
        }

        if (data.trim().equals("")) {
            JOptionPane.showMessageDialog(
                null, "No data entered"
            );
            System.exit(0);
        }

        StringTokenizer tokens = new StringTokenizer(data);
```

```
        boolean numberEntered = false;
        int min = 0, max = 0;
        String token = "";

        while (tokens.hasMoreTokens()) {
            int number;
            try {
                token = tokens.nextToken();
                if (token.charAt(0) == '+')
                    token = token.substring(1);

                number = Integer.parseInt(token);

                if (!numberEntered) {
                    numberEntered = true;
                    min = max = number;
                }

                max = Math.max(number, max);
                min = Math.min(number, min);
            }
            catch (NumberFormatException e) {}
        }

        if (numberEntered)
            JOptionPane.showMessageDialog(
                null, "Result = " + (max - min)
            );
        else
            JOptionPane.showMessageDialog(
                null, "No number entered"
            );

        System.exit(0);
    }
}
```

Rozwiązanie poprawne 2

Bez analizatora

```
package jbPack;

import javax.swing.*;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }
```

```
private boolean numberFound;
private int min, max, number;

public Program()
{
    String data = JOptionPane.showInputDialog("Enter data");

    if (data == null) {
        JOptionPane.showMessageDialog(
          null, "Dialog closed by Esc"
        );
        System.exit(0);
    }

    if (data.trim().equals("")) {
        JOptionPane.showMessageDialog(
            null, "No data entered"
        );
        System.exit(0);
    }

    String token = "";

    while (!data.equals("")) {
        try {
            data = data + ' ';
            int spaceIndex = data.indexOf(' ');
            token = data.substring(0, spaceIndex);
            data = data.substring(spaceIndex).trim();
            improveMinMax(token);
        }
        catch (NumberFormatException e) {
            if (token.charAt(0) == '+')
                improveMinMax(token.substring(1));
        }
    }

    if (numberFound)
        JOptionPane.showMessageDialog(
            null, "Result = " + (max - min)
        );
    else
        JOptionPane.showMessageDialog(
            null, "No number found"
        );

    System.exit(0);
}

public void improveMinMax(String token)
{
    number = Integer.parseInt(token);

    if (!numberFound) {
        numberFound = true;
```

```
            min = max = number;
        }

        max = Math.max(number, max);
        min = Math.min(number, min);
    }
}
```

Podsumowanie testów

Dla danych **10 30** program powinien wyprowadzić liczbę 20.

Dla danych **10 30 Ewa** program powinien wyprowadzić liczbę 20.

Dla danych **Ewa 10 30** program powinien wyprowadzić liczbę 20.

Dla danych **10 Ewa 30** program powinien wyprowadzić liczbę 20.

Dla danych **+30 10** program powinien wyprowadzić liczbę 20.

Dla danych **+30 +10** program powinien wyprowadzić liczbę 20.

Dla danych **Ewa Iza** program powinien wyprowadzić komunikat o braku liczb.

Aplety

Apletem jest program, wykonywany pod nadzorem ***przeglądarki***. Przeglądarce należy dostarczyć ***opis apletu*** określający nazwę programu oraz rozmiary ***pulpitu***: prostokątnego obszaru graficznego do sporządzania wykresów.

Opis apletu

Jeśli program apletu jest opisany przez klasę **Program** zawartą w pakiecie **jbPack**, a rozmiary pulpitu apletu mają wynosić **400** *x* **400** pikseli, to opisem apletu jest

```
<applet  code=jbPack.Program.class
         width=400 height=400>
</applet>
```

Opis apletu umieszcza się w pliku z rozszerzeniem **.html**, na przykład **Index.html**. W takim przypadku odpalenie apletu odbywa się za pomocą dwu-kliknięcia nazwy takiego pliku.

dla dociekliwych

Aplety zaleca się odpalać w środowisku *Kawa*, za pomocą ikony ***Start Run***.

W oknie *MS-DOS* aplet odpala się przez wydanie polecenia

```
appletviewer Index.html
```

a w przeglądarce przez otwarcie pliku **Index.html**.

W celu zapoznania się z ***konsolą apletu*** należy

w przeglądarce *Netscape*
Wydać polecenie ***Communicator / Tools / Java Console***.

w przeglądarce *Internet Explorer*
Wydać polecenie ***Narzędzia / Opcje internetowe / Zaawansowane***, w zakładce ***Ustawienia*** na pozycji ***Microsoft VM*** odhaczyć ustawienie ***Konsola Javy*** włączona, a po zamknięciu i ponownym otwarciu przeglądarki wydać polecenie ***Widok / Java Console***.

Szkielet apletu

Szkielet apletu opartego na klasie **Applet**, znajduje się w pliku **Program.java**, w katalogu *base***jbKawa\jbApplet** (por. Dodatek ***Stosowanie szkieletów***).

W szkielecie umieszczono polecenia importu niezbędne do tworzenia dialogów i posługiwania się kolekcjami.

W celu użycia szkieletu programu apletowego należy się otworzyć projekt

*base***jbKawa\jbApplet\jbApplet.kpx**

a następnie zmodyfikować albo wymienić szkielet.

Zniszczone pliki **Index.html** i **Program.java** można odtworzyć z katalogu *base***jbKawa**, gdzie mają nazwy **Index.html** i **Applet-Program.java**.

W szkielecie zdefiniowano funkcje **init** i **paint**. Funkcja **init** jest wywoływana **jednokrotnie**, bezpośrednio po odpaleniu apletu i służy do wykonania czynności inicjujących. Funkcja **paint** jest wywołana tuż po **init** i może być wywoływana **wielokrotnie**, w chwilach, których nie da się przewidzieć. Należy ją zaprogramować w taki sposób, aby **odtwarzała** cały pulpit (to ważne wymaganie jest niestety często pomijane).

```
package jbPack;

import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
```

```
public
class Program
    extends Applet {

    // deklaracje zmiennych

    public void init()
    {
        // czynności inicjujące
    }

    public void paint(Graphics gDC)
    {
        super.paint(gDC);

        // wykreślanie na pulpicie
    }
}
```

Pulpit apletu

Pulpitem apletu jest prostokątny obszar o rozmiarach określonych w opisie apletu. Lewy-górny narożnik pulpitu ma współrzędne (**0**, **0**). Współrzędne ***x*** pulpitu rosną **w-prawo**, a jego współrzędne ***y*** rosną **w-dół**. Do dostarczenia rozmiarów pulpitu służą funkcje **getWidth** i **getHeight**.

getWidth()
Dostarcza szerokość pulpitu.

getHeight()
Dostarcza wysokość pulpitu.

Wydawanie poleceń

Parametr funkcji **paint** jest odnośnikiem do ***wykreślacza***: obiektu zarządzającego wykreślaniem na pulpicie apletu. Wykreślaczowi można wydawać polecenia wykreślania (np. **drawOval**) albo polecenia ustawiania jego parametrów (np. **setColor**).

Uwaga: Kolory są reprezentowane m.in. przez napisy **Color.red**, **Color.green**, **Color.blue**, **Color.yellow**, **Color.magenta**, **Color.cyan**, **Color.white**, **Color.black**.

gDC.setBackground(*color*);
Ustawia kolor pulpitu na ***color***:

gDC.setColor(*color*);
Przygotowuje wykreślacz do wykreślania w kolorze ***color***:

gDC.setFont(new Font(*face*, *style*, *size*));
Przygotowuje wykreślacz do wykreślania napisów czcionką o kroju ***face*** (np. **"Lucida Bright"**), stylu ***style*** (np. **Font.ITALIC**) i rozmiarze ***size*** (np. **60** punktów).

gDC.drawString(*string*, *x*, *y*);
Wykreśla napis ***string*** (np. "Hello"), w taki sposób, aby lewy koniec domyślnego odcinka, na którym napis **spoczywa** miał współrzędne (***x***, ***y***).

gDC.drawOval(*x*, *y*, *w*-1, *h*-1);
Wykreśla **obrzeże owalu** wpisanego w domyślny prostokąt o lewym-górnym narożniku w (***x***, ***y***) i rozmiarach ***w*** *x* ***h***.

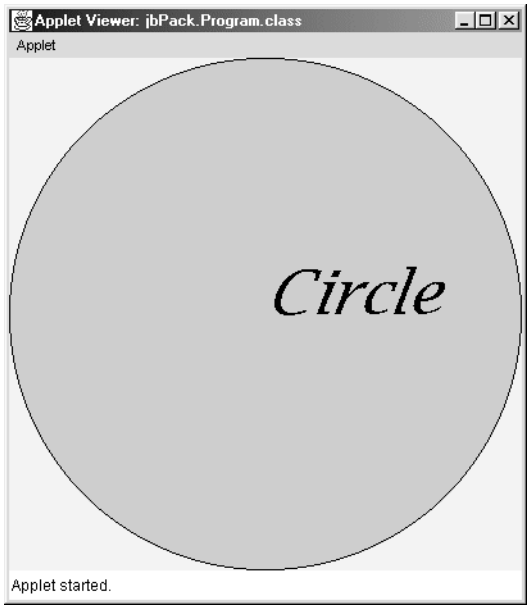
gDC.fillOval(*x*, *y*, *w*, *h*);
Wykreśla **wnętrze owalu** wpisanego w domyślny prostokąt o lewym-górnym narożniku w (***x***, ***y***) i rozmiarach ***w*** *x* ***h***.

gDC.drawRect(*x*, *y*, *w*-1, *h*-1);
Wykreśla **obrzeże prostokąta** o lewym-górnym narożniku w (***x***, ***y***) i rozmiarach ***w*** *x* ***h***.

gDC.fillRect(*x*, *y*, *w*, *h*);
Wykreśla **wnętrze prostokąta** o lewym-górnym narożniku w (***x***, ***y***) i rozmiarach ***w*** *x* ***h***.

Następujący aplet, pokazany na ekranie ***Wydawanie poleceń***, wykreśla na żółtym tle, otoczone czarną obwódką zielone koło, w całości mieszczące się na pulpicie, o tak dobranym promieniu, aby był jak największy. Ponadto, czarną i pochyloną czcionką **Lucida Bright**, o rozmiarze **48** pt, wykreśla napis ***Circle***.

Ekran
Wydawanie poleceń




```
package jbPack;

import java.applet.*;
import java.awt.*;
```

```
public
class Program
    extends Applet {

    private int w, h, d, r;

    public void init()
    {
        setBackground(Color.yellow);

        w = getWidth();
        h = getHeight();
        if (w > h)
            d = h;
        else
            d = w;

        r = d/2;
    }

    public void paint(Graphics gDC)

    {
        super.paint(gDC);

        gDC.setColor(Color.green);
        gDC.fillOval(w/2-r, h/2-r, d, d);
        gDC.setColor(Color.black);
        gDC.drawOval(w/2-r, h/2-r, d-1, d-1);
        gDC.setFont(
            new Font("Lucida Bright", Font.ITALIC, 48)
        );
        gDC.drawString("Circle", w/2, h/2);
    }
}
```

Własny wykreślacz

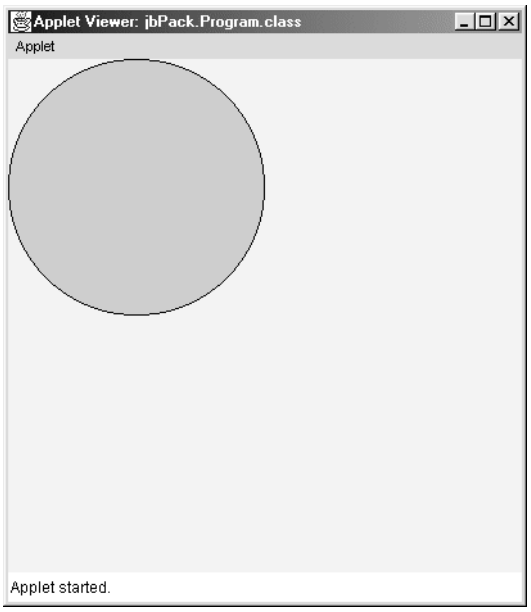
Niekiedy istnieje potrzeba sporządzenia wykresu spoza funkcji **paint**. W takim wypadku przydaje się ***własny wykreślacz***. Odniesienie do własnego wykreślacza dostarcza funkcja **getGraphics**.

`getGraphics()`
Dostarcza odniesienie do odrębnego wykreślacza. Można je przypisać odnośnikowi typu **`Graphics`**.

Użycie własnego wykreślacza jest skuteczne dopiero po podjęciu pierwszego wykonania funkcji **paint**.

Następujący aplet, pokazany na ekranie ***Własny wykreślacz***, wykreśla czarną obwiednię koła za pomocą własnego wykreślacza.

Ekran
Własny wykreślacz




```
package jbPack;

import java.applet.*;
import java.awt.*;

public
class Program
     extends Applet {

     public void init()
     {
          setBackground(Color.yellow);
     }

     public void paint(Graphics gDC)
     {
          super.paint(gDC);

          gDC.setColor(Color.green);
          gDC.fillOval(0, 0, 200, 200);
          Graphics pDC = getGraphics();
          pDC.drawOval(0, 0, 200-1, 200-1);
     }
}
```

Obsługiwanie zdarzeń

Na pulpicie można wykonywać operacje za pomocą myszki. W chwili wykonania takiej operacji jest tworzony ***obiekt zdarzeniowy*** opisujący zaistniałe zdarzenie (m.in. określający współrzędne tego punktu pulpitu, nad którym znajdował się kursor), a następnie jest wywoływana funkcja ***obsługi zdarzenia***.

W celu zapewnienia obsługi, należy dla zdarzeń **mousePressed**, **mouseReleased** i **mouseClicked** wydać polecenie

```
addMouseListener(
    new MouseAdapter() {

        // deklaracje zmiennych
        // definicje funkcji obsługi
    }
);
```

a dla zdarzeń **mouseMoved** i **mouseDragged** polecenie

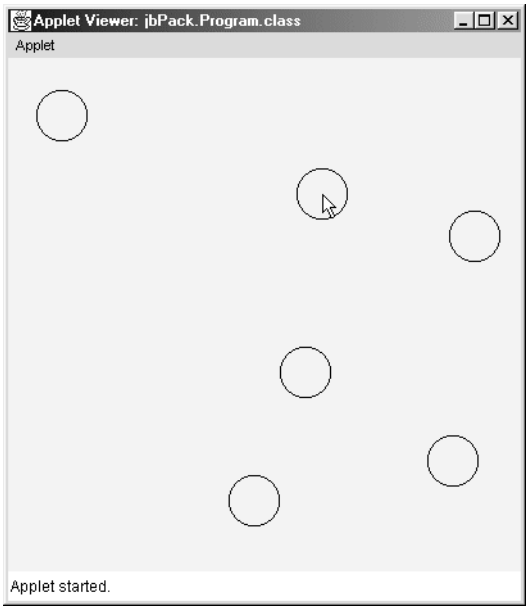
```
addMouseMotionListener(
    new MouseMotionAdapter() {

        // deklaracje zmiennych
        // defnicje funkcji obsługi
    }
);
```

Polecenie obsługi zdarzeń wydaje się z reguły w funkcji **init**.

Następujący aplet, pokazany na ekranie ***Obsługiwanie zdarzeń***, umożliwia wykreślanie czarnych okręgów w otoczeniu punktu kliknięcia.

Z apletu usunięto funkcję **paint**, ponieważ jej ciało było **puste**.

Ekran
Obsługiwanie zdarzeń



```
package jbPack;

import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public
class Program
    extends Applet {

    private int r = 20, d = r*2;

    public void init()
    {
        setBackground(Color.yellow);

        addMouseListener(
            new MouseAdapter() {

                public void mouseClicked(MouseEvent evt)
                {
                    int x = evt.getX(),
                        y = evt.getY();
                    Graphics pDC = getGraphics();
                    pDC.drawOval(x-r, y-r, d-1, d-1);
                }
            }
        );
    }
}
```

Animowanie wykreśleń

Animowanie wykreśleń polega na takim oprogramowaniu obsługi zdarzeń, aby wykonywanie operacji za pomocą myszki powodowało przemieszczanie obiektów graficznych.

Do animowania wykreśleń przydają się polecenia wyrażone przez funkcje **setXORMode** i **setPaintMode**. Umożliwiają one przygotowanie wykreślacza do wykreślania w trybie **XOR** oraz **Paint**.

Wykreślanie w trybie **XOR** charakteryzuje się tym, że **dwukrotne** wykreślenie w tym samym miejscu ekranu, obiektu graficznego w tym samym trybie **XOR**, czyni oba wykreślenia **niebyłymi**, to jest przywraca pierwotny wygląd tła, na którym sporządzono wykres.

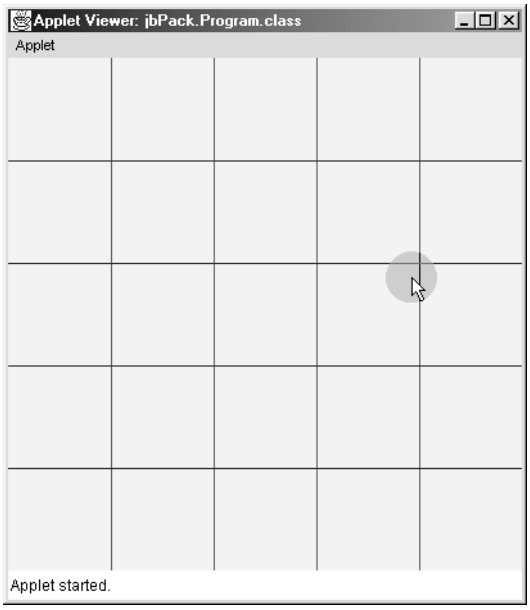
`pDC.setXORMode(color);`
Przygotowuje wykreślacz do wykreślania w trybie **XOR** z kolorem ***color***.

`pDC.setPaintMode();`
Przygotowuje wykreślacz do wykreślania w zwykłym trybie.

Podczas wykreślania w trybie **XOR**, kolor wykreślonego piksela jest **wypadkową** koloru wstawionego do wykreślacza, koloru w punkcie wykreślania i koloru określonego w poleceniu **setXORMode**.

Następujący aplet, pokazany na ekranie ***Animowanie wykreśleń***, wykreśla koło wokół punktu kliknięcia oraz umożliwia przeciąganie go bez niszczenia kolorowej siatki wykreślonej na pulpicie.

Ekran
Animowanie wykreśleń




```
package jbPack;

import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public
class Program
    extends Applet {

    private int N = 5;
    private int w, h, r = 20, d = r*2;
    private Graphics pDC;
    private int xOld, yOld;

    public void init()
    {
        setBackground(Color.yellow);

        w = getWidth();
        h = getHeight();

        pDC = getGraphics();
```

```
        addMouseListener(
            new MouseAdapter() {

                public void mousePressed(MouseEvent evt)
                {
                    int x = xOld = evt.getX(),
                        y = yOld = evt.getY();
                    pDC.setColor(Color.green);
                    pDC.setXORMode(Color.yellow);
                    pDC.fillOval(x-r, y-r, d, d);
                }

                public void mouseReleased(MouseEvent evt)
                {
                    int x = evt.getX(),
                        y = evt.getY();
                    pDC.setPaintMode();
                    pDC.setColor(Color.cyan);
                    pDC.fillOval(x-r, y-r, d, d);
                }
            }
        );

        addMouseMotionListener(
            new MouseMotionAdapter() {

                public void mouseDragged(MouseEvent evt)
                {
                    pDC.fillOval(xOld-r, yOld-r, d, d);
                    int x = xOld = evt.getX(),
                        y = yOld = evt.getY();
                    pDC.fillOval(x-r, y-r, d, d);
                }
            }
        );
    }

    public void paint(Graphics gDC)
    {
        super.paint(gDC);

        gDC.setColor(Color.red);
        for (int i = 0; i < w ; i += w/N)
            if (i != 0)
                gDC.drawLine(i, 0, i, h-1);

        gDC.setColor(Color.blue);
        for (int i = 0; i < h ; i += h/N) {
            if (i != 0)
                gDC.drawLine(0, i, w-1, i);
        }
    }
}
```

Wątki

W nowoczesnych programach często doprowadza się do aktywowania więcej niż jednego ***przepływu sterowania***. Każdy przepływ sterowania przez instrukcje programu jest wówczas realizowany przez odrębny ***wątek***.

Jeśli więcej niż jeden wątek odwołuje się do ***wspólnego zasobu*** (np. zmiennej albo urządzenia), to instrukcje dotyczące tego zasobu muszą być wykonane w ***sekcji krytycznej*** wyznaczonej przez instrukcję **synchronized**.

Instrukcja synchronized

Instrukcja synchronized zawiera odnośnik do obiektu nazywanego ***synchronizatorem***. Jeśli jakiś wątek podejmie wykonywanie bloku instrukcji **synchronized**, to każdy inny watek, który napotka taką instrukcję odwołującą się do tego samego synchronizatora zostanie **zatrzymany** do chwili gdy zakończy się wykonywanie bloku instrukcji synchronizującej.

Dzięki temu uzyskuje się **gwarancję**, że w danej chwili co najwyżej jeden wątek znajduje się w bloku instrukcji synchronizującej posługującej się danym synchronizatorem.

Odniesienie do synchronizatora można otrzymać za pomocą fabrykatora **new Object()**. Przypisuje się je zazwyczaj odnośnikowi typu **Object**.

Utworzenie wątku

Najprostszym sposobem ***utworzenia wątku*** i doprowadzenia do przepływu sterowania przez instrukcje funkcji **run** zawartej w klasie ***Name***, podanej tu w postaci łatwego do zmodyfikowania **szkieletu**, jest opracowanie instrukcji

```
new Name(aArg, bArg, ... , zArg);
```

Zniszczenie wątku nastąpi tuż po zakończeniu wykonywania funkcji **run**.

```
class Name
    extends Thread {

    private aType a;
    private bType b;
    ...
    private zType z;
```

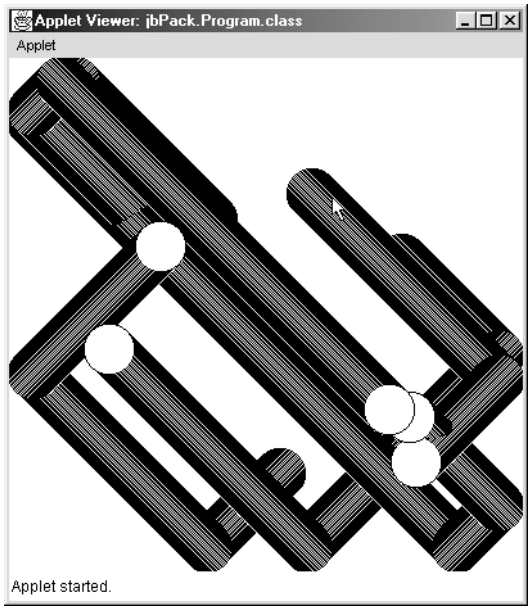
```
    public Name(aType aPar, bType bPar, ... , zType zPar)
    {
        a = aPar;
        b = bPar;
        ...
        z = zPar;

        start();
    }

    // deklaracje zmiennych
    // definicje funkcji

    public void run()
    {
        // przepływ sterowania
    }
}
```

Przykład A

Następujący aplet, pokazany na ekranie ***Kule bilardowe***, tak animuje koła wykreślane wokół punktu kliknięcia pulpitu, że każde kliknięcie powoduje aktywowanie kolejnej animacji, realizującej przepływ sterowania przez instrukcje **tej samej** funkcji **run**.

W celu uproszczenia apletu przyjęto, że kliknięcia będą wykonywane w "dostatecznie" dużej odległości od krawędzi apletu.

Ekran
Kule bilardowe



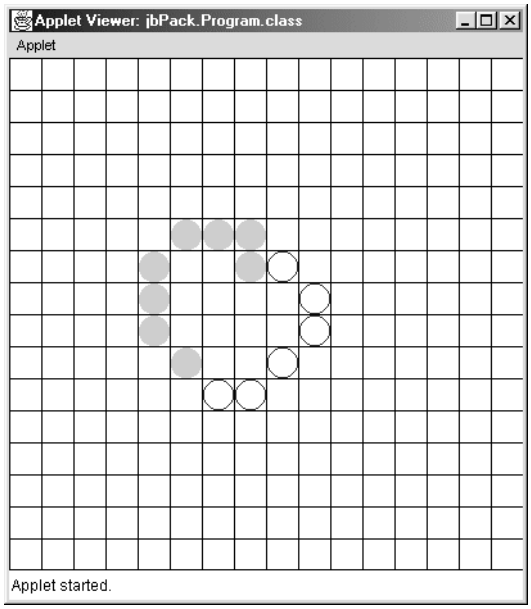
```
package jbPack;

import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public
class Program
    extends Applet {

    private Graphics pDC;
    private Object screenLock = new Object();

    public void init()
    {
        pDC = getGraphics();

        addMouseListener(
            new MouseAdapter() {

                public void mouseReleased(MouseEvent evt)
                {
                    int x = evt.getX(),
                        y = evt.getY();

                    new Animator(x, y, 20);
                }
            }
        );
    }

    class Animator
        extends Thread {

        private int x, y, r;

        public Animator(int xPar, int yPar, int rPar)
        {
            x = xPar;
            y = yPar;
            r = rPar;

            start();
        }

        public void run()
        {
           int w  = getWidth(),
               h  = getHeight(),
               dx = getRandom(),
               dy = getRandom();

           while (true) {
              if (x < r || x > w-r)
                  dx = -dx;

              if (y < r || y > h-r)
                  dy = -dy;
```

```
                x += dx;
                y += dy;

                synchronized (screenLock) {
                    pDC.setColor(Color.white);
                    pDC.fillOval(x-r, y-r, 2*r, 2*r);
                    pDC.setColor(Color.black);
                    pDC.drawOval(x-r, y-r, 2*r-1, 2*r-1);
                }

                try {
                    Thread.sleep(10);
                }
                catch (InterruptedException e) {}
            }
        }
    }

    public int getRandom()
    {
        return (int)(Math.random() * 2) * 2 - 1;
    }
}
```

Przykład B

Następujący aplet, pokazany na ekranie ***Animowanie na torach***, wykreśla na prostokątnej siatce i w otoczeniu punktów kliknięcia, zielone elipsy wypełniające klatki siatki i wytyczające tor ruchu elips. Napisano go w taki sposób, że po kliknięciu w obrębie zapełnionej klatki, zaczyna się animacja czerwonej elipsy poruszającej się po klatkach toru, a w miejscu przez nią zwolnionym jest wykreślane białe koło z niebieską obwódką.

Ekran
Animowanie na torach



```
package jbPack;

import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public
class Program
    extends Applet {

    private final int N = 160;

    private int[] xCell = new int [N],
                  yCell = new int [N];
    private int count;
    private Graphics pDC;
    private int w, h, ww, hh;
    private boolean isAnimating;

    public void init()
    {
        ww = (w = getWidth())  / N;
        hh = (h = getHeight()) / N;

        pDC = getGraphics();

        addMouseListener(
            new MouseAdapter() {

                public void mouseReleased(MouseEvent evt)
                {
                    if (isAnimating)
                        return;

                    int x = evt.getX(),
                        y = evt.getY();
                    int xx = xCell[count] = x / ww,
                        yy = yCell[count] = y / hh;

                    for (int pos = 0; pos < count ; pos++) {
                       if (xCell[pos] == xx && yCell[pos] == yy) {
                            new Animator(count, pos);
                            return;
                        }
                    }
                    pDC.setColor(Color.green);
                   pDC.fillOval(1+xx * ww, 1+yy * hh, ww-1, hh-1);
                    count++;
                }
            }
        );
    }

    class Animator
        extends Thread {

        private int count, pos;
```

```
        public Animator(int count, int pos)
        {
            if (count == 1)
                System.exit(0);

            this.count = count;
            this.pos = pos;

            isAnimating = true;
            start();
        }

        public void run()
        {
            while (true) {
                pDC.setColor(Color.red);
                pDC.fillOval(
                    1+xCell[pos] * ww, 1+yCell[pos] * hh,
                    ww-1, hh-1
                );

                try {
                    Thread.sleep(200);
                }
                catch (InterruptedException evt) {}

                pDC.setColor(Color.white);
                pDC.fillOval(
                    1+xCell[pos] * ww, 1+yCell[pos] * hh,
                    ww-1, hh-1
                );

                pDC.setColor(Color.blue);
                pDC.drawOval(
                    1+xCell[pos] * ww, 1+yCell[pos] * hh,
                    ww-2, hh-2
                );

                pos = (pos+1) % count;
            }
        }
    }

    public void paint(Graphics gDC)
    {
        super.paint(gDC);

        for (int i = 0; i < N ; i++) {
            gDC.drawLine(0, hh * i, w-1, hh * i);  // poziom
            gDC.drawLine(ww * i, 0, ww * i, h-1);  // pion
        }
        gDC.drawLine(0, h-1, w-1, h-1);
    }
}
```

Aplikacje

Programem jest zestaw definicji ***klas*** *i* ***interfejsów*** zapisanych w ***plikach źródłowych***. Każdy plik źródłowy składa się z **określenia pakietu**, **poleceń importu** oraz **definicji klas i interfejsów**.

Jeśli program zawiera funkcję główna (**main**), a jego wykonanie nie wymaga użycia przeglądarki, to jest ***aplikacją***.

Interfejsy są w istocie okrojonymi **klasami**. Dlatego w miejscach, gdzie należałoby użyć niewygodnego określenia "klasa albo interfejs" będzie stosowane określenie ***inter-klasa***.

Komentarze

Jeśli podczas analizy aplikacji napotka się parę znaków **//** (*ukośnik*, *ukośnik*), to wszystkie następne znaki, aż do końca wiersza włącznie, zostaną uznane za ***komentarz***.

Za komentarz zostaną uznane również wszystkie znaki od pary **/*** (*skośnik*, *gwiazdka*) do najbliższej pary ***/** (*gwiazdka*, *skośnik*) włącznie.

```
/*  Aplikacja wyprowadzająca
    pozdrowienie
*/

package jbPack;         // określenie pakietu

import javax.swing.*;  // polecenie importu

public
class Program {

        // funkcja główna
    public static void main(String[] args)
    {
            // wprowadzenie imienia
        String name =
            JOptionPane.showInputDialog(
                "Enter your name"
            );
```

```
            // wyprowadzenie pozdrowienia
        if (name != null)
            JOptionPane.showMessageDialog(
                null, "Hello, " + name
            );

            // zakończenie wykonywania
        System.exit(0);
    }
}
```

Nazwy plików

Nazwą pliku, który zawiera ***publiczną*** inter-klasę ***Name***, musi być ***Name*.java**. Jeśli w pliku znajduje się inter-klasa ***Name***, to po skompilowaniu powstaje z niej plik ***Name*.class**.

Plik ***Name*.class** jest zapisany w ***B-kodzie***. Wykonanie *B*-kodu powierza się ***Maszynie Wirtualnej***. W emulatory *Maszyn Wirtualnych* są wyposażone wszystkie znaczące ***Systemy Operacyjne*** (np. *Windows*) i ***Przeglądarki*** (np. *Netscape*).

Odwołania do inter-klas

Odwołanie do inter-klasy ***Name*** (np. **Graphics**) zawartej w pakiecie ***packet*** (np. **java.awt**), ma w ogólnym przypadku postać ***packet*.*Name*** (np. **java.awt.Graphics**).

Zapis ten można uprościć do ***Name*** jeśli z pakietu ***packet*** importuje się inter-klasę ***Name***

```
import packet.Name;    // np. import java.awt.Graphics;
```

albo gdy importuje się **wszystkie** inter-klasy tego pakietu

```
import packet.*;       // np. import java.awt.*;
```

Inter-klasy pakietu **java.lang** są importowane **domyślnie**. Dlatego odwołania do takich jego inter-klas jak **java.lang.String**, **java.lang.Math**, itp. nie wymagają ani kwalifikowania nazwą pakietu, ani użycia poleceń importu.

Polecenia importu służą do importowania inter-klas, a nie do importowania pakietów. Jeśli w pakiecie jest zawarty inny pakiet, to **nie jest** on importowany nawet wówczas, gdy posłużono się "*gwiazdką*". W szczególności, jeśli chce się importować wszystkie inter-klasy pakietów **java.awt** i **java.awt.event**, to należy użyć poleceń

```
import java.awt.*;
import java.awt.event.*;
```

a nie tylko pierwszego z nich.

Aplikacje konsolowe

Aplikacja ***konsolowa*** posługuje się wyłącznie ***konsolą***, nie ma interfejsu graficznego i nie tworzy innych okienek niż ***dialogi***.

Szkielet aplikacji konsolowej opartej na klasie **Console** (***©JanB***), znajduje się w pliku **Program.java**, w katalogu *base***jbKawa\jbConsole** (por. Dodatek ***Stosowanie szkieletów***).

W szkielecie umieszczono polecenia importu niezbędne do tworzenia dialogów, posługiwania się kolekcjami oraz do wykonywania operacji wejścia-wyjścia.

W celu użycia szkieletu aplikacji konsolowej należy otworzyć projekt

*base***jbKawa\jbConsole\jbConsole.kpx**

a następnie zmodyfikować albo wymienić szkielet.

Zniszczone pliki **Console.java** i **Program.java** można odtworzyć z katalogu *base***jbKawa**, gdzie występują pod nazwami **!Console.java** i **Console-Program.java**.

Szkielet aplikacji konsolowej

```
package jbPack;

import javax.swing.*;
import java.util.*;
import java.io.*;

public
class Program
    extends Console {

    public static void main(String[] args)
    {
        new Program(args);
    }

    public Program(String[] args)
    {
    }
}
```

Następująca aplikacja, posługująca się szkieletem konsolowym, wyprowadza na konsolę ***przypadkową*** liczbę całkowitą z przedziału [**0**, **255**]. Usunięto **wszystkie** polecenia importu, ponieważ aplikacja używa tylko klas pakietu **java.lang**.

Funkcja **random** dostarcza liczbę ułamkową z przedziału [**0**, **1**). Po pomnożeniu jej przez ***n*** i przekształceniu za pomocą operatora **(int)** do typu **int**, otrzymuje się liczbę całkowitą z przedziału [**0**, ***n***-1].

```
package jbPack;

public
class Program
    extends Console {

    public static void main(String[] args)
    {
        new Program();
    }

    private double random;

    public Program()
    {
        random = Math.random();
        System.out.println();
        System.out.println(
            (int)(random * 256)
        );
        System.out.println();
    }
}
```

Wprowadzanie danych

Najprostszym sposobem wprowadzenia danych jest pobranie ich z **konsoli**.

klasa InputStream

```
int read()
Wprowadza z konsoli jeden znak i dostarcza jego kod (wiersz kończą
znaki '\r' i '\n').
np. int chr = System.in.read();
```

Ponieważ wprowadzanie znaków z konsoli jest **buforowane**, więc poszczególne znaki wiersza są dostarczane do aplikacji dopiero **po** naciśnięciu klawisza *Enter*.

dla dociekliwych

Wprowadzanie pojedynczych znaków może być dość uciążliwe. Dlatego znacznie wygodniej jest wprowadzać **wiersze** znaków, a następnie, za pomocą ***analizatora***, dzielić je na **słowa**.

W tym celu można zastosować następujący schemat

```
BufferedReader br;
br = new BufferedReader(
          new InputStreamReader(System.in)
     );
```

```
// ...

try {
    String line = br.readLine();

    // przetworzenie wiersza
}
catch (IOException e) {
    e.printStackTrace();
    System.exit(0);
}
```

Następująca aplikacja wprowadza sekwencję nie-pustych wierszy, sumując zawarte w nich liczby. Kończy się po rozpoznaniu wiersza nie zawierającego danych.

```
package jbPack;

import javax.swing.*;
import java.util.*;
import java.io.*;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    private boolean numberPresent;

    public Program()
    {
        BufferedReader br;
        br = new BufferedReader(
                  new InputStreamReader(System.in)
             );

        System.out.println("Please enter numbers");

        int sum = 0;
        while (true) {
            String line = "";
            try {
                line = br.readLine();
            }
            catch (IOException e) {
                System.out.println(
                    "IO exception"
                );
                System.exit(0);
            }
```

```
            if (line.trim().equals("")) {
                if (numberPresent)
                    System.out.println(
                        "Sum = " + sum
                    );
                else
                    System.out.println(
                        "No numbers present"
                    );
                System.exit(0);
            } else {
                StringTokenizer st = new StringTokenizer(line);
                while (st.hasMoreTokens()) {
                    String word = st.nextToken();
                    try {
                        int number = Integer.parseInt(word);
                        sum += number;
                        numberPresent = true;
                    }
                    catch (NumberFormatException e) {}
                }
            }
        }
    }
}
```

Wyprowadzanie wyników

Do wyprowadzania wyników na konsolę służą polecenia **System.out.print** i **System.out.println**. Pierwsze wyprowadza wartość argumentu, a drugie dodatkowo przemieszcza karetkę do następnego wiersza. Argumentami poleceń mogą być m.in. wyrażenia ***liczbowe*** (np. **-12**), ***znakowe*** (np. **'*'**), ***łańcuchowe*** (np. **"Ewa"**) i ***orzecznikowe*** (np. **speed > Limit**).

klasa PrintStream

```
void println(int value)
void println(long value)
void println(char value)
void println(String value)
void println(boolean value)
void println(Object object)
Wyprowadza na konsolę wartość value albo wartość object.toString()
po czym przemieszcza karetkę do następnego wiersza.
np. System.out.println("Hello" + "World");

void print(String string)
Wyprowadza na konsolę łańcuch string.
np. System.out.print("Hello\n");  // równoważne println("Hello");
```

Następująca aplikacja wyprowadza na konsolę pozdrowienie osoby o imieniu określonym przez argument.

```
package jbPack;

public
class Program
    extends Console {

    public static void main(String[] args)
    {
        new Program(args);
    }

    public Program(String[] args)
    {
        if (args.length == 1)
            println("Hello " + args[0]);
        else
            println("Wrong argument");
    }
}
```

dla dociekliwych

Należy zwrócić uwagę, że wyrażenie **'2'** jest typu **char** (o wartości **50**), ale wyrażenie **'2' + 0** (o takiej samej wartości!) jest typu **int**.

```
System.out.println('2');       // 2
System.out.println('2' + 0);   // 50
```

Okna dialogowe

Zamiast przygotowywać **argumenty** tuż przed wywołaniem aplikacji, można je dostarczać w dialogach wyświetlanych podczas jej wykonywania. W podobny sposób można ujawniać **rezultaty**.

Wykonanie aplikacji posługującej się dialogami należy zakończyć wywołaniem funkcji **System.exit**. Jeśli się tego nie uczyni, aplikacja pozostanie **aktywna** nawet **po zakończeniu** wykonywania funkcji głównej.

klasa JOptionPane

```
String showInputDialog(String cmd)
Wyświetla dialog z poleceniem cmd. Dostarcza łańcuch znaków podany
w klatce dialogu. Jeśli dialog zamknie się za pomocą przycisku
Cancel, to dostarcza null.
np. String result =
        JOptionPane.showInputDialog("Enter your name");
```

```
void showMessageDialog(Component parent, String msg)
void showMessageDialog(Component parent, String msg,
                       String title, int type)
```
W oknie ***parent*** (albo na środku ekranu, jeśli ***parent*** ma wartość **null**) wyświetla dialog z tytułem ***title*** i ikoną ***type***: m.in. **INFORMATION_MESSAGE**, **ERROR_MESSAGE**, **QUESTION_MESSAGE** (wszystkie z klasy **JOptionPane**) oraz komunikat ***msg***.
```
np. JOptionPane.showMessageDialog(
        null, results, "Program results",
        JOptionPane.INFORMATION_MESSAGE
    );

int showConfirmDialog(Component parent, Object msg)
```
W oknie ***parent*** (albo na środku ekranu, jeśli ***parent*** ma wartość **null**) wyświetla dialog z przyciskami **Yes**, **No** i **Cancel** oraz komunikat ***msg***. Dostarcza informację o naciśniętym przycisku: **YES_OPTION**, **NO_OPTION**, **CANCEL_OPTION** (wszystkie z klasy **JOptionPane**).
```
np. boolean isReady =
        showConfirmDialog(
            null, "Ready to exit?",
        ) == JOptionPane.YES_OPTION;
```

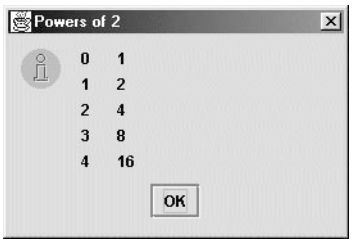
klasa System

```
void exit(int code)
```
Kończy wykonywanie aplikacji z kodem powrotu ***code***.
```
np. System.exit(0);
```

Następująca aplikacja, pokazana na ekranach ***Dialog wejściowy*** i ***Dialog wyjściowy***, ilustruje użycie dialogów do wprowadzania danych i wyprowadzania wyników.

Ekran
Dialog wejściowy

Ekran
Dialog wyjściowy




```
package jbPack;

import javax.swing.*;
```

```
public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    public Program()
    {
        String reply =
            JOptionPane.showInputDialog("Enter count");

        int count = Integer.parseInt(reply);

        String results = "";
        for (int i = 0; i < count ; i++)
            results += i + "  " +
                       (int)Math.pow(2, i) + '\n';

        JOptionPane.showMessageDialog(
            null, results, "Powers of 2",
            JOptionPane.INFORMATION_MESSAGE
        );

        System.exit(0);
    }
}
```

Aplikacje okienkowe

Aplikacja okienkowa tworzy ***okno*** zawierające **pulpit**. Istnieje możliwość dowolnego określenia ***rozmiarów***, ***położenia***, ***koloru*** i ***tytułu*** okna.

Szkielet aplikacji okienkowej opartej na klasie **Context** (***©JanB***), znajduje się w pliku **Program.java**, w katalogu *base***jbKawa\jbContext** (por. Dodatek ***Stosowanie szkieletów***).

W szkielecie umieszczono liczne polecenia importu, wystarczające do większości zastosowań.

W celu użycia szkieletu aplikacji okienkowej należy otworzyć projekt

*base***jbKawa\jbContext\jbContext.kpx**

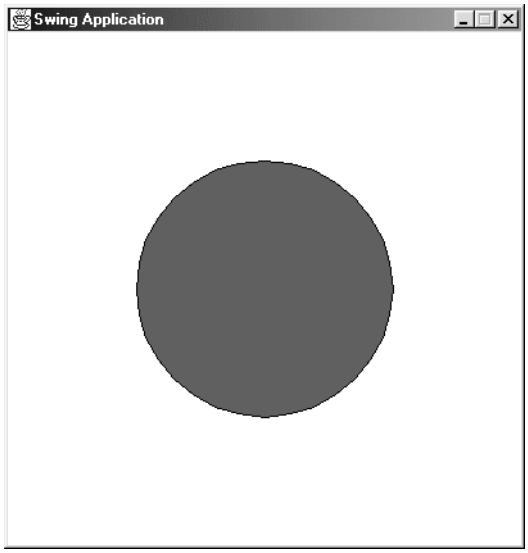
a następnie zmodyfikować albo wymienić szkielet.

Zniszczone pliki **Context.java** i **Program.java** można odtworzyć z katalogu *base***jbKawa**, gdzie występują pod nazwami **!Context.java** i **Context-Program.java**.

Większość aplikacji okienkowych różni się od innych tylko definicją klasy **ContentPane** nazywaną w skrócie ***pod-aplikacją***. Dlatego testując przykłady podane w książce, wystarczy się ograniczyć do wymiany **pod-aplikacji** zawartej w pliku **Program.java** na pod-aplikację wziętą z książki.

Następująca pod-aplikacja, pokazana na ekranie ***Koło i okrąg***, wykreśla na pulpicie czerwone koło z czarną obwódką.

Ekran
Koło i okrąg




```
public
class ContentPane
    extends Content {

    private int w, h;

    public void initPane()
    {
        w = getWidth();
        h = getHeight();
    }

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        gDC.setColor(Color.red);
        gDC.fillOval(w/4, h/4, w/2, h/2);

        gDC.setColor(Color.black);
        gDC.drawOval(w/4, h/4, w/2-1, h/2-1);
    }
}
```

Szkielet aplikacji okienkowej

Szkielet, który wymaga dołączenia do projektu pliku **Context.java**, zaprojektowano w taki sposób, że ułatwia programowanie aplikacji graficznych. Z przytoczonego zestawu ***poleceń importu*** pierwszy jest potrzebny niemal **zawsze**, drugi **często**, a trzeci tylko w aplikacjach **zaawansowanych**.

W szkielecie zawarto: funkcję inicjującą **initPane** i funkcję odtwarzającą **paintComponent,** wywoływane w podanej kolejności. Funkcja **paintComponent** jest wywoływana tuż po wyświetleniu zawierającego ją okna, a ponadto za każdym razem, gdy nastąpi zasłonięcie i odsłonięcie pulpitu. Każda z wymienionych funkcji może być **pominięta**.

Nie zaleca się wstawienia do szkieletu konstruktora klasy **ContentPane**, ale przydatne może być wstawienie tam definicji bezparametrowej funkcji **initFocus** z wywołaniem funkcji **dispose**. W takim przypadku nie nastąpi wyświetlenie pulpitu, a aplikacja okienkowa upodobni się do konsolowej.

```
package jbPack;

// polecenia importu

public
class Program
    extends Context {

    public static void main(String[] args)
    {
        new Program();
    }

// ====== szkielet pod-aplikacji ======================== //

    public
    class ContentPane
        extends Content {

        public void initPane()
        {
        }

        public void paintComponent(Graphics gDC)
        {
            super.paintComponent(gDC);
        }
    }

// ====================================================== //

}
```

klasa ContentPane (©JanB)

```
void initPane()
Wykonuje czynności inicjujące nie obejmujące wykreślania. W ciele
funkcji wywołuje się zazwyczaj funkcje setBackground, getWidth,
getHeight i getGraphics, ale nie wywołuje się funkcji requestFocus.

void paintComponent(Graphics gDC)
Czyści pulpit kolorem tła i wykreśla na nim obiekty graficzne.

void initFocus()
Umożliwia nastawienie celownika na dowolny komponent pulpitu (por.
rozdział Celownik).
```

Jeśli w funkcji **initFocus** umieści się wywołanie funkcji **dispose()**, to nie dojdzie do wyświetlania pulpitu. Umożliwia to nadanie aplikacji okienkowej charakteru aplikacji konsolowej.

dla dociekliwych

Oparcie aplikacji na klasie **Context** umożliwia użycie szeregu wygodnych funkcji opisanych w Dodatku ***Klasy Szkieletowe***. W szczególności, jeśli każde wywołanie funkcji **System.out.println** zmieni się na wywołanie funkcji **println**, a tuż przed zakończeniem wykonania aplikacji wywoła funkcję **showConsole**, to wyniki zostaną dodatkowo wyświetlone w dialogu **Message**.

Pulpit

Pulpitem jest prostokątny obszar okna przewidziany do rozmieszczania ***komponentów*** (np. przycisków) oraz do wykreślania ***grafiki*** (np. prostokątów i elips). Lewy-górny narożnik pulpitu ma współrzędne (**0**, **0**) Współrzędne ***x*** pulpitu (*odcięte*) rosną ***w-prawo***, a współrzędne ***y*** (*rzędne*) rosną ***w-dół***. Do rozpoznania rozmiarów pulpitu służą funkcje **getWidth** i **getHeight**.

klasa Component

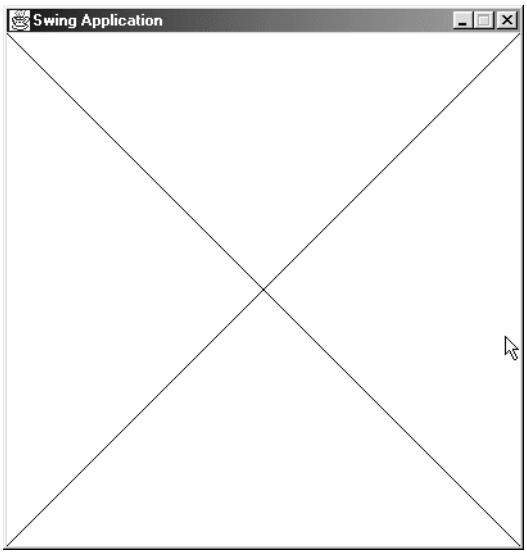
```
int getWidth()
Dostarcza szerokość pulpitu wyrażoną w pikselach.
np. int w = getWidth();

int getHeight()
Dostarcza wysokość pulpitu wyrażoną w pikselach.
np. int h = getHeight();

void setBackground(Color color)
Nadaje pulpitowi kolor tła color.
np. setBackground(Color.yellow);
```

```
void repaint()
Poleca Systemowi wywołać funkcję paintComponent. Nie musi to nastąpić
"natychmiast", bo o chwili wywołania tej funkcji decyduje System.
np. repaint();
```

Kilkanaście kolorów podstawowych ma oznaczenia symboliczne, m.in. **Color.white**, **Color.black**, **Color.red**, **Color.cyan**. **Color.magenta**, itp.

Na ekranie ***Pulpit aplikacji***, pokazano okno z pulpitem, na którym wykreślono jego przekątne.

Ekran
Pulpit aplikacji



Wykreślacz systemowy

Wykreślanie w obrębie pulpitu odbywa się za pomocą ***wykreślacza***. Odniesienie do wykreślacza jest dostarczane w parametrze funkcji **paintComponent**.

Wykreślaczowi można wydawać ***polecenia***: **setColor** (ustaw kolor), **drawString** (wykreśl napis), **drawLine** (wykreśl odcinek), **drawOval** (wykreśl owal, w tym okrąg), **drawRect** (wykreśl prostokąt, w tym kwadrat).

klasa Graphics

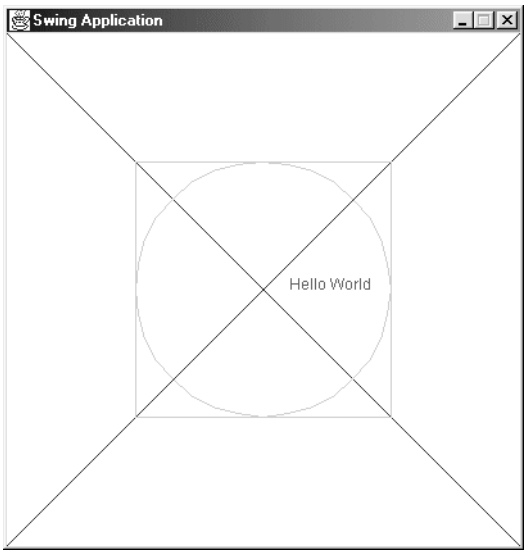
```
void setColor(int color)
Określa kolor pióra, które będzie używane do sporządzania wykresów.
np. gDC.setColor(Color.black);

void drawString(String string, int x, int y)
Wykreśla napis string, spoczywający na domyślnym odcinku bazowym
o lewym końcu w (x, y).
np. gDC.drawString("Hello", 100, 100);
```

```
void drawLine(int xA, int yA, int xZ, int yZ)
Wykreśla odcinek łączący punkty (xA, yA) i (xZ, yZ). Jeśli xA == xZ
oraz yA == yZ, wykreśla punkt.
np. gDC.drawLine(10, 10, 50, 50);

void drawOval(int x, int y, int w, int h)
Wykreśla owal wpisany w domyślny prostokąt o lewym-górnym wierzchoł-
ku w (x, y) i rozmiarach w+1 x h+1. W szczególności wykreśla okrąg.
np. gDC.drawOval(0, 0, 100-1, 100-1);

void drawRect(int x, int y, int w, int h)
Wykreśla prostokąt o lewym-górnym wierzchołku w (x, y) i rozmiarach
w+1 x h+1. W szczególności wykreśla kwadrat.
np. gDC.drawRect(100, 100, 200-1, 200-1);
```

W celu wykreślenia owalu albo prostokąta o rozmiarach ***width*** *x* ***height***, należy funkcjom **drawOval** i **drawRect** dostarczyć argumenty ***width***-1 i ***height***-1.

Następująca pod-aplikacja, pokazana na ekranie ***Funkcje graficzne***, wykreśla napis ***Hello World***, przekątne pulpitu oraz okrąg wpisany w kwadrat.

Ekran
Funkcje graficzne




```
public
class ContentPane
    extends Content {

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        int w = getWidth(),
            h = getHeight();
```

```
        gDC.setColor(Color.red);
        gDC.drawString("Hello World", w/2+20, h/2);

        gDC.setColor(Color.blue);
        gDC.drawLine(0, 0, w-1, h-1);
        gDC.drawLine(w-1, 0, 0, h-1);

        gDC.setColor(Color.green);
        gDC.drawRect(w/4, h/4, w/2-1, h/2-1);
        gDC.drawOval(w/4, h/4, w/2-1, h/2-1);
    }
}
```

Własny wykreślacz

Wykreślanie na pulpicie należy niekiedy wykonać **poza** funkcją **paintComponent**. W takim wypadku należy utworzyć **własny wykreślacz**.

Własny wykreślacz można utworzyć podczas wykonywania funkcji **initPane** albo później. Należy jednak pamiętać, że wykreślanie na pulpicie jest domyślnie **buforowane**, a więc wykres, który sporządzi się za pomocą **własnego** wykreślacza w funkcji **paintComponent**, zniknie z ekranu tuż po zakończeniu jej wykonywania (stanie się to na skutek skopiowania bufora na ekran).

W celu ***dezaktywowania buforowania*** należy wywołać funkcję **setDoubleBufferingEnabled** z argumentem **false**. Można to wykonać za pomocą instrukcji

```
RepaintManager.currentManager(this).
    setDoubleBufferingEnabled(false);
```

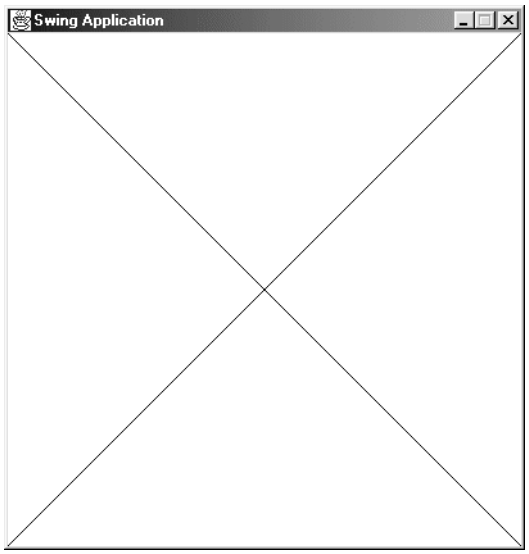
umieszczonej w konstruktorze klasy **ContentPane** albo w funkcji **initPane**.

klasa Component

```
Graphics getGraphics()
Dostarcza odniesienie do własnego wykreślacza.
np. Graphics pDC = getGraphics();

void dispose()
Zwalnia zasoby systemowe związane z własnym wykreślaczem.
np. pDC.dispose();
```

Następująca pod-aplikacja, pokazana na ekranie ***Własny wykreślacz***, wykreśla przekątne pulpitu: jedną za pomocą wykreślacza systemowego i drugą za pomocą własnego wykreślacza. Gdyby nie dezaktywowano buforowania (argument **false**), to wkrótce po wykreśleniu przekątnych jedna z nich **zniknęłaby** z pulpitu.

Ekran
Własny wykreślacz




```
public
class ContentPane
    extends Content {

    public ContentPane()
    {
        RepaintManager.currentManager(this).
            setDoubleBufferingEnabled(false);
    }

    private Graphics pDC;

    public void initPane()
    {
        pDC = getGraphics();
    }

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        int w = getWidth(),
            h = getHeight();

        gDC.drawLine(0, 0, w-1, h-1);
        pDC.drawLine(w-1, 0, 0, h-1);
    }
}
```

Celownik

Jeśli aplikację wyposaży się w ***oblicze graficzne***, składające się z takich komponentów jak ***pulpit*** (*content pane*), ***przycisk*** (*button*), ***etykieta*** (*label*) czy ***klatka*** (*text field*), a następnie naciśnie się **klawisz** klawiatury, to kod naciśniętego klawisza zostanie przekazany do komponentu, na który jest nastawiony ***celownik*** (*focus*).

Celownik jest wstępnie nastawiony na **pulpit**. Do nastawienia celownika na **widoczny** komponent służy funkcja **requestFocus**. Wykonanie tej czynności jest możliwe podczas wykonywania funkcji **initFocus** albo później.

klasa Component

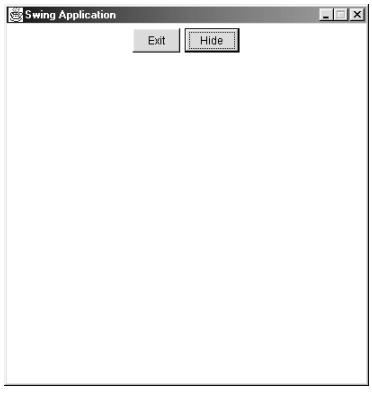
```
void setVisible(boolean visible)
Określa widoczność komponentu na podstawie visible. Domyślnie wszyst-
kie komponenty są widoczne.
np. button.setVisible(false);

void requestFocus()
Przenosi celownik na widoczny komponent, któremu wydano to polece-
nie. Można ją wywołać w funkcji initFocus albo później (w przeciw-
nym razie nie ma skutku).
np. requestFocus();  // nastawienie celownika na pulpit
```

Jeśli komponent, na który nastawiono celownik przestaje być widoczny, to następuje utrata celownika. W celu nastawienia go na inny komponent należy użyć funkcji **requestFocus**.

Następująca pod-aplikacja, pokazana na ekranie ***Nastawienie celownika***, ilustruje użycie funkcji **requestFocus** do nastawienia celownika: najpierw na przycisk ***Hide***, a następnie na przycisk ***Exit***. Dzięki temu naciśnięcie **dowolnego** klawisza, a następnie klawisza *Esc* powoduje zakończenie wykonywania aplikacji.

Gdyby w funkcji **initFocus** nie wywołano funkcji **requestFocus**, to znaki wprowadzane z klawiatury byłyby odbierane przez pulpit. A ponieważ nie przygotowano go do odbierania znaków, więc naciśnięcia klawiszy byłyby **ignorowane**.

Ekran
Nastawienie celownika



```
public
class ContentPane
    extends Content {

    private JButton exitButton, hideButton;

    public void initPane()
    {
            // utworzenie przycisku
        exitButton = new JButton("Exit");
        hideButton = new JButton("Hide");

            // zdefiniowanie obsługi klawiatury
        Watcher watcher = new Watcher();
        hideButton.addKeyListener(watcher);
        exitButton.addKeyListener(watcher);

            // umieszczenie przycisków na pulpicie
        add(exitButton);
        add(hideButton);
    }

    class Watcher
        extends KeyAdapter {

        public void
        keyReleased(KeyEvent evt)
        {
            if (evt.getSource() == hideButton) {
                hideButton.setVisible(false);
                exitButton.requestFocus();
                return;
            }

            int keyCode = evt.getKeyCode();
            if (keyCode == KeyEvent.VK_ESCAPE)
                System.exit(0);
        }
    }

    public void initFocus()
    {
            // celownik na przycisk
        hideButton.requestFocus();
    }
}
```

Aplikacje samodzielne

Aplikacją ***samodzielną*** jest aplikacja **okienkowa** nie posługująca się klasą **Context**.

Szkielet aplikacji samodzielnej, opartej na klasie **JFrame.java**, znajduje się w pliku **Program.java** (***©JanB***), w katalogu *base***jbKawa\jbFrame** (por. Dodatek ***Stosowanie szkieletów***).

W szkielecie umieszczono liczne polecenia importu, wystarczające do większości zastosowań.

W celu użycia szkieletu aplikacji okienkowej-samodzielnej należy otworzyć projekt

*base***jbKawa\jbFrame\jbFrame.kpx**

a następnie zmodyfikować albo wymienić pod-aplikację.

Zniszczony plik **Program.java** można odtworzyć z katalogu *base***jbKawa**, gdzie występuje pod nazwą **Frame-Program.java**.

klasa JFrame

JFrame(String caption)
Konstruuje obiekt klasy **JFrame** reprezentujący okno opatrzone paskiem tytułowym z napisem ***caption***.
np. JFrame frame = new JFrame("Swing Application");

void **setTitle**(String title)
Umieszcza na pasku tytułowym okna napis ***title***.
np. frame.setTitle("Swing application");

void **setResizable**(boolean resizable)
Na podstawie ***resizable*** określa, czy mogą ulec zmianie rozmiary okna.
np. frame.setResizable(false);

void **pack**()
Upakowuje okno, to jest tak dobiera jego rozmiary, aby były **jak najmniejsze**, ale **wystarczające** do pomieszczenia wszystkich jego komponentów.
np. frame.pack();

void **show**()
Wyświetla okno.
np. frame.show();

klasa Component

void **setSize**(int w, int h)
Nadaje komponentowi rozmiary ***w*** *x* ***h***.
np. frame.setSize(400, 400);

void **setLocation**(int x, int y)
Umieszcza lewy-górny narożnik komponentu w punkcie (***x***, ***y***).
np. frame.setLocation(150, 150);

void **setBounds**(int x, int y, int w, int h)
Umieszcza lewy-górny narożnik komponentu w punkcie (***x***, ***y***) i nadaje mu rozmiary ***w*** *x* ***h***.
np. frame.setBounds(150, 150, 400, 400);

```
void setBackground(Color color)
Jako kolor tła komponentu obiera color.
np. frame.setBackground(Color.red);

void setForeground(Color color)
Jako kolor pierwszo-planowego elementu komponentu obiera color.
np. frame.setForeground(Color.green);
```

Szkielet aplikacji samodzielnej

Szkielet aplikacji samodzielnej zamieszczono w Dodatku ***Klasy szkieletowe***. Poniżej przedstawiono jej wycinek, eksponujący **funkcję główną**, **konstruktor** i **pod-aplikację**.

```
package jbPack;

// polecenia importu

public
class Program
    extends JFrame {

    // deklaracje konfigurujące

    public static void main(String[] args)
    {
        new Program();
    }

    public Program()
    {
        // instrukcje konfigurujące
    }

    // pomocnicza klasa Content

// ====== pod-aplikacja ================================= //

    public
    class ContentPane
        extends Content {

        public void initPane()
        {
        }

        public void paintComponent(Graphics gDC)
        {
            super.paintComponent(gDC);
        }
    }

// ====================================================== //

}
```

Programy

Programem jest opis czynności przewidzianych do wykonania przez komputer.

Program może być zapisany w więcej niż jednym **pliku**. Każdy plik składa się z co najwyżej jednego określenia pakietu, może zawierać polecenia importu oraz definicje klas i interfejsów.

W skład klas i interfejsów wchodzą **deklaracje** i **instrukcje**. Deklaracje opisują właściwości zmiennych i funkcji, a instrukcje opisują czynności. Deklaracje i instrukcje odwołują się do **literałów**, **zmiennych** i **funkcji**.

Literały

Literałem jest napis reprezentujący **stałą**, o wartości wynikającej z zapisu literału. Przykładami literałów są ***liczby*** (np. **12** oraz **12.4**), ***znaki*** (np. **'a'**), ***łańcuchy*** (np. **"Yes"**), ***orzeczniki*** (np. **true**) oraz ***odnośnik pusty*** (**null**).

Jeśli znak albo łańcuch ma zawierać **"** (*cudzysłów*) albo **** (*ukośnik*), to należy go poprzedzić ukośnikiem (np. **'\\'**). Jeśli ma zawierać oznaczenie ***nowego wiersza***, to należy je zapisać jako **\n** (*ukośnik*, *litera* ***n***).

Napisy **–12** oraz **–12.4** nie są literałami. Są to **wyrażenia** składające się z literału poprzedzonego znakiem **–** (*minus*).

Literały szesnastkowe

Literałem szesnastkowym typu **int** jest napis o postaci **0x***cc* **…** *c*, w którym każde *c* jest cyfrą szesnastkową, zapisaną za pomocą cyfry dziesiętnej albo za pomocą małej albo dużej litery z przedziału **a** .. **f** (np. **0xc**, o wartości **12** i **0xFF**, o wartości **255**).

Literały szesnastkowe typu **long** zapisuje się podobnie jak literały szesnastkowe typu **int**, z tym, że literał musi kończyć mała albo duża litera **L** (np. **0xc4L**).

Zmienne

Program wykonuje operacje na zmiennych. Zmienną można postrzegać jako ***obszar pamięci***, przeznaczony do przechowywania danych określonego typu.

W szczególności instrukcja

```
int age = 16;
```

deklaruje zmienną **age** typu **int** (całkowitego), którą podczas deklarowania zainicjowano daną o wartości **16**, natomiast instrukcja

```
System.out.println(age);
```

jest opisem czynności, która polega na pobraniu danej przypisanej zmiennej **age** i wyprowadzeniu jej wartości na konsolę.

Typy zmiennych

Każda zmienna jest określonego ***typu***. Typ zmiennej określa się w ***deklaracji***. Z określenia typu wynika jakie dane można przypisywać zmiennej.

Inicjowanie zmiennych

Daną początkową można przypisać zmiennej już podczas jej deklarowania. Takie przypisanie jest nazywane ***zainicjowaniem*** zmiennej.

Zmienne ***nielokalne*** (zdefiniowane na zewnątrz funkcji) są inicjowane **niejawnie** (danymi o wartości **0**, **0.0**, **false** albo **null**).

Zmiennej ***lokalnej*** należy jawnie nadać wartość przed pierwszym użyciem (zainicjować ją albo przypisać jej daną).

Jeśli zmienną zadeklaruje się ze specyfikatorem **final**, to nadanie wartości może być co najwyżej jednokrotne.

Zmienna finalna ***Const*** zadeklarowana w inter-klasie ***Name*** ma zazwyczaj nazwę ***Name.Const*** (np. **Math.PI**, **Color.red**).

```
final double Pi;
Pi = 3.14;
System.out.println(Math.PI - Pi);  // ok. 0.0016
Pi = 3.1415;   // błąd (ponowne przypisanie wartości)
```

Zmienne numeryczne

Zmiennym numerycznym można przypisywać tylko **dane liczbowe**. Zmiennym typu **int** i **char** całkowite (np. **127**), a zmiennym typu **double** rzeczywiste (np. **-12.4**).

Zmiennym typu **char** można przypisywać tylko **kody** znaków ***Unikodu***. Kody te wyraża się zazwyczaj za pomocą literałów znakowych jak **'2'** (kod cyfry **2**) albo **'\n'** (kod znaku nowego wiersza) albo za pomocą liczb całkowitych; odpowiednio **50** i **10**.

Stosowanie *Unikodu*, a nie na przykład kodu *EBCDIC*, *ISO-7* albo *ASCII* umożliwia posługiwanie się znakami narodowymi wielu krajów, w tym znakami **polskimi** (także w identyfikatorach!).

```
int age;
int width = 40, height = 50;
final double Pi = 3.14;
// ...
age = 24;
```

dla dociekliwych

Zmienne całkowite są reprezentowane w ***zapisie-uzupełnieniowym-do-2***. Ma on tę właściwość, że jeśli zna się układ bitów reprezentujących pewną wartość ***v***, to można uzyskać układ bitów wartości ***-v*** przez zanegowanie bitów ***v*** i dodanie liczby **1**, tak jak to pokazano w tabeli ***Zapis uzupełnieniowy***.

Tabela Zapis-uzupełnieniowy

```
0000  ...   0000  0100    // 4

1111  ...   1111  1011    // po zanegowaniu
                  1    // dodanie 1
======================
1111  ...   1111  1100    // -4
```

Zmienne orzecznikowe

Zmienne orzecznikowe są typu **boolean**. Można im przypisywać tylko dane reprezentowane przez orzeczenia o wartości **true** (*prawda*) i **false** (*fałsz*). Takie orzeczenia mają zazwyczaj postać ***relacji*** (np. **speed > limit**) albo postać ***negacji***, ***koniunkcji*** i ***alternatywy*** wyrażeń orzecznikowych (np. **age > 12 && age < 20**, orzekającego czy **age** jest wiekiem **angielskiego** "nastolatka").

```
boolean isMarried = true;
// ...
isMarried = false;
```

Zmienne odnośnikowe

Zmienne odnośnikowe służą do identyfikowania obiektów. Zmiennej odnośnikowej można przypisać ***odniesienie puste***, nie identyfikujące żadnego obiektu (zapisane za pomocą literału odnośnikowego **null**) oraz ***odniesienia*** dostarczane przez ***fabrykatory***.

Przykładem fabrykatora jest

```
new String("JanB")
```

Dostarcza on odniesienie do obiektu typu **String** reprezentującego łańcuch składający się ze znaków napisu **JanB**.

Każdy fabrykator dostarcza odniesienie do **innego** obiektu, a każde stałe wyrażenie łańcuchowe o takiej samej wartości, dostarcza odniesienie do **tego samego** obiektu.

```
System.out.println(new String("Jan") == "Jan");  // false
System.out.println("Ja" + "nB" == "JanB");       // true
```

Tablice

Tablicą jest **zmienna** składająca się z zestawu ***elementów***. Każdy element tablicy jest zmienną typu **pierwotnego** albo **odnośnikowego**.

Wszystkie elementy tablicy są takiego samego typu. Liczba elementów nie może być większa od największej liczby typu **int**.

Z każdym elementem tablicy jest związany ***indeks***. Indeks jest liczbą **nieujemną**. Element tablicy poprzedzający wszystkie jej pozostałe elementy ma indeks **0**. W szczególności indeksami tablicy **3**-elementowej są liczby **0**, **1**, **2**.

Fabrykowanie tablic

Przykładowa instrukcja

```
int[] numbers;
```

nie tworzy tablicy o elementach typu **int**. Tworzy ona odnośnik do dowolnej **1**-wymiarowej tablicy o bliżej nie określonej liczbie elementów typu **int**.

Dopiero po użyciu fabrykatora **new int [5]**, który utworzy **5**-elementową tablicę o elementach typu **int** i dostarczy do niej odniesienie

```
numbers = new int [5];
```

można, za pomocą odnośnika **numbers**, wykonywać na elementach tablicy takie operacje jak na przykład

```
i = 2;
numbers[i+2] = 127;       // odwołanie do elementu o indeksie 4
```

albo

```
int[] numbers = new int [5];
i = 2;
numbers[i+2] = 127;
```

Deklaracja odnośnika do tablicy nie może zawierać określenia liczby elementów tablicy. Liczba elementów wynika z użytego fabrykatora, a **temu samemu** odnośnikowi można przypisywać odniesienia do tablic różniących się ***liczbą elementów***.

Inicjowanie tablic

Inicjator tablicy ma postać

```
{ list }
```

w której ***list*** jest **listą wyrażeń**, **inicjatorem tablicy**, słowem kluczowym **null** albo jest puste.

Inicjator tablicy może wystąpić w **deklaracji tablicy**

```
int[]   vec  = { 10, 20, 30 };
int[][] vec2 = {     // tablica nie-prostokątna (sic!)
    { 10 },
    { 20, 30 },
    { 40, 50, 60 }
};
```

oraz w **fabrykatorze tablicy**, w którym nie występuje określenie liczby jej elementów

```
int[]   vec  = new int[] { 10, 20, 30 };
int[][] vec2 =
    new int[][] {
        { 10 },
        { 20, 30 },
        { 40, 50, 60 }
    };
```

Jeśli nie użyje się inicjatora, to elementy tablicy zostaną zainicjowane domyślnie: numeryczne wartościami **0**, orzecznikowe wartościami **false**, a odnośnikowe wartościami **null**.

Każde wyrażenie albo pod-inicjator występujący w inicjatorze tablicy musi zapewniać zgodność w sensie przypisania z elementami inicjowanej tablicy.

Rozpoznawanie tablic

W ogólnym przypadku tablice mogą być ***jedno-wymiarowe*** i ***wielo-wymiarowe***. W Javie istnieją **tylko** tablice jedno-wymiarowe. Namiastką tablic wielo-wymiarowych są tablice ***wielo-poziomowe***.

Tablica jest ***jedno-poziomowa***, jeśli jej elementami są zmienne typu **pierwotnego** (np. **int**) albo typu **odnośnikowego** nie-tablicowego (np. **String**). Każda tablica jedno-poziomowa jest tablicą ***prostą***.

```
int[] vec = new int [] { 10, 20 };
String[] vec = new String [] { "Ewa", "Iza" };
```

Tablica jest ***wielo-poziomowa***, jeśli jej elementami są odnośniki do tablic.

```
int[][] arr = new int [] { { 10, 20 }, { 30 } };
```

Elementami ***podstawowymi*** są te elementy tablicy, które nie są odnośnikami do tablic. Wszystkie elementy podstawowe muszą być **takiego samego** typu.

W szczególności

`new int[] { 10, 20 }`
fabrykuje tablicę prostą o elementach typu **int**

`new String [] { "Ewa", "Iza" }`
fabrykuje tablicę prostą o elementach typu **String**

`new String[][] { { "Ewa", "Jan" }, { "Iza" } }`
fabrykuje dwu-poziomową tablicę nie-prostokątną o elementach typu **String[]**, której elementy podstawowe są typu **String**

Wszystkie wymienione tablice są jedno-wymiarowe (sic!).

Tablice wielo-poziomowe

W fabrykatorze tablicy wielo-poziomowej musi wystąpić tyle par nawiasów kwadratowych ile w deklaracji występuje par nawiasów kwadratowych. Dowolny zestaw **końcowych** par nawiasów kwadratowych może być **pusty**. Jeśli choć jedna taka para nawiasów jest **pusta**, to tablica wielo-poziomowa jest ***nie-kompletna***.

W szczególności

`new int[4][]`
fabrykuje **4**-elementową tablicę odnośników typu **int[]** zainicjowanych wartościami **null**

`new int[4][3]`
fabrykuje **4**-elementową tablicę odnośników typu **int[]** do **3**-elementowych tablic pierwotnych, składających się z elementów typu **int** o wartości **0**.

`new String[4][3]`
fabrykuje **4**-elementową tablicę odnośników typu **String[]** do **3**-elementowych tablic prostych, składających się z elementów typu **String** o wartości **null**.

Tablice nie-prostokątne

Tablice wielo-poziomowe nie muszą być prostokątne.

W szczególności

```
int[][] tri = {
    { 10, 20, 30 },
    { 40, 50 },
    { 60 }
};
```

deklaruje tablicę ***trójkątną***, która jest **3**-elementową tablicą odnośników na tablice proste o nie-identycznej liczbie elementów.

Opracowanie następującej deklaracji

```
int[][] tri = {
    new int[] { 10, 20, 30 },
    new int[] { 40, 50 },
    new int[] { 60 }
};
```

ma taki sam skutek jak wykonanie instrukcji

```
int[][] tri;
tri = new int [3][];
tri[0] = new int[] { 10, 20, 30 };
tri[1] = new int[] { 40, 50 };
tri[2] = new int[] { 60 };
```

Przetwarzanie tablic

Przetwarzanie tablicy polega na wykonywaniu operacji na jej **elementach**. Jeśli nazwą tablicy jest ***name***, to liczbę jej elementów określa wyrażenie ***name*.length**.

```
int[] vec = new int[] { 10, 20, 30, };
System.out.println(vec.length);      // 3

int[][] tri = { { 10, 20 }, { 30 }, { } };
```

```
System.out.println(tri[0].length);  // 2
System.out.println(tri[1].length);  // 1
System.out.println(tri[2].length);  // 0
```

Na ogół nie wolno dopuścić do tego, aby podczas wykonania programu wystąpiło odwołanie do elementu, który **nie istnieje**. Naruszenie tego wymagania powoduje bowiem wysłanie wyjątku klasy **ArrayIndexOutOfBoundsException**.

Kompletność tablic

Tablica jest ***kompletna***, jeśli żaden jej element nie ma wartości **null**. Tablicą kompletną jest w szczególności każda tablica prosta, o elementach typu pierwotnego.

W szczególności **jest** kompletna tablica

```
int[][] values = {
    { },
    {10, 20 }
};
```

ale **nie jest** kompletna tablica

```
int[][] values = {
    null,
    {10, 20 }
};
```

ani tablica

```
String[][] matrix = new String[2][];
```

dla dociekliwych

Następująca aplikacja ilustruje przetwarzanie tablic nie-kompletnych. Zawiera ona funkcję, która może być użyta do sumowania elementów podstawowych kompletnych i nie-kompletnych tablic typu **int[][]**.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }
```

```
    public Program()
    {
        int[][] arr = new int [][]  {
            { 10, 20, 30 },
            null,
            { 40, 50 },
            { },
            { 60 }
        };

        System.out.println(arr.length);      // 5
        System.out.println(arr[0].length);   // 3
        System.out.println(arr[3].length);   // 0

        System.out.println(sum(arr));        // 210

    }

    public int sum(int[][] par)
    {
        int sum = 0;
        for (int i = 0; i < par.length ; i++) {
            for (int j = 0;
                 par[i] != null &&
                 j < par[i].length ; j++
                ) {
                sum += par[i][j];
            }
        }
        return sum;
    }
}
```

Tablice dynamiczne

Rozmiar tablicy jest określany w chwili jej fabrykowania i nie może po tym ulec zmianie. Można jednak utworzyć **większą** tablicę, skopiować do niej elementy pierwszej i dalej posługiwać się tym samym odnośnikiem do tablicy.

Następujący program tworzy tablicę, w której umieszcza całkowite **argumenty programu**, a następnie wyprowadza je w kolejności ***od-końca-do-początku***. Dzięki dynamicznemu zarządzaniu pamięcią operacyjną, nie wprowadzono ograniczenia co do maksymalnej liczby przetwarzanych argumentów.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Program(args);
    }
```

```
    private int size = 1;
    private int[] vec = new int [size];
    private int pos;

    public Program(String[] args)
    {
        int count = args.length;
        for (int i = 0; i < count ; i++) {
            String arg = args[i];
            try {
                int number = Integer.parseInt(arg);
                saveArg(number);
            }
            catch (NumberFormatException e) {
                e.printStackTrace();
                System.exit(0);
            }
        }

        for (int i = pos-1; i >= 0  ; i--)
            System.out.println(vec[i]);
    }

    public void saveArg(int number)
    {
        if (pos == size) {
            int[] ref = new int [2 * size];
            System.arraycopy(vec, 0, ref, 0, size);
            vec = ref;
            size *= 2;
        }

        vec[pos++] = number;
    }
}
```

Wyrażenia

Wyrażenia są zestawami ***operacji***. Do wykonywania operacji na zmiennych służą ***operatory***. W celu określenia **kolejności** w jakiej mają być wykonane operacje, ***respektuje się nawiasy*** oraz odwołuje do takich pojęć jak ***priorytet*** i ***wiązanie*** operatora (por. Dodatek ***Priorytety operatorów***).

Respektowanie nawiasów

Jeśli wyrażenie zawarte w nawiasach okrągłych jest poprzedzone operatorem, to wykonanie operacji określonej przez ten operator nastąpi dopiero **po** opracowaniu wyrażenia w nawiasach.

Przykład

Podczas opracowywania wyrażenia

```
a + (b - c)
```

*operacja **dodawania** zostanie wykonana dopiero **po** wykonaniu operacji odejmowania.*

Uwzględnianie priorytetów

Jeśli wyrażenie można uznać za argument **2** operacji o **różnych** priorytetach, to operację wyrażoną przez operator o niższym priorytecie wykonuje się **po** operacji o wyższym priorytecie.

Przykład

*Ponieważ priorytet mnożenia jest **wyższy** od priorytetu dodawania, więc podczas opracowywania wyrażenia*

```
a + b * c
```

*najpierw zostanie wykonana operacja **mnożenia**, a dopiero po tym operacja **dodawania**.*

Uwzględnianie wiązań

Jeśli wyrażenie można uznać za argument **2** operacji o **równych** priorytetach, to kolejność wykonania operacji jest określona przez **wiązanie**.

Jeśli wiązanie operacji jest ***lewe***, to najpierw wykonuje się operację wyrażoną przez operator znajdujący się z **lewej** strony wyrażenia, a jeśli jest ***prawe***, to najpierw wykonuje się operację wyrażoną przez operator znajdujący się z **prawej** strony wyrażenia.

Przykład

*Ponieważ wiązanie operatorów dodawania i odejmowania jest **lewe**, więc wyrażenie*

```
a - b + c
```

zostanie potraktowane jak wyrażenie

```
(a - b) + c       // a nie jak wyrażenie a - (b + c)
```

*Ponieważ wiązanie operatora przypisania jest **prawe**, więc wyrażenie*

```
a = b = c
```

zostanie potraktowane jak wyrażenie

```
a = (b = c)       // a nie jak wyrażenie (a = b) = c
```

Uwzględnianie kolejności

Argumenty wywołań funkcji opracowuje się w kolejności ***od-lewej-do-prawej***. Jeśli operacja jest **2**-argumentowa, to najpierw opracowuje się jej **lewy** argument, a dopiero po tym **prawy**.

Jeśli opracowanie lewego argumentu zakończy się "gwałtownie" (np. na skutek powstania sytuacji wyjątkowej), to nie dojdzie do opracowania żadnej części prawego argumentu.

Przykład 1

Podczas opracowywania wyrażenia

```
fun1() + fun2()
```

*najpierw zostanie wywołana funkcja **fun1**, a dopiero po tym funkcja **fun2**.*

Przykład 2

Podczas opracowywania wyrażenia

```
a[b] = ++b
```

*zidentyfikowanie elementu, którego ma dotyczyć przypisanie odbędzie się **przed** wykonaniem operacji zwiększenia.*

Operacje przypisania

Prosta operacja przypisania ma postać

a = ***b***

w której ***a*** jest nazwą zmiennej, a ***b*** jest wyrażeniem.

Wykonanie operacji polega na wyznaczeniu wartości wyrażenia ***b*** i po ewentualnym niejawnym przekształceniu do typu zmiennej ***a***, przypisaniu jej tej zmiennej.

W miejscu wykonania operacji zostanie dostarczona wartość zmiennej ***a***.

Niejawne przekształcenie powinno być numeryczną konwersją **rozszerzającą**, konwersją z klasy pochodnej na bazową albo konwersją z klasy na implementowany przez nią interfejs.

Złożona operacja przypisania ma postać

a @= ***b***

w której @= jest jednym z operatorów języka (np. **>>>=**).

Operację ***a*** @= ***b*** (np. **a += b**) wykonuje się **podobnie** jak operację

a = ***a*** @ ***b***

ale jej **lewy** argument opracowuje się **jednokrotnie**.

Przykład

Po wykonaniu instrukcji

```
int fix = 2;
fix += 3;
```

*zmienna **fix** otrzyma wartość **5**.*

Zabrania się wykonywania przypisań ***zawężających*** (np. przypisania zmiennej typu **char** wartości zmiennej typu **int**). Jeśli przypisanie zawężające **musi** być wykona-

ne, to należy stosować ***konwersje obcinające*** (np. **(byte)259** ma wartość **3**, a **(byte)(256+128)** ma wartość **-128**).

Operacje arytmetyczne

Operacje arytmetyczne wykonuje się za pomocą operatorów wymienionych w tabeli ***Operacje arytmetyczne***.

Tabela Operacje arytmetyczne

+	(*dodawanie*)	**-**	(*odejmowanie*)
*****	(*mnożenie*)	**/**	(*dzielenie*)
%	(*reszta*)		
++	(*zwiększenie o 1*)	**--**	(*zmniejszenie o 1*)
+=	(*dodanie*)	**-=**	(*odjęcie*)
*=	(*pomnożenie*)	/=	(*podzielenie*)

Sposób wykonania podstawowych operacji arytmetycznych nie wymaga opisu. Należy jedynie zauważyć, że rezultat dzielenia całkowitego jest **całkowity**, a argumenty wyznaczania reszty **muszą** być całkowite (np. **11 / 4** ma wartość **2**, a **11 % 4** ma wartość **3**).

Dla jednego albo dwóch argumentów ujemnych, rezultat operacji **a % b** ma znak **pierwszego** argumentu, a jego wartość bezwzględna jest równa reszcie z dzielenia wartości bezwzględnej pierwszego argumentu przez wartość bezwzględną drugiego (np. zarówno **-11 % -4** jak i **-11 % 4** ma wartość **-3**).

Operacje przedrostkowe

Wykonanie operacji ***++num*** powoduje **zwiększenie** wartości zmiennej ***num*** o **1**. W miejscu wykonania operacji dostarcza się zwiększone ***num***.

Wykonanie operacji ***--num*** powoduje **zmniejszenie** wartości zmiennej ***num*** o **1**. W miejscu wykonania operacji dostarcza się zmniejszone ***num***.

Wyrażając to **poglądowo**: operacja przedrostkowa modyfikuje argument, zwiększając albo zmniejszając go o **1**, a w miejscu jej wykonania dostarcza **nową** wartość argumentu.

Przykład

```
int fix = 10;
int fix2 = ++fix;
```

*Zmienne **fix** i **fix2** otrzymują ostatecznie wartości **11**.*

Operacje przyrostkowe

Wykonanie operacji ***num++*** powoduje **zwiększenie** wartości zmiennej ***num*** o **1**. W miejscu wykonania operacji dostarcza się kopię pierwotnego ***num***.

Wykonanie operacji ***num--*** powoduje **zmniejszenie** wartości zmiennej ***num*** o **1** W miejscu wykonania operacji dostarcza się kopię pierwotnego ***num***.

Wyrażając to **poglądowo**: operacja przyrostkowa modyfikuje argument, zwiększając albo zmniejszając go o **1**, a w miejscu jej wykonania dostarcza **starą** wartość argumentu.

Przykład

```
int fix = 10;
int fix2 = fix--;
```

*Zmienne **fix** i **fix2** otrzymują ostatecznie wartości **9** i **10**.*

Operacje porównania

Operacje porównania wykonuje się za pomocą operatorów wymienionych w tabeli ***Operacje porównania***.

Tabela Operacje porównania

==	(*równe*)	**!=**	(*nie równe*),
<	(*mniejsze*)	**>**	(*większe*),
<=	(*mniejsze lub równe*)	**>=**	(*większe lub równe*)

Jeśli porównanie wyraża ***orzeczenie*** prawdziwe, to jego rezultat ma wartość **true** (*prawda*). W przeciwnym razie ma wartość **false** (*fałsz*).

Porównanie na równość wykonuje się za pomocą operacji **==,** a nie za pomocą operacji **=**.

Przykład

```
char chr = 'q';
boolean isGreater = chr > 'a';
```

*Ponieważ kod litery **q** jest większy niż kod litery **a**, więc zmienna **isGreater** otrzymuje wartość **true**.*

Porównania odnośników

Porównanie **odnośników** nie ma w zasadzie żadnego związku z porównaniem wartości identyfikowanych przez nie **zmiennych**. Dlatego porównania zmiennych identyfikowanych przez odnośniki powinny być wykonywane za pomocą wyspecjalizowanych funkcji, takich jak **equals** i **compareTo**.

Przed porównaniem łańcuchów ignoruje się wszystkie znaki najdłuższego podciągu, od którego zaczynają się oba łańcuchy. Jeśli po wykonaniu takiej operacji **oba** łańcuchy są **puste**, to uznaje się je za **równe**. Jeśli tylko **jeden** jest pusty, to uznaje się go za **mniejszy**. Jeśli oba są **niepuste**, to mniejszy jest ten, którego pierwszy znak ma **mniejszy** kod.

klasa String

```
boolean equals(Object object)
Dostarcza orzeczenie o wartości: "obiekt, któremu wydano to polece-
nie, jest identyczny z obiektem identyfikowanym przez object".
np. boolean isEqual =
        ("Jan" + "B").equals("JanB");  // true

int compareTo(Object object)
Dostarcza liczbę o wartości ujemnej, 0, dodatniej, stosownie do
tego czy obiekt, któremu wydano to polecenie ma wartość mniejszą,
równą, większą od wartości obiektu identyfikowanego przez object.
np.  boolean isNegative =
         "Ewa".compareTo("Jan") < 0;   // true

String intern()
Dostarcza odniesienie do niejawnego literału (sic!) o takiej samej
wartości, jaką reprezentuje obiekt któremu wydano to polecenie.
np. String four = ("" + 12/3).intern();
```

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    public Program()
    {
        String s1 = "Yes";
        String s2 = new String("Yes");
        String s3 = new String("Yes").intern();
        System.out.println(s1 == s2);       // false
        System.out.println(s1.equals(s2));  // true
        System.out.println(s1 == s3);       // true
    }
}
```

Operacje orzecznikowe

Operacje orzecznikowe wykonuje się za pomocą operatorów wymienionych w tabeli ***Operacje orzecznikowe***.

Tabela Operacje orzecznikowe

`!`	(*zaprzeczenie*)	`&&`	(*koniunkcja*)	`||`	(*alternatywa*)

Obowiązują następujące zasady

1. Argumenty i rezultaty operacji orzecznikowych są typu **boolean** i mają wartości **false** albo **true**.
2. Rezultat ***zaprzeczenia*** ma wartość **true** tylko wówczas, gdy argument ma wartość **false**.
3. Rezultat ***koniunkcji*** ma wartość **true** tylko wówczas, gdy oba argumenty mają wartość **true**.
4. Rezultat ***alternatywy*** ma wartość **false** tylko wówczas, gdy oba argumenty mają wartość **false**.

Operacje koniunkcji i alternatywy wykonuje się w taki sposób, że jeśli po opracowaniu lewego argumentu jest **znany** rezultat całej operacji (bo dla **koniunkcji** lewy argument ma wartość **false**, a dla **alternatywy** ma wartość **true**), to **rezygnuje się** z opracowania prawego argumentu.

Przykład

```
int var1 = 1;
boolean var2 = true;
if (var1 < 0 && (var2 = false))
    ;
System.out.println(var2);  // true

if (var1 > 0 || (var2 = false))
    ;
System.out.println(var2);  // true
```

Operacje konwersji

Wykonanie konwersji ma na celu przekształcenie zmiennej pewnego typu w zmienną typu ***docelowego***.

Operacja konwersji wyrażenia ***e*** do typu ***Type*** ma postać

```
(Type)e
```

Przykład

*Jeśli zmiennej **chr** przypisano kod **małej** litery alfabetu angielskiego, to wykonanie instrukcji*

```
char chr = (char)(chr - 'a' + 'A');
```

*powoduje przypisanie jej kodu **dużej** litery.*

*Użycie konwersji jest **niezbędne**, ponieważ wyrażenie w nawiasach jest typu **int**, a nie istnieje niejawna konwersja z typu **int** do **char**.*

Operacje wyboru

Operację wyboru wykonuje się za pomocą trój-argumentowego operatora wyboru **?:** (*pytajnik*, *dwukropek*).

W miejscu opracowania wyrażenia

e ? ***eT*** : ***eF***

dostarcza się ***eT*** jeśli ***e*** ma wartość **true** albo ***eF*** w przeciwnym razie.

Po opracowaniu wyrażenia ***e***, opracowuje się tylko **jedno** z wyrażeń ***eT*** i ***eF***.

Przykład

Wykonanie instrukcji

```
String reply = a < b ? "Smaller" : "Greater";
```

*powoduje przypisanie zmiennej **reply** łańcucha orzekającego o wyniku porównania **a** i **b**.*

Operacje bitowe

Operacje bitowe wykonuje się na bitach zmiennych całkowitych. Często przydają się do tego ***literały szesnastkowe***.

Literał szesnastkowy ma postać **0x*hh…h***, w której każde ***h*** jest cyfrą szesnastkową (**0**..**9** i **a**..**f**). W tabeli ***Literały szesnastkowe*** podano: **zapis** literału, jego **wartość** i **reprezentację**.

Tabela Literały szesnastkowe

Zapis	*Wartość*	*Reprezentacja*
0x2	2	0000 ... 0000 0000 0010
0xf	15	0000 ... 0000 0000 1111
0x12	18	0000 ... 0000 0001 0010
0x2a	42	0000 ... 0000 0010 1010

Podczas wykonywania operacji bitowych, typem wspólnym argumentów jest **int**, ale jeśli jeden z argumentów jest typu **long**, to jest nim **long**.

Operacja negowania bitów

Operacja zanegowania bitów ma postać

~***exp***

w której ***exp*** jest wyrażeniem całkowitym.

Rezultatem operacji zanegowania bitów jest zmienna powstała ze zmiennej reprezentowanej przez wyrażenie ***exp***, po ***zanegowaniu*** każdego jej bitu.

Negacją bitu **1** jest bit **0**, a negacją bitu **0** jest bit **1**. Zanegowanie bitów zmiennej, a następnie dodanie do niej liczby **1** ma taki sam skutek jak zmiana wartości zmiennej na przeciwną (np. **a + ~a + 1** ma zawsze wartość **0**).

Przykład

```
int val = 3;              // 00 ... 0011
val = ~val;               // 11 ... 1100
boolean res =
    -val == 1 + ~val;     // true
```

*Niezależnie od wartości zmiennej **val**, wartością zmiennej **res** jest **true**.*

Operacja iloczynowania bitów

Operacja iloczynowania bitów ma postać

expL & ***expR***

w której ***expL*** i ***expR*** są wyrażeniami całkowitymi.

W celu utworzenia rezultatu operacji, zmienne reprezentowane przez ***expL*** i ***expR*** poddaje się konwersjom do typu wspólnego, a następnie każdy bit rezultatu tworzy się z odpowiadających sobie bitów argumentów, wyznaczając ich ***iloczyn logiczny***.

Iloczyn logiczny pary bitów ma wartość **1** tylko wówczas gdy oba te bity mają wartość **1**.

Przykład

```
int fix = 6;              // 00 ... 0110
int mask = 0x3;           // 00 ... 0011
fix &= ~mask;             // 00 ... 0100
```

*Wykonanie operacji na zmiennej **fix** powoduje **wyzerowanie** tych wszystkich jej bitów, które w **mask** są **jedynkowe**.*

Operacja sumowania bitów

Operacja sumowania bitów ma postać

expL | ***expR***

w której ***expL*** i ***expR*** są wyrażeniami całkowitymi.

W celu utworzenia rezultatu operacji, zmienne reprezentowane przez ***expL*** i ***expR*** poddaje się konwersjom do typu wspólnego, a następnie każdy bit rezultatu tworzy się z odpowiadających sobie bitów argumentów, wyznaczając ich ***sumę logiczną***.

Suma logiczna pary bitów ma wartość **0** tylko wówczas, gdy oba te bity mają wartość **0**.

Przykład

```
int fix = 5;              // 00 ... 0101
int mask = 0x3;           // 00 ... 0011
fix |= mask;              // 00 ... 0111
```

*Wykonanie operacji na zmiennej **fix** powoduje **ustawienie** tych wszystkich jej bitów, które w **mask** są **jedynkowe**.*

Operacja sumowania bitów modulo 2

Operacja sumowania bitów *modulo* **2** bitów ma postać

expL ^ ***expR***

w której ***expL*** i ***expR*** są wyrażeniami całkowitymi.

W celu utworzenia rezultatu operacji, zmienne reprezentowane przez ***expL*** i ***expR*** poddaje się konwersjom do typu wspólnego, a następnie każdy bit rezultatu tworzy się z odpowiadających sobie bitów argumentów, wyznaczając ich ***sumę bitów modulo 2***.

Suma *modulo* ***2*** pary bitów ma wartość **1** tylko wówczas gdy bity te są **różne**.

Przykład

```
int fix = 6;              // 00 ... 0110
int mask = 0x3;           // 00 ... 0011
fix ^= mask;              // 00 ... 0101
```

Wykonanie operacji na zmiennej ***fix*** *powoduje* ***zanegowanie*** *tych wszystkich jej bitów, które w* ***mask*** *są* ***jedynkowe***.

Operacja przesunięcia w lewo

Operacja przesunięcia bitów **w lewo** ma postać

```
expL << n
```

w której ***expL*** i ***n*** są wyrażeniami całkowitymi.

Każdy bit rezultatu tworzy się z bitów zmiennej reprezentowanej przez ***expL***, po przesunięciu ich o ***n*** pozycji **w lewo**, z tym, że przed wykonaniem przesunięcia, argument ***n*** zastępuje się liczbą znajdującą się na **5** (dla **int**) albo na **6** (dla **long**) jego dolnych bitów (np. **1 << 33** ma wartość **2**, a nie **0**).

Podczas przesuwania **w lewo**, kolejne bity "wysuwane ze zmiennej" są **odrzucane**, a na pozycje najmniej znaczące wchodzą bity **0**.

Przykład

```
int fix = 3;              // 00 ... 0011
fix <<= 2;                // 00 ... 1100
```

Bity zmiennej ***fix*** *przesunięto o* ***2*** *pozycje* ***w lewo***.

Operacja przesunięcia w prawo

Operacja przesunięcia bitów **w prawo** ma postać

```
expL >> n
```

w której ***expL*** i ***n*** są wyrażeniami całkowitymi.

Każdy bit rezultatu tworzy się z bitów zmiennej reprezentowanej przez ***expL***, po przesunięciu ich o ***n*** pozycji **w prawo**, z tym, że przed wykonaniem przesunięcia, argument ***n*** zastępuje się liczbą znajdującą się na **5** (dla **int**) albo na **6** (dla **long**) jego dolnych bitów (np. **2 >> 33** ma wartość **1**, a nie **0**).

Podczas przesuwania **w prawo**, kolejne bity "wysuwane ze zmiennej" są **odrzucane**, natomiast ***bit znaku*** zachowuje swoją wartość.

Przykład

```
int fix = 7;              // 00 ... 0111
fix >>= 2;                // 00 ... 0001
```

*Bity zmiennej **fix** przesunięto o **2** pozycje **w prawo**.*

Operacja przesunięcia w prawo „bez znaku”

Operacja przesunięcia bitów **w prawo bez znaku** ma postać

```
expL >>> n
```

w której ***expL*** i ***n*** są wyrażeniami całkowitymi.

Każdy bit rezultatu tworzy się z bitów zmiennej ***expL***, po przesunięciu ich o ***n*** pozycji **w prawo**, z tym, że przed wykonaniem przesunięcia, argument ***n*** zastępuje się liczbą znajdującą się na **5** (dla **int**) albo na **6** (dla **long**) jego dolnych bitów (np. **2 >>> 33** ma wartość **1**, a nie **0**).

Podczas przesuwania **w prawo**, kolejne bity "wysuwane ze zmiennej" są **odrzucane**. Natomiast bit znaku jest ustawiany na wartość **0**.

Przykład

```
int fix = -1;             // 11 ... 1111
fix >>>= 1;               // 01 ... 1111
```

*Bity zmiennej **fix** przesunięto o **1** pozycję **w prawo**.*

Operacja rozpoznania

Operacja rozpoznania ma na celu rozstrzygnięcie, czy odnośnik identyfikuje obiekt określonej ***klasy*** albo czy identyfikuje obiekt klasy implementującej określony ***interfejs***. Ma ona postać

ref instanceof ***Type***

w której ***ref*** jest odnośnikiem, a ***Type*** jest **opisem typu** (np. **String** albo **int[]**).

Jeśli ***Type*** jest opisem typu tablicowego, to jest dostarczane orzeczenie o wartości: "*odnośnikowi **ref** przypisano odniesienie do tablicy typu **Type***".

Jeśli ***Type*** jest identyfikatorem ***Name***, to jest dostarczane orzeczenie o wartości: "*odnośnikowi **ref** przypisano odniesienie do obiektu klasy **Name** albo odniesienie do obiektu klasy implementującej interfejs **Name***".

Przykład

Po wykonaniu instrukcji

```
Object firstName = "Isabel";
boolean isString = firstName instanceof String;
```

*orzecznik **isString** ma wartość **true**.*

dla dociekliwych

Następująca funkcja, w zależności od tego, czy elementy dostarczonej jej tablicy są typu **int** czy **String**, zeruje je albo przypisuje im wartości **null**.

```
public boolean setNils(Object arg)
{
    if (arg instanceof int []) {
        int[] vec = (int [])arg;
        for (int i = 0; i < vec.length ; i++)
            vec[i] = 0;
    } else if (arg instanceof String []) {
        String[] vec = (String [])arg;
        for (int i = 0; i < vec.length ; i++)
            vec[i] = null;
    } else
        return false;
    return true;
}
```

Wyrażenia stałe

W pewnych miejscach programu, mogą wystąpić tylko wyrażenia ***stałe***.

Wyrażeniem stałym jest takie wyrażenie, które składa się wyłącznie z literałów i odwołań do statycznych zmiennych finalnych zainicjowanych wyrażeniami stałymi, ale nie zawiera operacji **–**, ++, ani **instanceof**.

W szczególności, w zasięgu następujących deklaracji

```
final int one = 1;
final double two = 2.9;
```

wyrażenie

```
(int)(two + 2)
```

jest wyrażeniem **stałym** całkowitym o wartości **4** (sic!).

Konwersje

Konwersją jest ***przekształcenie*** zmiennej pewnego typu w zmienną na ogół **innego** typu (np. zmiennej typu **double** w zmienną **int**).

1. Konwersją ***numeryczną*** jest przekształcenie numeryku w numeryk. Odbywa się to na ogół z zachowaniem zbliżonej wartości. Na przykład **(int)4.8** ma wartość **4**, a **(double)4** ma wartość **4.0**.
2. Konwersją ***odnośnikową*** jest przekształcenie odnośnika w odnośnik. Na przykład odnośnika typu **String** w odnośnik typu **Object** albo odwrotnie.
3. Konwersją ***orzecznikową*** jest przekształcenie orzecznika w orzecznik. Ponieważ jednak istnieje tylko jeden typ orzecznikowy, więc taka konwersja jest zawsze ***tożsamościowa*** i jej przydatność jest żadna.
4. Innych konwersji **nie ma**!

Konwersje odnośnikowe są weryfikowane przez *Kompilator* i przez *Maszynę Wirtualną*. Konwersja zaakceptowana przez *Kompilator* jest poprawna **statycznie**, a zaakceptowana przez *Maszynę Wirtualną* jest poprawna **dynamicznie**.

Konwersja odnośnikowa jest ***poprawna*** tylko wówczas, gdy jest poprawna **statycznie** i **dynamicznie**. Konwersje tożsamościowe są poprawne **zawsze**.

Konwersje, które mogą być wykonane niejawnie, to jest bez użycia operatora konwersji, są konwersjami ***standardowymi***. Są nimi konwersje ***rozszerzające*** (m.in. z **char** do **int**, z **int** do **long** i z **long** do **double**), ale nie są nimi konwersje ***zawężające*** (np. z **int** do **char** i z **long** do **int**), chyba że jeszcze przed podjęciem wykonania programu wiadomo, że niejawne zastosowanie konwersji zawężającej nie spowoduje zmiany wartości (jak w przypisaniu zmiennej typu **char** danej typu **int** o wartości **51**).

```
String str;
int val;
char chr;
int one = 1;

str = "Ewa";              // "Ewa"
str += 2;                 // "Ewa2"
```

```
str = "Ewa";
str = str + 2 + 3;        // "Ewa23"

str = "Ewa";
str += 2 + 3;             // "Ewa5"

str = "Ewa";
str = str + (2 + 3);      // "Ewa5"

val = '2' + one;          // 51
val = 1 << 16;            // 65536
val = (char)(1 << 16);    // 0

chr = 51;                 // '3'
chr = '2' + 1;            // '3'
chr = "3".charAt(0);      // '3'
chr = (char)(50 + one);   // '3'

chr = 50 + one;           // błąd
chr = '2' + one;          // błąd
chr = "2";                // błąd
```

Instrukcje

Instrukcje dzielą się na ***czynne*** i ***bierne***. Instrukcją bierną jest instrukcja ***deklaracyjna***, a instrukcjami czynnymi są: instrukcja ***pusta***, ***grupująca***, ***wyrażeniowa*** i ***warunkowa***, instrukcje ***iteracyjne*** i ***decyzyjne*** wraz z instrukcjami ***zaniechania*** i ***kontynuowania*** oraz instrukcje ***powrotu*** i ***wyjątków***. Każda instrukcja jest zakończona **klamrą** albo **średnikiem**.

Instrukcja **czynna** może być poprzedzona jedną albo większą liczbą ***etykiet***. Etykieta ma postać **identyfikatora**, a od następującej po niej instrukcji albo etykiety, oddziela ją znak `:` (*dwukropek*).

Przykład

Następująca instrukcja powrotu jest poprzedzona etykietą ***Finish***

```
Finish: return true;
```

Instrukcja pusta

Instrukcja pusta ma postać

```
;
```

Jej wykonanie nie wywołuje żadnych skutków.

Przykład

```
{
JustLabel:
    ;       // instrukcja pusta
    Fin:    // błąd (brak instrukcji po dwukropku)
}
```

Instrukcja grupująca

Instrukcja grupująca ma postać

```
{ Ins Ins ... Ins }
```

w której każde ***Ins*** jest dowolną instrukcją albo jest **puste**.

Wykonanie instrukcji grupującej składa się z sekwencyjnego wykonania zawartych w niej instrukcji ***Ins***.

Użycie instrukcji grupującej ma na celu utworzenie **z sekwencji instrukcji** dokładnie **jednej instrukcji**.

Przykład

*Następujący napis jest zapisem dokładnie **jednej** instrukcji grupującej.*

```
{
    String msg = "Hello";
    System.out.println(msg);
}
```

Instrukcja wyrażeniowa

Instrukcja wyrażeniowa ma postać

exp;

w której ***exp*** jest: **przypisaniem**, **wywołaniem**, **zwiększeniem** albo **zmniejszeniem**.

Wykonanie instrukcji wyrażeniowej składa się z opracowania wyrażenia ***exp***, a następnie zignorowania rezultatu tego opracowania (jeśli taki istnieje).

Jeśli instrukcja wyrażeniowa jest wywołaniem funkcji, to powinna pociągać za sobą ***skutki uboczne***, takie jak ***przypisywanie*** i ***przesyłanie*** danych. W przeciwnym razie jej użycie jest **zbyteczne**.

Przykłady

```
var = 10;                  // przypisanie
var1 = var2 = 10;          // przypisanie
System.out.println(var);   // przesłanie
var2++;                    // zwiększenie
var1 + var2++;             // błąd
fun1() && fun2();          // błąd
```

Instrukcja warunkowa

Instrukcja warunkowa ma postać

```
if (exp) InsT
```

albo

```
if (exp) InsT else InsF
```

w których ***exp*** jest wyrażeniem orzecznikowym, a ***InsT*** oraz ***InsF*** jest instrukcją.

Jeśli ***InsT*** jest instrukcją **warunkową**, to stosuje się zasadę, że każdej frazie **else** odpowiada najbliższa, poprzedzająca ją fraza **if**.

W szczególności, następująca instrukcja, w której ***Ins1*** ani ***Ins2*** nie jest instrukcją warunkową,

```
if (exp1) if (exp2) Ins1 else Ins2
```

jest równoważna instrukcji

```
if (exp1) {
    if (exp2) {
        Ins1
    else
        Ins2
}
```

a nie instrukcji

```
if (exp1) {
    if (exp2)
        Ins1
} else
    Ins2
```

Wykonanie instrukcji warunkowej zaczyna się od wyznaczenia wartości wyrażenia ***exp***. Jeśli jest nią **true**, to wykonuje się instrukcję ***InsT***, a w przeciwnym razie instrukcję ***InsF*** (o ile występuje **else** ***InsF***). Po wykonaniu tych czynności, wykonanie instrukcji warunkowej zostaje zakończone.

Przykłady

```
if (speed > limit)          // jeśli speed większe niż limit
    speed = limit;          // przypisz speed wartość limit
else                        // w przeciwnym razie
    speed += 10;            // zwiększ speed o 10


if (age > 10 && age < 20)   // jeśli age jest wiekiem nastolatka
    System.exit(0);         // zakończ wykonywanie programu


if (var < 5 && var != 0)
    System.out.println(var);
```

```
if (flag || var >= 5)
    var = 0;
else
    System.out.println(var);


if (var >= 0) {
    if (var <= 9)
        System.out.println(var);
} else
    var = -1;
```

W ostatnim przykładzie użyto instrukcji grupującej po to, aby przed frazą ***else*** *nie wystąpiła instrukcja grupująca bez frazy* ***else****.*

Gdyby pominięto nawiasy klamrowe, to rozpatrywana instrukcja stałaby się równoważna instrukcji

```
if (var >= 0) {
    if (var <= 9)
        System.out.println(var);
    else
        var = -1;
}
```

Instrukcje iteracyjne

Instrukcje iteracyjne umożliwiają ***cykliczne*** wykonywanie objętych nimi instrukcji. Instrukcja iteracyjna powinna **gwarantować** zakończenie jej wykonania.

Instrukcja while

Instrukcja *while* ma postać

```
while (expC) Ins
```

w której ***expC*** jest wyrażeniem orzecznikowym, a ***Ins*** jest instrukcją.

Przykład

```
int i = -3;
while (i++ < 0)
    System.out.println(i);  // -2 -1 0
```

Wykonanie instrukcji *while* składa się z cyklicznego wykonywania następujących czynności

1. Opracowania i wyznaczenia wartości wyrażenia ***expC***.
2. Jeśli wyrażenie ***expC*** ma wartość **true**, wykonania instrukcji ***Ins***.

Jeśli podczas kolejnego (w tym pierwszego) opracowania wyrażenia ***expC*** okaże się, że ma ono wartość **false**, to wykonanie instrukcji *while* zostanie zakończone.

Instrukcja iteracyjna *while* dobrze nadaje się do przypadków, gdy nie można przewidzieć wymaganej liczby iteracji.

Przykład 1
Zliczanie bitów

Następujący program podaje ile **zer** kończy reprezentację podanej liczby niezerowej.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    private int number = 12;

    public Program()
    {
        int count = 0;
        while ((number & 1) == 0) {
            count++;
            number >>= 1;
        }

        System.out.println("Count = " + count);
    }
}
```

Przykład 2
Zliczanie elementów

Następujący program zlicza te początkowe elementy tablicy, które poprzedzają element o wartości **0**.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
```

```
    {
        new Program();
    }

    private int[] numbers = { 10, 20, 30, 0, 40, };

    public Program()
    {
        int i = 0,
            length = numbers.length,
            count = 0;
        while (numbers[i++] != 0) {
            if (i == length) {
                System.out.println("Zero not found!");
                return;
            }
            count++;
        }

        System.out.println("Count = " + count);
    }
}
```

Instrukcja for

Instrukcja *for* ma postać

```
for (ini0; expC ; expI) Ins
```

w której ***ini0*** jest deklaracją albo listą wyrażeń jakie mogą wystąpić w instrukcji wyrażeniowej, ***expC*** jest wyrażeniem orzecznikowym, ***expI*** jest wyrażeniem albo listą wyrażeń jakie mogą wystąpić w instrukcji wyrażeniowej, a ***Ins*** jest instrukcją.

Każdy z napisów: ***ini0***, ***expC*** oraz ***expI*** może być pusty. Jeśli puste jest ***expC***, to w jego miejsce niejawnie wstawia się literał **true**.

Przykład

```
for (int i = 0, j = 0; i < 5 ; i++, j++) {
    if (i == 2)
        j = 0;
    System.out.println(j);  // 0 1 0 1 2
}
```

Wykonanie instrukcji *for* składa się z jednokrotnego wykonania instrukcji ***ini0;*** oraz z cyklicznego wykonywania następujących czynności

1. Opracowania i wyznaczenia wartości wyrażenia ***expC***.
2. Jeśli wyrażenie ***expC*** ma wartość **true**, wykonania instrukcji ***Ins*** oraz instrukcji utworzonych z wyrażeń ***expI*** (po uzupełnieniu każdego średnikiem).

Jeśli podczas kolejnego (w tym pierwszego) opracowania wyrażenia ***expC*** okaże się, że ma ono wartość **false**, to wykonanie instrukcji *for* zostanie zakończone.

Należy odnotować, że wykonanie instrukcji

```
for (ini0; expC ; exp, exp, ... , exp)
    Ins
```

ma taki sam skutek jak wykonanie instrukcji

```
{
    ini0
    while (expC) {
        Ins
        exp; exp; ... exp;
    }
}
```

a więc deklaracje zmiennych występujące w ***ini0*** nie są widoczne poza pętlą *for*.

Przykład 1

```
    // zadeklarowanie odnośnika
int[] vec;

    // sfabrykowanie tablicy
    // przypisanie odnośnikowi,
    // odniesienia do tablicy
vec = new int [5];

    // dla i = 0, dopóki i < 5,
    // przypisanie elementom tablicy
    // kwadratów ich indeksów,
    // a po każdym przypisaniu
    // wykonanie operację i++
for (int i = 0; i < 5 ; i++)
    vec[i] = i * i;

    // wyprowadzenie kwadratów
for (int i = 0; i < 5 ; i++)
    System.out.println(vec[i]);
```

Przykład 2

```
    // wyznaczenie sumy elementów tablicy,
    // po uprzednim wyzerowaniu tych,
    // które mają wartości większe niż 10
int vec[] = { 1, 20, 3, 40, },
    sum = 0;
```

```
for (int i = 0; i < vec.length ; i++) {
    if (vec[i] > 10)
        vec[i] = 0;
    sum += vec[i];
}
System.out.println(
    "Sum =" + sum  // Sum = 4
);
```

Przykład 3

```
    // wyznaczenie sumy elementów
    // dwu-poziomowej tablicy
int[][] matrix =
    {
        { 1, 2, 3 },
        { 4, 5, 6 },
    };
int sum = 0;
for (int i = 0; i < matrix.length ; i++)
   for (int j = 0; j < matrix[i].length ; j++)
       sum += matrix[i][j];
System.out.println(
    "Sum = " + sum  // Sum = 21
);
```

Przykład 4

```
    // wyznaczenie sumy elementów
    // trójkątnej tablicy
    // częściowo utworzonej w deklaracji
int[][] matrix =
    {
        { 1, },
        null,
        { 4, 5, 6 },
    };
int sum = 0;
matrix[1] = new int[] { 2, 3, };

for (int i = 0; i < matrix.length ; i++)
   for (int j = 0; j < matrix[i].length ; j++)
       sum += matrix[i][j];
System.out.println(
    "Sum = " + sum  // Sum = 21
);
```

Przykład 5
Normalizowanie liczb

Następujący program ***normalizuje*** i wyprowadza liczby zawarte w tablicy (z maksimum **3** cyframi ułamkowymi).

Normalizacja polega na podzieleniu każdej liczby przez ***średnią arytmetyczną*** liczb.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    private int[] numbers = { 2, 4, 2, };

    public Program()
    {
        double average;
        int count = numbers.length,
            sum = 0;
        for (int i = 0; i < count ; i++)
            sum += numbers[i];
        average = (double)sum / count;

        String result = "";
        for (int i = 0; i < count ; i++) {
            String number = numbers[i] / average + "";
            int dotPos = number.indexOf('.'),
                ePos   = number.indexOf('e');
            if (ePos == -1)
                ePos = number.indexOf('E');
            if (ePos == -1)
                ePos = number.length();
            if (ePos-dotPos > 3)
            number = number.substring(0, dotPos+4) +
                     number.substring(ePos);
            result +=  number + ' ';
        }

        System.out.println("Result is: " + result);
    }
}
```

Przykład 6
Sortowanie liczb

Następujący program ***sortuje*** w porządku **rosnącym** i wyprowadza liczby zawarte w tablicy. Zastosowane tu sortowanie polega na porównywaniu i ewentualnym przestawianiu sąsiadujących liczb, tak aby **mniejsza** wystąpiła **wcześniej**. Operacja ta jest powtarzana do chwili, gdy przestawianie okaże się zbyteczne.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    private int[] numbers = { 30, 10, 20, };

    public Program()
    {
        int count = numbers.length,
            upper, lower;
        boolean swapped = true;
        while (swapped) {
            swapped = false;
            for (int i = 0; i < count-1 ; i++) {
                upper = numbers[i];
                lower = numbers[i+1];
                if (lower < upper) {
                    numbers[i]   = lower;
                    numbers[i+1] = upper;
                    swapped = true;
                }
            }
        }

        String result = "";
        for (int i = 0; i < count; i++)
            result += numbers[i] + " ";

        System.out.println("After sort:  " + result);
    }
}
```

Przykład 7
Sortowanie łańcuchów

Następujący program wyprowadza łańcuchy zawarte w tablicy, po posortowaniu ich w porządku **rosnącym**. Zastosowane tu sortowanie polega na wyszukiwaniu najmniejszego łańcucha i umieszczaniu go na pozycji docelowej. Operacja ta jest powtarzana **wielokrotnie**, do chwili gdy wszystkie łańcuchy zajmą pozycje docelowe.

Jeśli tablica zawiera *N* elementów, to wyszukiwanie dotyczy kolejno: a) pierwszych *N* elementów począwszy od elementu **0**, b) *N-1* elementów począwszy od elementu **1**, c) *N-2* elementów począwszy od elementu **2**, itd.

```
package jbPack;

public
class Program {
```

```
    public static void main(String[] args)
    {
        new Program();
    }

    private String[] strings = { "Kaja", "Iza", "Ewa", };

    public Program()
    {
        int count = strings.length;
        for (int i = 0; i < count ; i++) {
            String min = strings[i], tmp;
            int pos = i;
            for (int j = i; j < count ; j++) {
                tmp = strings[j];
                if (tmp.compareTo(min) < 0) {
                    min = tmp;
                    pos = j;
                }
            }
            tmp = strings[i];
            strings[i] = strings[pos];
            strings[pos] = tmp;
        }

        String result = "";
        for (int i = 0; i < count ; i++)
            result += strings[i] + ' ';

        System.out.println("After sort:  " + result);
    }
}
```

Instrukcja do

Instrukcja ***do*** ma postać

```
do
    Ins
while (expC);
```

w której ***Ins*** jest instrukcją, a ***expC*** jest wyrażeniem orzecznikowym.

Przykład

```
int i = -3;
do
    System.out.println(i);  // -3 -2 -1
while (++i < 0);
```

Wykonanie instrukcji ***do*** składa się z cyklicznego wykonywania następujących czynności

1. Wykonania instrukcji ***Ins***.
2. Opracowania i wyznaczenia wartości wyrażenia ***expC***.
3. Jeśli wyrażenie ***expC*** ma wartość **true**, ponownego wykonania podanych czynności.

Jeśli podczas kolejnego (w tym pierwszego) opracowania wyrażenia ***expC*** okaże się, że ma ono wartość **false**, to wykonanie instrukcji *do* zostanie zakończone.

Instrukcja ***Ins*** jest wykonywana co najmniej jeden raz.

Przykład 1

```
    // wyprowadzenie pierwszego znaku,
    // który w wierszu wprowadzonym z konsoli
    // jest różny od spacji
int chr;
do
    chr = System.in.read();
while (chr == ' ');
System.out.println((char)chr);
```

Przykład 2
Generowanie liczb

Następujący program wyprowadza nieujemne, pseudo-losowe liczby **2**-cyfrowe, podzielne przez podaną liczbę.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    private int number = 5;

    public Program()
    {
        String result = "";
        int random;
        for (int i = 0; i < 5 ; i++) {
            do
                random = (int)(Math.random() * 100);
            while (random % number != 0);

            result += random + " ";
        }
```

```
        System.out.println("Randoms are: " + result);
    }
}
```

Przykład 3
Symulowanie kalkulatora

Każde wyrażenie można zapisać w postaci ***beznawiasowej***. Podczas opracowywania jego wartości, kolejne liczby są umieszczane na ***stosie***, a kolejne operacje są wykonywane na szczytowych elementach stosu. Jeśli ograniczyć się do dwuargumentowych operacji + (*dodaj*) i * (*pomnóż*), to obowiązują odwzorowania podane w tabeli ***Notacja odwrotna***.

Na szczyt stosu można ***dokładać*** elementy, ale ze stosu można je ***zdejmować*** tylko **po-jednym** i zawsze **od-szczytu**. Opracowanie wyrażenia **1 2 + 3 *** odbywa się następująco: a) na stos **1**, b) na stos **2**, c) ze stosu **2**, ze stosu **1**, na stos suma **1** i **2**, d) na stos **3**, ze stosu **3**, ze stosu **3**, na stos iloczyn **3** i **3**, e) ze stosu **9**, która jest **wynikiem**.

Tabela Notacja odwrotna

Z nawiasami	*Bez nawiasów*
`(1 + 2) * 3`	`1 2 + 3 *`
`1 * (2 + 3)`	`1 2 3 + *`

Następujący program traktuje zawartość tablicy jako zapis uproszczonego wyrażenia w ***Polskiej Notacji Odwrotnej*** (tylko dodawania i mnożenia!) i wyprowadza jego wartość.

```
package jbPack;

import java.util.*;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    private String string = "1 2 + 3 *";

    public Program()
    {
        StringTokenizer tokens =
            new StringTokenizer(string);
        int count = tokens.countTokens();
        int[] stack = new int [count];
        int i = 0, pos = 0;
```

```
        do {
            String item = tokens.nextToken();
            if (isInteger(item))
                stack[pos++] = Integer.parseInt(item);
            else {
                if (i < 2)
                    error("Too few arguments");

                if (item.equals("+"))
                    stack[--pos-1] += stack[pos];
                else if (item.equals("*"))
                    stack[--pos-1] *= stack[pos];
                else
                    error("Not implemented");
            }
            i++;
        } while (tokens.hasMoreTokens());

        if (pos != 1)
            error("Too many arguments");

        System.out.println("Result is " + stack[0]);

    }

    public boolean isInteger(String item)
    {
        try {
            Integer.parseInt(item);
        }
        catch (NumberFormatException e) {
            return false;
        }
        return true;
    }

    public void error(String error)
    {
        System.out.println(error);
        System.exit(0);
    }
}
```

Instrukcje zaniechania i kontynuowania

We wnętrzu pętli mogą być użyte instrukcje

```
break;
```

oraz

```
continue;
```

Wykonanie instrukcji ***zaniechania*** (**break**) powoduje zakończenie wykonywania pętli, natomiast wykonanie instrukcji ***kontynuowania*** (**continue**) powoduje potraktowanie jeszcze nie wykonanych instrukcji **bieżącego** obrotu pętli jak instrukcji **pustej**.

Przykład 1

```
    // wyznaczenie sumy najkrótszego zestawu
    // początkowych elementów tablicy,
    // których suma przekracza 100
int vec[] = { 1, 2, 100, 3, };
int sum = 0, i = 0;
while (i < vec.length)
    if ((sum += vec[i++]) > 100)
            break;
if (sum > 100)
    System.out.println(
        "Sum = " + sum  // Sum = 103
    );
else
    System.out.println("Sum is less than 100");
```

Przykład 2

```
int sum = 0;
for (int i = 1; i <= 9 ; i++) {
    if (i < 3) {
        sum--;
        continue;
    } else if (sum > 7)
        break;
    sum = sum + 2*i;
}
System.out.println("Sum = " + sum);  // Sum = 12
```

*Zmienna **sum** przyjmuje kolejno wartości: **0**, **-1**, **-2**, **4**, **12**.*

Zakończenie wykonywania pętli następuje znacznie wcześniej niż wynikałoby z rozpatrzenia jej pierwszego wiersza.

Użycie etykiet

Instrukcje zaniechania i kontynuowania mogą mieć również postać

```
break Lab;
```

oraz

```
continue Lab;
```

w których ***Lab*** jest **etykietą** opatrującą instrukcję iteracyjną w skład której wchodzi dana instrukcja zaniechania albo kontynuowania.

Wykonanie instrukcji zaniechania **z etykietą** powoduje **zakończenie** wykonywania instrukcji iteracyjnej poprzedzonej podaną etykietą, a wykonanie instrukcji kontynuowania **z etykietą** powoduje zaniechanie wykonania jeszcze nie wykonanych instrukcji bieżącego obrotu pętli, a po wykonaniu instrukcji wyrażeniowych kończących bieżący obrót pętli (np. **i++;**) ponowne wykonanie instrukcji iteracyjnej poprzedzonej podaną etykietą.

Przykład 1

```
    // wyznaczenie sumy elementów tablicy,
    // z wyjątkiem tych elementów,
    // które w danym wierszu są ujemne
    // albo następują po elemencie ujemnym
int arr[][] =
    {
        { 1,  2,  0,  2 },
        { 2, -1,  2,    },
        { 1,  2         },
    };
int sum = 0, val;
Loop:
for (int i = 0; i < arr.length ; i++) {
    for (int j = 0; j < arr[i].length ; j++) {
        if ((val = arr[i][j]) < 0)
            continue Loop;
        sum += val;
    }
}
System.out.println(
    "Sum = " + sum  // Sum = 10
);
```

Przykład 2

Pętla sumuje elementy zawarte w kolejnych wierszach tablicy, ale kończy się w chwili napotkania elementu o wartości ***0****. Gdyby z instrukcji zaniechania usunięto etykietę, to nastąpiłoby wyprowadzenie liczby* ***12****.*

```
int arr[][] = {
                    { 1, 2, 3 },
                    { 4, 0, 5 },
                    { 1, 1, 0 }
              };

int sum = 0;

Loop:
for (int i = 0; i < 3 ; i++)
    for (int j = 0; j < 3 ; j++)
        if (arr[i][j] == 0)
            break Loop;
        else
            sum += arr[i][j];

System.out.println("Sum = " + sum);  //  Sum = 10
```

Instrukcja decyzyjna

Instrukcja decyzyjna ma postać

```
switch (exp0) {
    Case Case ... Case Default
}
```

w której każde ***Case*** jest frazą o postaci

```
case exp: Ins Ins ... Ins break;
```

a ***Default*** jest frazą o postaci

```
default: Ins Ins ... Ins break;
```

W takim zapisie, ***exp0*** jest wyrażeniem całkowitym, każde ***exp*** jest wyrażeniem stałym całkowitym (zazwyczaj liczbą albo identyfikatorem zmiennej ustalonej), a każde ***Ins*** jest instrukcją albo jest napisem **pustym**.

Pominięcie frazy ***Default*** powoduje jej domniemanie w postaci

```
default : break;
```

Wykonanie instrukcji wyboru zaczyna się od opracowania i wyznaczenia wartości wyrażenia ***exp0***. Następnie jego wartość jest porównywana z wartościami wyrażeń ***exp*** zawartych w kolejnych frazach ***Case***, aż do stwierdzenia równości.

Po stwierdzeniu ***równości*** są wykonywane instrukcje danej frazy. W przeciwnym razie są wykonywane instrukcje frazy ***Default***. Po zakończeniu tych czynności wykonanie instrukcji wyboru zostaje zakończone.

Jeśli fraza **case** albo **default** nie kończy się instrukcją **break** albo **return**, to są wykonywane instrukcje **kolejnej** frazy. Wykorzystano to w następującej aplikacji, którą można wywołać z wieloma zestawami argumentów.

W szczególności, jeśli aplikację wywoła się z **2** argumentami, to określą one wartości zmiennych **width** i **height**, podczas gdy **time** przyjmie wartość domyślną.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
            // wartości domyślne
        String width  = "400",
               height = "400",
               time   = "30";
```

```
        switch (args.length) {
            case 3:
               time   = args[2];
            case 2:
               height = args[1];
            case 1:
               width  = args[0];
            case 0:
               break;
            default:
                System.exit(0);
        }

        // ...

        new Program();
    }

    // ...
}
```

Instrukcje powrotu

Instrukcja powrotu ma postać

```
return;
```

albo

```
return exp;
```

w której ***exp*** jest wyrażeniem.

Wykonanie instrukcji powrotu powoduje zakończenie wykonywania zawierającej ją funkcji. Jeśli funkcja jest ***rezultatowa***, to jej rezultat jest inicjowany wartością wyrażenia ***exp***, po ewentualnym niejawnym przekształceniu jej do typu rezultatu.

Niejawne przekształcenie powinno być numeryczną konwersją rozszerzającą, konwersją z klasy pochodnej na bazową albo konwersją z klasy na implementowany przez nią interfejs.

Przykład

```
public int fun(int par)
{
    if (par > 0)
        return par * par;
    else
        return 0;
}
```

Instrukcje wyjątków

Z tego miejsca programu, w którym zostanie rozpoznana ***sytuacja wyjątkowa***, jest wysyłany ***wyjątek***: **obiekt** klasy pochodnej od **Throwable**, w tym obiekt klasy **Error** i **Exception**.

Do zarządzania wyjątkami służą instrukcje *try* i *throw*. Instrukcji **try** używa się do obsługiwania, a instrukcji **throw** do wysyłania wyjątków.

Instrukcja obsługi wyjątków

Instrukcja obsługi wyjątków ma postać

```
try {
    block
}
Catch
Finally
```

w której ***block*** jest blokiem, ***Catch*** jest zestawem fraz

```
catch (Dcl) {
    block
 }
```

zawierających deklarację ***Dcl*** parametru anonimowej funkcji do obsługiwania wyjątków, a ***Finally*** jest frazą

```
finally {
    block
}
```

W instrukcji obsługi wyjątków musi wystąpić co najmniej jedna fraza ***Catch*** albo ***Finally***.

Przykład

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        try {
            System.out.println(
                Integer.parseInt(args[0])
            );
        }
```

```
        catch (IndexOutOfBoundsException e) {
            System.out.println("Please supply argument");
        }
        catch (NumberFormatException e) {
            System.out.println("Argument is not a number");
        }
        finally {
            System.out.println(
                args.length + " argument(s) supplied"
            );
        }
    }
}
```

Wykonanie instrukcji wyjątków składa się z wykonania instrukcji zawartych w bloku występującym po frazie **try**. Jeśli podczas ich wykonywania zostanie wysłany wyjątek, to dalsze wykonywanie bloku zostanie **zaniechane**, a ***wyjątek*** zostanie odebrany i obsłużony przez tę pierwszą frazę **catch**, z której parametrem można skojarzyć odnośnik do wyjątku.

Jeśli żadna z fraz **catch** instrukcji wyjątków nie jest w stanie odebrać wyjątku, to nastąpi zakończenie wykonywania funkcji zawierającej tę instrukcję i wysłanie nie odebranego wyjątku w miejscu jej wywołania. Zabrania się, aby takie ***propagowanie*** wyjątku spowodowało wysłanie go poza program.

Niezależnie od tego jaki był przebieg wykonania instrukcji *try*, ale bezpośrednio przed jej zakończeniem (w szczególności przed wysłaniem wyjątku poza funkcję zawierającą tę instrukcję), jest wykonywany blok frazy **finally**. Użycie go jest m.in. przydatne do zwolnienia zasobów przydzielonych w bloku frazy **try**, nie zwolnionych na skutek zaniechania jego dalszego wykonywania, spowodowanego wysłaniem wyjątku.

Przykład

W celu dostarczenia wartości bezwzględnej elementu tablicy o podanym indeksie (albo liczby ***-1*** *jeśli element taki nie istnieje), przydaje się funkcja, którą można zdefiniować w sposób* ***tradycyjny*** *albo* ***zalecany****.*

sposób tradycyjny

```
int getAbs(int[] vec, int i)
{
    int size = vec.length;
    if (i < 0 || i >= size)
        return -1;
    return Math.abs(vec[i]);
}
```

sposób zalecany

```
int getAbs(int[] vec, int i)
{
    try {
        return abs(vec[i]);
    }
    catch (ArrayIndexOutOfBoundsException e) {
        return -1;
    }
}
```

Instrukcja wysłania wyjątku

Instrukcja wysłania wyjątku ma postać

```
throw ref;
```

w której ***ref*** jest odnośnikiem do obiektu klasy pochodnej od ***Throwable***.

Wykonanie instrukcji wysłania wyjątku powoduje wysłanie wyjątku identyfikowanego przez ***ref***.

Odnośnik ***ref*** staje się wówczas niejawnym argumentem frazy **catch** tej instrukcji wyjątków, która ***dynamicznie obejmuje*** wywołanie funkcji, z której wysłano wyjątek.

Przykład

```
throw new IllegalArgumentException("Negative");
```

Następujący program informuje o tym, czy jego argument jest całkowity (np. o wartości **12** albo **12e2**).

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        if (args.length != 1)
            System.out.println("Supply one argument!");
        else
            try {
                check(args[0]);
                System.out.println("Argument is integral");
            }
            catch (NumberFormatException e) {
                System.out.println(
                    "Argument is " + e.getMessage());
            }
    }
```

```
    public static void check(String item)
        throws NumberFormatException
    {
        try {
            double number = Double.parseDouble(item);
            if (number != (int)number)
                throw new NumberFormatException("not integral");
        }
        catch (NumberFormatException e) {
            throw new NumberFormatException("not a number");
        }
    }
}
```

Instrukcja synchronizująca

Instrukcja synchronizująca ma postać

```
synchronized (lock) {
    block
}
```

w której ***lock*** jest odnośnikiem do obiektu ***synchronizatora***, a ***block*** jest ciałem instrukcji, wyznaczającym ***sekcję krytyczną***.

Jeśli pewien wątek wykona instrukcję synchronizującą z synchronizatorem ***lock***, to każdy inny wątek, którego sterowanie napotka instrukcję synchronizującą dotyczącą **tego samego** synchronizatora, zostanie ***zatrzymany*** do chwili, gdy pierwszy wątek zakończy wykonywanie instrukcje ***block***.

Przykład

```
Object buffer = new Object();
// ...
synchronized (buffer) {
    // ...
}
```

dla dociekliwych

Instrukcje synchronizujące mogą być ***zagnieżdżone***. Jeśli w bloku instrukcji synchronizującej posługującej się pewnym synchronizatorem dojdzie do wykonania instrukcji synchronizującej posługującej się **tym samym** synchronizatorem, to **nie spowoduje** to zatrzymania wątku.

```
synchronized (buffer) {

    // ...

    synchronized (buffer) {
        // ...
    }
}
```

Funkcje

Funkcją jest wyodrębniony i nazwany ***opis czynności***. Deklaracja funkcji składa się z ***nagłówka*** opisującego jej zewnętrzne właściwości, a mianowicie

1. Kto może z niej korzystać (np. **public**, każdy).
2. Czy jest funkcją statyczną czy nie-statyczną.
3. Czy i jakiego dostarcza rezultatu (np. **int**, całkowitego).
4. Jaką ma nazwę i jakie jest jej przeznaczenie (np. **divide**, wykonuje dzielenie).
5. Jakiego typu są jej parametry (np. **int, int**, oba typu całkowitego).
6. Czy wysyła wyjątki (np. **throws ArithmeticException**, tak)

oraz z ***ciała*** wyszczególniającego czynności.

Ciało funkcji

Ciałem funkcji jest ***blok*** ujęty w nawiasy **klamrowe**. Blok zawiera ***deklaracje zmiennych*** i ***instrukcje***. Zabrania się, aby ciało funkcji zawierało definicję funkcji.

W deklaracjach zmiennych nie wolno używać specyfikatorów dostępu, a ponadto należy zapewnić, aby każde odwołanie do zmiennej było poprzedzone przypisaniem jej wartości.

```
public void fun()
{
    int fix1 = 10, fix2;
    System.out.println(fix1);  // dobrze
    System.out.println(fix2);  // błąd
}
```

Lokalność

Nazwy zadeklarowane w ***ciele*** funkcji są nazwami jej zmiennych ***lokalnych***. Nazwa lokalna zadeklarowana w pewnym bloku jest widoczna od miejsca jej zadeklarowania, do końca **tego** bloku.

Jeśli w bloku zadeklarowano nazwę ***name***, to zabrania się, aby w jakimkolwiek **zawartym** w nim bloku zadeklarowano taką samą nazwę. Nie zabrania się natomiast deklarowania takich samych nazw w **rozłącznych** blokach.

Jeśli funkcja ma ***parametry***, to z punktu widzenia podanego zakazu, należy je uznać za zadeklarowane na początku najbardziej zewnętrznego bloku funkcji.

```
public void fun(int a)
{
    int a = 10;              // błąd
    {
        int b = 20;

        {
            int c = 30;      // ok
            int b = 40;      // błąd
        }

        {
             int c = 50;     // ok
        }
    }
}
```

Nielokalność

Nazwami ***nielokalnymi*** są nazwy zadeklarowane **w ciele klasy**, ale **na zewnętrz funkcji**. Posługiwanie się **nielokalnymi** nazwami zmiennych umożliwia ***komunikowanie*** się funkcji.

Jeśli w klasie zadeklarowano nielokalną nazwę ***name***, a w funkcji tej klasy także zadeklarowano nazwę ***name***, to nazwa zawarta w funkcji ***przesłania*** nazwę nielokalną.

Do obiektowej nazwy nielokalnej, przesłoniętej przez nazwę lokalną, można się odwoływać przez **this.*name***. Natomiast do klasowej nielokalnej nazwy klasy ***Name***, przesłoniętej przez nazwę lokalną, można się odwoływać przez ***Name.name*** albo przez ***ref.name*** (jeśli ***ref***, np. **this**, jest odnośnikiem typu ***Name***).

```
public
class Program {

    public static void main(String[] args)
    {
        new Program(10, 20);
    }

    private int x, y;        // nazwy nielokalne

    public Program(
               int x, int y  // nazwy lokalne
           )
    {
        this.x = x;
        this.y = y;
        int minXY =          // nazwa lokalna
            Math.min(x, y);
    }

    // ...
}
```

Widoczność

W każdym miejscu klasy jest ***widoczna*** nazwa, której deklaracja nie została przesłonięta.

W klasie pochodnej są widoczne wszystkie nie przesłonięte składniki widoczne w jej klasie **bazowej**, a w klasie wewnętrznej są widoczne wszystkie nie przesłonięte składniki widoczne w klasie ją **zawierającej**.

Jeśli w klasie bazowej jest widoczne pole ***field***, ale w klasie pochodnej nazwa ta została **przesłonięta**, to do przesłoniętego pola można się odwoływać za pomocą kwalifikowanej nazwy **super.*field***. Nie ma natomiast możliwości odwoływania się do składników widocznych w innej niż **bezpośrednia** klasie bazowej (np. przez **super.super.*field***).

```
class Person {

    protected String fullName;  // nazwisko
    protected int age;

    public String getName()
    {
        return fullName;
    }

    // ...
}

class Woman
    extends Person {

    private boolean isMarried;
    private String fullName;  // nazwisko męża

    public int getAge()
    {
            // pole Person.age jest tu widoczne
            // jego pełną nazwą jest
            // this.age albo super.age
        return age;
    }

    public String getBoth()
    {
            // pole Person.fullName jest tu niewidoczne
            // dlatego odwołano się do niego
            // poprzez widoczną funkcję oraz
            // poprzez kwalifikator super
        return getName() + super.fullName;
    }

    // ...
}
```

Przeciążenie funkcji

Każda funkcja wchodzi w skład pewnej klasy. Klasa może zawierać więcej niż jedną funkcję o takiej samej nazwie, na przykład

```
public int sum(int par1, int par2, int par3)
{
    return par1 + par2 + par3;
}
```

oraz

```
public int sum(int par1, int par2)
{
    sum(par1, par2, 0);
}
```

byle tylko każda para takich funkcji ***przeciążonych*** różniła się ***sygnaturami***.

Sygnaturę funkcji otrzymuje się z jej **nagłówka**, pozostawiając jedynie **nazwę** funkcji i **listę typów** parametrów ujętą w nawiasy okrągłe. W szczególności, sygnaturą funkcji o nagłówku

```
public static double divide(int parOne, int parTwo)
```

jest **divide(int, int)**.

Dostarczenie rezultatu

Nagłówek funkcji podaje typ jej **rezultatu**. Każde wywołanie funkcji można traktować jako jego **nazwę**.

Jeśli w miejscu określenia typu znajduje się słowo kluczowe **void**, to funkcja jest ***bez-rezultatowa***. Powrót z wykonywania funkcji bez-rezultatowej następuje w chwili wykonania instrukcji powrotu nie zawierającej wyrażenia.

Instrukcję powrotu **bez wyrażenia** domniemywa się tuż przed klamrą kończącą ciało funkcji bez-rezultatowej. W wielu wypadkach umożliwia to **zrezygnowanie** z jawnego użycia takiej instrukcji.

Jeśli typ rezultatu funkcji jest różny od **void**, to funkcja jest ***rezultatowa*** i musi zawierać instrukcję powrotu zawierającą **wyrażenie**. Wykonanie takiej instrukcji powoduje zakończenie wykonywania funkcji rezultatowej.

Wyrażenie jest opracowywane ***od-lewej-do-prawej***, a jego wartość przypisuje się anonimowemu **rezultatowi funkcji**. Wymaga się, aby takie przypisanie było **wykonalne**.

W szczególności, w zasięgu definicji

```
public static int sqr(int par)
{
    return par * par;
}
```

wyrażenie

```
sqr(sqr(3))
```

jest nazwą anonimowej zmiennej typu **int** o wartości **81**.

Wyszczególnienie czynności

Wyszczególnienie czynności wykonywanych przez funkcję jest zawarte w ***ciele funkcji***.

Ciałem funkcji jest wnętrze **instrukcji grupującej**. Ciało musi zawierać jawną albo domniemaną ***instrukcję powrotu*** (*return*). Jej wykonanie powoduje zakończenie wykonywania czynności opisanych przez funkcję i ***powrót sterowania*** w miejsce jej wywołania.

W miejscu wywołania ***funkcji rezultatowej*** jest dostarczany jej **rezultat**. Jego ***wartość*** określa wyrażenie występujące w instrukcji powrotu.

Wyrażenie występujące w instrukcji powrotu może być poddane niejawnej konwersji **standardowej** (np. z typu **int** do **double**).

W szczególności następującą funkcję

```
public double divide(int parOne, int parTwo)
    throws ArithmeticException
{
        // sprawdzenie dzielnika
    if (parTwo == 0)
         throw new ArithmeticException("Division by 0");

        // wykonanie dzielenia
    double result = (0.0 + parOne) / parTwo;

        // dostarczenie rezultatu
    return result;   // instrukcja powrotu
}
```

zadeklarowano jako

1. Publiczną, to jest dostępną **każdemu** składnikowi programu.
2. Nie-statyczną, a więc metodę.
3. Rezultatową, o rezultacie typu **double**.
4. Występującą pod nazwą **divide**.
5. Dwuparametrową, o obu parametrach typu **int**.
6. Wysyłającą wyjątek klasy **ArithmeticException**.

Z analizy ciała funkcji wynika, że po zainicjowaniu parametrów argumentami, dzieli ona pierwszy parametr przez drugi, a rezultat dzielenia dostarcza w zmiennej typu **double**. W szczególności, jeśli funkcja **divide** zostanie wywołana w instrukcji

```
double result = divide(8, 2);
```

to zmienna **result** otrzyma wartość **4.0**.

Jeśli dzielenie odbywa się przez **0**, to z miejsca jego wykonania funkcja wysyła wyjątek klasy **ArithmeticException**. W takim wypadku **nie istnieje** rezultat funkcji.

Metody i sposoby

Funkcje dzielą się na ***metody***, ***sposoby*** i ***konstruktory***.

Metodą jest funkcja zadeklarowana bez specyfikatora **static**. Sposobem jest funkcja zadeklarowana ze specyfikatorem **static**, a konstruktorem jest funkcja o nazwie identycznej z nazwą zawierającej ją klasy, zadeklarowana bez określenia typu rezultatu.

Deklaracja funkcji zaczyna się od nagłówka

```
specs type fun(dcl, dcl, ... , dcl)
```

w której ***specs*** jest zestawem specyfikatorów (w tym m.in. **public** oraz **static**), ***type*** określa typ rezultatu (m.in. **void**), a każde ***dcl*** jest deklaracją zmiennej stanowiącej ***parametr*** funkcji(np. **int[] vec**).

```
class Point {

    private int x, y;           // pola obiektowe
    public static Point p0;     // pole klasowe

    public Point(int x, int y)  // konstruktor
    {
        // ...
    }

    public int getX()           // metoda
    {
        return x;
    }

    // ...

    public static Point getP0()  // sposób
    {
        return p0;
    }
}
```

Wywołanie funkcji

Wywołanie funkcji ma na celu wykonanie czynności określonych przez jej ***ciało*** oraz ***argumenty***.

Argumentami funkcji są ***wyrażenia***. Tuż przed wywołaniem funkcji każde wyrażenie jest opracowywane ***od-lewej-do-prawej***, a jego wartość przypisuje się parametrowi funkcji. Począwszy od tego momentu, wszystkie operacje wykonywane na parametrach nie dotyczą już argumentów funkcji.

Wywołanie **metody** ma postać

```
ref.fun(arg, arg, ... , arg)
```

w której ***ref*** jest nazwą odnośnika do obiektu, któremu jest wydawane polecenie ***fun*** z argumentami ***arg***.

Natomiast **zalecane** wywołanie **sposobu** ma postać

```
Name.fun(arg, arg, ... , arg)
```

w której ***Name*** jest nazwą klasy, a każde ***arg*** jest ***argumentem*** wywołania funkcji ***fun***.

Jeśli argumentem funkcji jest odnośnik, to zmienna obiektowa identyfikowana przez przypisane mu odniesienie jest ***argumentem obiektowym***.

Przykład

W instrukcji

```
fun(new String("Ewa"))
```

argumentem jest anonimowy odnośnik, którego nazwą jest fabrykator, a **argumentem obiektowym** jest obiekt reprezentujący łańcuch "Ewa".

Dobranie definicji

Do każdego wywołania funkcji dobiera się definicję, która ***najlepiej*** pasuje do tego wywołania, a następnie każdy parametr funkcji ***kojarzy się*** z odpowiadającym mu argumentem.

Przykład 1

W zasięgu deklaracji

```
void fun(int par);
```

```
void fun(double par);
```

do wywołania

```
fun(12)
```

*dobiera się funkcję z parametrem typu **int** (ponieważ pasuje ona do wywołania **lepiej** niż funkcja z parametrem typu **double**), natomiast do wywołania*

```
fun(12.0)
```

*dobiera się funkcję z parametrem typu **double**.*

Przykład 2

W zasięgu deklaracji

```
void set(int[] par, double val)
{
    par[0] = (int)val;
}

void set(Object par, int val)
{
    set((int[])par, (double)val);
}

int[] tab = { 10 };
```

do wywołania

```
set(tab, 20);  // wywołanie niejednoznaczne
```

*nie udaje się dobrać funkcji, ponieważ pasują do niego **obie** funkcje, ale **żadna** nie pasuje najlepiej.*

Skojarzenie parametru z argumentem

Skojarzenie parametru ***par*** z argumentem ***arg*** realizuje się jako **zainicjowanie** parametru **wartością** skojarzonego z nim argumentu (nigdy argumentu obiektowego!)

```
par = arg
```

Skojarzenie uznaje za poprawne tylko wówczas, gdy przypisanie jest **poprawne** oraz nie wymaga niejawnego zastosowania ***konwersji zawężającej*** (np. z typu **long** do **int**).

Jeśli parametr jest **nie**-odnośnikowy, to skojarzony z nim argument musi być typu nie-odnośnikowego, a jeśli jest odnośnikowy, to argument musi być typu odnośnikowego.

Przykład 1

W zasięgu deklaracji

```
void fun(char par);
```

jest *poprawna instrukcja*

```
char val = '0' + 0;  // domyślnie: char val =(char)('0' + 0);
```

ale ***nie jest*** *poprawna instrukcja*

```
fun('0' + 0);        // blad: nie stosuje sie domniemania
```

Przykład 2

Jeśli w zasięgu deklaracji

```
void fun(String par);
```

wykona się instrukcję

```
fun(new String("Ewa"));
```

to w ramach skojarzenia parametru z argumentem, parametrowi ***par*** *zostanie przypisane odniesienie dostarczone przez fabrykator.*

Skojarzenia nie-odnośnikowe

Operacja wykonana na parametrze nie-odnośnikowym dotyczy tylko parametru. A zatem nie ma **żadnego** wpływu na wartość argumentu skojarzonego z tym parametrem.

Dla parametru nie-odnośnikowego **nie istnieje** argument obiektowy.

Przykład

W zasięgu deklaracji

```
void set(int par, int val)
{
    par = val;
}
```

wykonanie instrukcji

```
int num = 10;
```

```
set(num, 20);
```

nie powoduje zmiany *wartości zmiennej* ***num***.

Skojarzenia odnośnikowe

Na parametrze odnośnikowym można wykonywać operacje **indeksowania** (np. **par[i]**) i **wybierania** (np. **par.age**), operacje

```
==
!=
instanceof
(Type)
```

oraz operacje wyrażone za pomocą ***przypisań*** (tylko **=**), ***poleceń*** i ***wywołań***.

Ponadto, jeśli parametr ***par*** jest odnośnikiem do tablicy, to można się posługiwać zapisem ***par***.**length**, stanowiącym nazwę anonimowej zmiennej, której wartość określa rozmiar tablicy.

W szczególności, jeśli zdefiniowano funkcję

```
public void fun1(Point par)
{
    par = new Point(20, 20);
}
```

a następnie wywołano ją w kontekście

```
Point point = new Point(10, 10);
fun1(point);
```

to po powrocie z funkcji **fun1** odnośnikowi **point** będzie w dalszym ciągu przypisane odniesienie do obiektu reprezentującego punkt o współrzędnych **(10, 10)**.

Natomiast, jeśli zdefiniowano funkcję

```
public void fun2(Point par)
{
    par.x = par.y = 20;
}
```

a następnie wywołano ją w kontekście

```
Point point = new Point(10, 10);
fun2(point);
```

to po powrocie z funkcji **fun2**, odnośnikowi **point** będzie przypisane odniesienie do obiektu reprezentującego punkt o współrzędnych **(20, 20)**.

Skojarzenie parametru z argumentem

Wykonanie operacji

```
ref.fun(..., arg, ... )
```

albo

```
Name.fun(... , arg, ... )
```

na parametrze ***par***, z którym skojarzono argument ***arg*** nie powoduje zmiany wartości tego argumentu, ale jeśli parametr jest odnośnikowy, to może powodować zmianę skojarzonego z nim argumentu **obiektowego**.

Przykład

W zasięgu deklaracji

```
void set(int[] par, double val)
{
    par[0] = (int)val;
}

void set(int[] par, int val)
{
    set(par, (double)val);
}

int[] tab = { 10 };
```

wykonanie instrukcji

```
set(tab, 20);
```

powoduje zmianę *wartości elementu tablicy identyfikowanej przez* ***tab***.

Przypisanie danej parametrowi

Wykonanie operacji

```
par = ...
```

w której ***par*** jest parametrem, **nie powoduje** zmiany wartości skojarzonego z nim argumentu, ani argumentu obiektowego.

Przykład

W zasięgu deklaracji

```
void set(String par)
{
```

```
    par = "Iza";
}
```

wykonanie instrukcji

```
String name = "Ewa";
set(name);
```

nie powoduje zmiany *wartości odnośnika* ***name****, ani nie powoduje zmiany wartości obiektu reprezentującego łańcuch* ***"Ewa"****.*

Anatomia poleceń

Wydaniem polecenia jest wywołanie metody

***ref.fun*(*arg*, *arg*, ... , *arg*)**

W ramach wykonania tej czynności następuje

1. Utworzenie odnośnika **this** i przypisanie mu wartości odnośnika ***ref***.
2. Skojarzenie parametrów funkcji ***fun*** z odpowiadającymi im argumentami ***arg***.
3. Wykonanie ciała funkcji.
4. Jeśli funkcja jest rezultatowa, dostarczenie rezultatu.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        Value value = new Value(10);

        value.show();  // 10
        value.add(2);
        value.show();  // 12
    }
}

class Value {

    private int value;

    public Value(int value)
    {
        this.value = value;
    }

    public void add(int par)
    {
        value += par;
    }
```

```
    public void show()
    {
        System.out.println(value);
    }
}
```

Funkcje rekurencyjne

Funkcja jest ***rekurencyjna***, jeśli istnieje takie jej wywołanie, że jeszcze przed wykonaniem w niej instrukcji powrotu, ta sama funkcja zostanie wywołana ponownie.

Zastosowanie rekurencji zazwyczaj upraszcza zapis algorytmu, ale może spowodować wydłużenie czasu obliczeń i zwiększenie zużycia pamięci operacyjnej.

Silnia

Funkcję ***factorial(n)*** definiuje się jako iloczyn

n * factorial(n-1)

przy założeniu, że ***factorial(1)*** ma wartość ***1***.

Ze względu na ograniczony rozmiar zmiennych typu **int**, następujące rozwiązania są słuszne tylko dla argumentów z przedziału **1 .. 12**. W celu poszerzenia tego przedziału należałoby zamiast typu **int** użyć na przykład typu **long** albo **double**.

wersja rekurencyjna

```
public static int factorial(int arg)
{
    if (arg == 1)
        return 1;
    return arg * factorial(arg-1);
}
```

wersja iteracyjna

```
public static int factorial(int arg)
{
    int result = 1;     // ustaw na 1
    for (int i = 1; i < arg+1 ; i++)
        result *= i;    // pomnóż przez i
    return result;
}
```

wersja obliczeniowa

```
public static int factorial(int arg)
{
    return (int)(
                1+Math.pow(arg/Math.E, arg) *
                Math.sqrt(2*Math.PI*arg)*(1+1/(12.0*arg))
                 );
}
```

Ciąg Fibonacciego

Funkcję ***fibonacci(n)*** definiuje się jako sumę

fibonacci(n-1) + fibonacci(n-2)

przy założeniu, że ***fibonacci(1)*** i ***fibonacci(2)*** mają odpowiednio wartości **1** i **2**.

Ze względu na ograniczony rozmiar zmiennych typu **int**, następujące rozwiązania są słuszne tylko dla argumentów z przedziału **1 .. 45**. W celu poszerzenia tego przedziału należałoby zamiast typu **int** użyć typu **long** albo **double**.

wersja rekurencyjna

```
public static int fibonacci(int arg)
{
    if (arg < 3)
        return arg;
    return fibonacci(arg-1) + fibonacci(arg-2);
}
```

wersja iteracyjna

```
public static int fibonacci(int arg)
{
    if (arg < 3)
        return arg;
    int befLast = 1,
        last    = 2;
    for (int i = 0; i < arg-2 ; i++) {
        int tmp = last;
        last += befLast;
        befLast = tmp;
    }
    return last;
}
```

wersja obliczeniowa

```
public static int fibonacci(int arg)
{
    double sqrt5 = Math.sqrt(5);
    return (int)(
                (Math.pow(((1 + sqrt5) / 2), arg +1) -
                 Math.pow(((1 - sqrt5) / 2), arg)+1) / sqrt5
            );
}
```

Biblioteki

Poza działaniami podstawowymi, wyrażanymi przez **operatory**, często zachodzi potrzeba wykonania działań złożonych, wyrażanych przez funkcje ***biblioteczne***. Do tej kategorii należy wyznaczenie ***pierwiastka***, ***potęgi*** i ***logarytmu***, wyznaczenie wartości funkcji ***trygonometrycznych*** oraz wykonanie operacji na ***łańcuchach znaków***.

Funkcje arytmetyczne

klasa Math

```
int abs(int val)
double abs(double val)
Dostarcza wartość bezwzględną val.
np. int val = Math.abs(a+b);

int min(int val1, int val2)
double min(double val1, double val2)
Dostarcza minimum z val1 i val2.
np. int val = Math.min(a, b);

int max(int val1, int val2)
double max(double val1, double val2)
Dostarcza maksimum z val1 i val2.
np. int val = Math.max(a, b);
```

Następujący program zawiera funkcję, która dostarcza wartość najmniejszego elementu tablicy identyfikowanej przez jej parametr.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }
```

```
    public Program()
    {
        System.out.println(
            getMin(
                new int[] { 20, 10, 30 }  // 10
            )
        );
    }

    public int getMin(int[] par)
    {
        int min = par[0];
        for (int i = 1 ; i < par.length ; i++)
            min = Math.min(min, par[i]);
        return min;
    }
}
```

Funkcje matematyczne

klasa Math

double **sqrt**(double val)
Dostarcza pierwiastek z ***val***. Jeśli argument jest ujemny, to dostarcza **Double.NaN** (***N****ot* ***A*** ***N****umber*).
np. double val = Math.sqrt(64);

double **exp**(double val)
Dostarcza potęgę o podstawie ***e*** (ok. **2.718 ...**), zdefiniowanej jako **Math.E**.
np. double val = Math.exp(2);

double **pow**(double base, double exp)
Dostarcza potęgę o podstawie ***base*** i wykładniku ***exp***. Jeśli ***base*** ma wartość **0**, a ***exp*** jest ujemne albo nie-całkowite, to dostarcza **Double.NaN**.
np. double val = Math.pow(Math.E, 2);

double **log**(double val)
Dostarcza logarytm naturalny ***val***. Jeśli argument jest niedodatni, to dostarcza **Double.NaN**.
np. double val = Math.log(Math.E);

double **sin**(double val)
Dostarcza sinus ***val***, przyjmując że argument funkcji wyrażono w radianach.
np. double val = Math.sin(Math.PI / 2);

double **cos**(double val)
Dostarcza cosinus ***val***, przyjmując że argument funkcji wyrażono w radianach.
np. double val = Math.cos(30 * Math.PI / 180);

```
double tan(double val)
Dostarcza tangens val, przyjmując że argument funkcji wyrażono
w radianach.
np. double val = Math.tan(Math.PI / 4);

double random()
Dostarcza liczbę pseudo-przypadkową z przedziału [0, 1).
np. int random = (int)(Math.random() * 256);  // 0 .. 255
```

Math.PI radianów reprezentuje ten sam kąt co **180** stopni. A więc w celu wyznaczenia cosinusa **30** stopni, należy funkcję **cos** wywołać z argumentem **Math.PI / 6**.

Następujący program zawiera funkcję, która orzeka, czy suma kwadratów sinusa i cosinusa tego samego kąta istotnie ma wartość **1**.

W *Matematyce* taka suma **zawsze** ma wartość **1**. Natomiast w *Informatyce*, ze względu na przybliżone reprezentowanie danych rzeczywistych i stosowanie zaokrągleń, wcale tak być nie musi!

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    public Program()
    {
        double arg;
        int count = 0;

        while (true) {
            count++;
            arg = Math.random();
            if (!isOne(arg))
                break;
        }

        System.out.println(
            "Failed for " + arg +
            " after " + count + " tries"
        );
    }

    public boolean isOne(double par)
    {
        double sin = Math.sin(par),
               cos = Math.cos(par);
        return sin * sin + cos * cos == 1;
    }
}
```

Funkcje łańcuchowe

Funkcje łańcuchowe występują w klasach **String** i **StringBuffer**. W odróżnieniu od łańcuchów klasy **String**, łańcuchy klasy **StringBuffer** są **modyfikowalne**.

klasa String

```
String(StringBuffer sb)
Konstruuje obiekt klasy String reprezentujący ten sam łańcuch co sb.
np. String string =
        new String(
            new StringBuffer("Ewa")
        );

char charAt(int pos)
Dostarcza znak, który w łańcuchu występuje na pozycji pos. Jeśli
w łańcuchu nie ma takiej pozycji, to wysyła wyjątek klasy IndexO-
utOfBoundsException.
np. char chr = "Kaja".charAt(2);          // j

int indexOf(int chr)
Dostarcza indeks pierwszego znaku o kodzie chr występującego w łań-
cuchu. Jeśli w łańcuchu nie ma takiego znaku, to dostarcza -1.
np. int pos = "Kaja".indexOf('a');        // 1

int indexOf(String substr)
Dostarcza indeks pierwszego znaku, od którego w łańcuchu zaczyna
się podłańcuch substr. Jeśli w łańcuchu nie ma takiego podłańcucha,
to dostarcza -1.
np. int pos = "Kaja".indexOf("ja");       // 2

boolean endsWith(String substr)
Dostarcza orzeczenie o wartości: "łańcuch kończy się pod-łańcuchem
substr".
np. boolean endsWithSlashes = "Kaja//".indexOf("//");      // true

String substring(int from, int to)
String substring(int from)
Dostarcza podłańcuch, począwszy od znaku o indeksie from aż do
znaku o indeksie to-1 włącznie; domyślnie: do ostatniego znaku.
Jeśli w łańcuchu nie ma pozycji od from do to-1 włącznie, to wysyła
wyjątek klasy IndexOutOfBoundsException.
np. String sub = "Kaja".substring(1, 3);  // "aj"

String trim()
Dostarcza łańcuch pozbawiony początkowych i końcowych znaków o ko-
dach 0x20 i mniejszych. W szczególności usuwa początkowe i końcowe
spacje, tabulacje i znaki sterujące.
np. String string = "  Ewa  ".trim();  // "Ewa"
```

Następujący program zawiera funkcję, która dostarcza liczbę **słów** występujących w łańcuchu znakowym.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    public Program()
    {
        System.out.println(
            "No of words: " +
            noOfWords("  Ewa  Iza  Jan ")
        );
    }

    public int noOfWords(String par)
    {
        int count = 0, pos;

        while (true) {
            par = par.trim();
            if (par.equals(""))
                break;

            count++;
            pos = par.indexOf(' ');
            if (pos == -1)
                break;
            par = par.substring(pos);
        }

        return count;
    }
}
```

klasa StringBuffer

```
void setCharAt(int index, char chr)
Na pozycji index zmienia znak na chr. Jeśli taka pozycja nie istnie-
je, wysyła wyjątek klasy IndexOutOfBoundsException.
np. StringBuffer string = new StringBuffer("Hello World");
    string.setCharAt(5, '*');
    String result = new String(string);    // Hello*World

StringBuffer append(String string)
Wydłuża łańcuch o łańcuch string.
np. StringBuffer string = new StringBuffer("Hello");
    string.append("World");
    String result = new String(string);    // HelloWorld
```

```
StringBuffer append(char chr)
Wydłuża łańcuch o znak chr.
np. StringBuffer string = new StringBuffer("Hello");
    string.append('*').append("World");
    String result = new String(string);    // Hello*World

StringBuffer insert(int offset, char chr)
Na pozycji offset wstawia znak chr. Jeśli taka pozycja nie istnie-
je, wysyła wyjątek klasy IndexOutOfBoundsException.
np. StringBuffer string = new StringBuffer("HelloWorld");
    string.insert(5, '*');
    String result = new String(string);    // Hello*World"

StringBuffer insert(int offset, String string)
Na pozycji offset wstawia łańcuch string. Jeśli taka pozycja nie
istnieje, wysyła wyjątek klasy StringIndexOutOfBoundsException.
np. StringBuffer string = new StringBuffer("HelloWorld");
    string.insert(5, "***");
    String result = new String(string);    // Hello***World"
```

Funkcje konwersyjne

Funkcjami konwersyjnymi są funkcje umożliwiające **przekształcenie** danej pewnego typu w daną innego typu.

klasa Integer

```
int parseInt(String str)
    throws NumberFormatException
Dostarcza liczbę zawartą w zapisie łańcucha str. Jeśli łańcuch nie
przedstawia liczby całkowitej bez znaku albo liczby całkowitej
poprzedzonej znakiem - (minus), to wysyła wyjątek klasy NumberFor-
matException.
np. int val = Integer.parseInt("-12");  // -12

String toHexString(int val)
Dostarcza łańcuch składający się z cyfr liczby szesnastkowej o ta-
kiej samej wartości jak val.
np. String hex = Integer.toHexString(12);  // "c"
```

klasa Double

```
double parseDouble(String str)
    throws NumberFormatException
Dostarcza liczbę zawartą w zapisie łańcucha str. Jeśli łańcuch nie
przedstawia liczby rzeczywistej, to wysyła wyjątek klasy NumberFor-
matException.
np. double val = Double.parseDouble("+12.4e-1");  // 1.24
```

Funkcje systemowe

Funkcjami systemowymi są funkcje wykonujące przydatne czynności **usługowe**, implementowane zazwyczaj przez ***funkcje rodzime***.

klasa System

```
void arraycopy(Object[] source, int srcPos,
               Object[] target, int trgPos,
               int length)
Kopiuje length elementów tablicy source, od jej pozycji srcPos, do
tablicy target od jej pozycji trgPos.
np. int src[] = { 10, 20, 30, },
        trg[] = new int[src.length];
    System.arraycopy(src, 0, trg, 0, src.length);
    System.out.println(trg[2]);
```

W przypadku tablicy wielo-poziomowej, kopiowanie dotyczy tylko pierwszego poziomu (jest płytkie!).

```
long currentTimeMillis()
Dostarcza bieżący czas, wyrażony w milisekundach i liczony od po-
czątku epoki (1 stycznia 1970).
np. long time = System.currentTimeMillis();

void exit(int code)
Kończy wykonywanie programu.
np. System.exit(0);
```

Zadanie A

Posługując się następującym wzorem wyznaczyć błąd przybliżenia liczby π

```
π/4 = 1 - 1/3 + 1/5 - 1/7 ...
```

```
package jbPack;

import java.util.*;
import java.text.*;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }
```

```
    public final int Count = 100;

    public Program()
    {
        double pi = 0;
        for (int i = 0; i < Count ; i++) {
            double val = 2*i + 1;
            if (i % 2 == 0)
                pi += 1 / val;
            else
                pi -= 1 / val;
        }
        pi *= 4;

        DecimalFormat df =
            (DecimalFormat)NumberFormat.
                          getInstance(Locale.ENGLISH);
        df.applyPattern("0.00");

        System.out.println(
            "Error = " +
            df.format(100 * (Math.abs(Math.PI - pi) /
                             Math.PI)) +
             '%'
        );
    }
}
```

Zadanie B

Sprawdzić, czy podany łańcuch znaków jest palindromem

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    public final String string =
          "A man, a plan, a canal: panama";

    public Program()
    {
        int count = string.length();
        String foreString = "";

        for (int i = 0; i < count ; i++) {
            char chr = string.charAt(i);
```

```
            if (chr >= 'a' && chr <= 'z' ||
                chr >= 'A' && chr <= 'Z')
                foreString += chr;
        }

        foreString = foreString.toLowerCase();
        count = foreString.length();
        String backString = "";

        for (int i = count-1; i >= 0 ; i--)
            backString += foreString.charAt(i);

        if (backString.equals(foreString))
            System.out.println(
                '\"' + string + '\"' + " is a palindrome"
            );
        else
            System.out.println(
                "String is not a palindrom: " + backString
            );
    }
}
```

Zadanie C

Wyprowadzić cyfry podanej liczby nieujemnej oddzielone spacjami

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    public final int Number = 127;

    public Program()
    {
        String spaceNum = "";
        int num = Number;

        if (num == 0)
            spaceNum = "0";
        else
            while (num != 0) {
                spaceNum = " " + (char)(num % 10 + '0') +
                    spaceNum;
                num /= 10;
            }
```

```
        System.out.println(
           Number + " =>" + spaceNum
        );
    }
}
```

Zadanie D

Przedstawić podaną liczbę nieujemną w postaci dwójkowej

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    public final int Number = 12;

    public Program()
    {
        String binNum = "";
        int num = Number;

        if (num == 0)
            binNum = "0";
        else
            while (num != 0) {
                binNum = num % 2 + binNum;
                num >>= 1;
            }

        System.out.println(
           Number + " is 0" + binNum
        );
    }
}
```

Zadanie E

Przedstawić podaną liczbę nieujemną w postaci szesnastkowej

```
package jbPack;

public
class Program {
```

```
    public static void main(String[] args)
    {
        new Program();
    }

    public final int Number = 127;

    public Program()
    {
        String hexNum = "";
        int num = Number;

        if (num == 0)
            hexNum = "0";
        else
            while (num != 0) {
                char digit = (char)(num % 16 + '0');
                if (digit > '9')
                    digit += (char)('a' - '9' - 1);
                hexNum = "" + (digit + hexNum);
                num >>= 4;
            }

        System.out.println(
           Number + " is 0x" + hexNum
        );
    }
}
```

Zadanie F

Przedstawić podaną liczbę rzymską w postaci dziesiętnej

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    public final String Roman = "MCDXCII";  // 1492

    // Uwaga: zapis krótszy, ale niepoprawny: MXDII

    public Program()
    {
        String digits = "MDCLXVI";
        int[] value = { 1000, 500, 100, 50, 10, 5, 1 };

        int count = Roman.length(),
```

```
            oldPos = -1, number = 0;

        for (int i = 0; i < count ; i++) {
            char chr = Roman.charAt(i);
            int pos = digits.indexOf(chr);
            if (pos < 0) {
                System.out.println(
                    "Wrong letter in: " + Roman
                );
                System.exit(0);
            }
            number += value[pos];
            if ((oldPos == 2 || oldPos == 4 || oldPos == 6) &&
                (pos-oldPos == -1 || pos-oldPos ==-2))
                number -= 2 * value[oldPos];
            oldPos = pos;
        }

        System.out.println(
            Roman + " is " + number
        );
    }
}
```

Zadanie G

Zapisać podaną liczbę w postaci wyrażenia składającego się tylko z cyfr od 1 do 9 (w rosnącej kolejności) oraz z operacji dodawania i odejmowania

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    public int Number = 13;

    public Program()
    {
        for (int i = 0; i < 256 ; i++) {
            int value = 1,
                mask  = i,
                digit = 1;

            for (int j = 0; j < 8 ; j++) {
                digit++;
```

```
                if (mask % 2 == 0)
                    value += digit;
                else
                    value -= digit;
                mask >>= 1;
            }

            if (value == Number) {
                mask = i;
                String result = "1";
                digit  = 1;

                for (int j = 0 ; j < 8 ; j++) {
                    digit++;
                    if (mask % 2 == 0)
                        result += "+" + digit;
                    else
                        result += "-" + digit;
                    mask >>= 1;
                }

                    // 13 = 1-2-3+4-5-6+7+8+9
                System.out.println(
                    Number + " == " + result
                );
                System.exit(0);
            }
        }

        System.out.println("No solution");
    }
}
```

Zadanie H

Zliczyć bity jedynkowe w dwójkowej reprezentacji podanej liczby

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    public int Number = 13;

    public Program()
    {
        int count = 0,
            number = Number;
```

```
        for (int i = 0; i < 32 ; i++) {
            if (number < 0)
                count++;
            number <<= 1;
        }

        System.out.println(
            "Count = " + count  // Count = 3
        );
    }
}
```

Zadanie 1

Zliczyć ilokrotnie następuje zmiana z 0 na 1 i odwrotnie w dwójkowej reprezentacji podanej liczby

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    public int Number = 12;

    public Program()
    {
        int count = 0,
            number = Number,
            lastBit = number & 1;
            bit;

        for (int i = 0; i < 32 ; i++) {
            if ((bit = number & 1) != lastBit)
                count++;
            lastBit = bit;
            number >>= 1;
        }

        System.out.println(
            "Count = " + count  // Count = 2
        );
    }
}
```

Zadanie J

Wyznaczyć wartość liczby, powstałej z podanej liczby, po lustrzanym odbiciu jej dwójkowej reprezentacji

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    public int Number = 2147483647;

    public Program()
    {
        int count = 0,
            number = Number,
            result = 0;

        for (int i = 0; i < 32 ; i++) {
            result <<= 1;
            result += number & 1;
            number >>= 1;
        }

        System.out.println(
            "Result = " + result  // Result = -2
        );
    }
}
```

Zadanie K

Wykonać konwersję szesnastkowo-dziesiętną i dziesiętno-szesnastkową

Następujący program, oparty na szkielecie konsolowym, oczekuje jednej liczby szesnastkowej (np. **0x12ab**), rozpoznaje jej wartość i przypisuje zmiennej **value**. Następnie wykonuje dowolną operację (np. **value <<= 4**), po czym wyprowadza nową wartość **value** w postaci szesnastkowej (np. **0x000012ab0**).

```
package jbPack;

import javax.swing.*;
import java.awt.*;
import java.util.*;
import java.io.*;
```

```
import java.net.*;
import java.text.*;

public
class Program
    extends Console {

    public static void main(String[] args)
    {
        new Program();
    }

    public Program()
    {
        String str = JOptionPane.showInputDialog(
                        "Enter Hex number"
                    );

        int value = 0,
            length = str.length(),
            hex;

        for (int i = 2; i < length ; i++) {
            char chr = str.charAt(i);
            if (chr >= '0' && chr <= '9')
                hex = chr - '0';
            else
                hex = chr - 'a' + 10;
            value = value * 16 + hex;
        }

        // ================================

        value <<= 4;   // dowolna operacja

        // ================================

        String s = "0x";

        for (int i = 0; i < 8 ; i++) {
            hex = (value >> (28 - i*4)) & 0xf;
            if (hex < 10)
               s += (char)('0' + hex);
            else
               s += (char)(hex-10 + 'a');
        }

        JOptionPane.showMessageDialog(
            null, s
        );

        System.exit(0);
    }
}
```

Wejście-wyjście

Operacje wejścia-wyjścia wykonuje się na **plikach**. Z poziomu *Systemu Operacyjnego* plik jest zestawem **bajtów**, ale w programie może być interpretowany jako zestaw ***bajtów***, ***znaków***, ***danych*** albo ***słów***. Operacje na plikach wykonuje się za pośrednictwem ***strumieni***.

Dostęp do elementów pliku może być ***sekwencyjny*** albo ***wyrywkowy***. Dostęp sekwencyjny polega na przetwarzaniu elementów pliku ***jeden-po-drugim***. Dostęp wyrywkowy umożliwia przetwarzanie ich ***w-dowolnej-kolejności***.

Klasa plikowa

Operacje na **plikach** i **katalogach** najłatwiej wykonać poprzez obiekty ***klasy plikowej*** (**File**).

Jeśli nazwa pliku ma zawierać znak **** (*ukośnik*), to z równym skutkiem można używać znaku **/** (*skośnik*). Należy jedynie pamiętać, że ukośnik zawarty w literale znakowym albo łańcuchowym musi mieć postać **** (*ukośnik*, *ukośnik*). A zatem, na przykład nazwę **c:\config.sys** można zapisać jako **"c:\\config.sys"** albo jako **"c:/config.sys"**.

klasa File

File(String fullName)
Konstruuje obiekt klasy **File** reprezentujący ***plik-katalog*** (plik albo katalog) o nazwie ***fullName***.
np. File file = new File("c:/autoexec.bat");

long **length**()
Dostarcza rozmiar pliku (w bajtach).
np. long len = file.length();

boolean **exists**()
Dostarcza orzeczenie o wartości: "*istnieje plik-katalog identyfikowany przez obiekt plikowy*".
np. boolean exists = file.exists();

boolean **isFile**()
Dostarcza orzeczenie o wartości: "*plik-katalog identyfikuje plik*".
np. boolean isFile = file.isFile();

```
boolean isDirectory()
Dostarcza orzeczenie o wartości: "plik-katalog identyfikuje kata-
log".
np. boolean isDir = file.isDirectory();

boolean canRead()
Dostarcza orzeczenie o wartości: "plik-katalog może być odczytany".
np. boolean canRead = file.canRead();

boolean canWrite()
Dostarcza orzeczenie o wartości: "do pliku-katalogu można dokonać
zapisu".
np. boolean canWrite = file.canWrite();

boolean mkdir()
Tworzy katalog (wraz z nad-katalogami) o nazwie reprezentowanej
przez plik-katalog. Dostarcza true tylko wówczas gdy katalog nie
istniał.
np. boolean wasMade = file.mkdir();

boolean renameTo(File trg)
Zmienia nazwę pliku-katalogu na podaną. Dostarcza true tylko wów-
czas gdy przemianowanie się powiodło.
np. boolean wasRenamed = file.renameTo(new File("c:/trash"));

boolean delete()
Usuwa plik-katalog. Dostarcza true, jeśli usunięcie się powiodło.
np. boolean wasDeleted = file.delete();

String[] list()
Dostarcza odniesienie do tablicy nazw plików i katalogów zawartych
w pliku-katalogu. Jeśli plik-katalog nie jest katalogiem, to do-
starcza null.
np. String[] names = file.list();

String getAbsolutePath()
Dostarcza bezwzględną nazwę ścieżki pliku-katalogu.
np. String path = file.getAbsolutePath();

URL toURL()
    throws MalformedURLException
Dostarcza lokalizator utworzony z nazwy pliku-katalogu. Nie bada
czy plik-katalog istnieje (sic!).
np. URL url = file.toURL();
```

Następująca aplikacja oczekuje **2** argumentów: ścieżki ***path*** (np. **c:/MainFrame**) oraz nazwy pliku-katalogu ***name*** (np. **autoexec.bat**). Na tej podstawie informuje czy plik-katalog ***path***/***name*** istnieje, czy jest plikiem czy katalogiem oraz wyznacza jego rozmiar.

```
package jbPack;

import java.io.*;
```

```
public
class Program
    extends Console {

    public static void main(String[] args)
    {
        if (args.length != 2) {
            System.out.println(
                "Usage is: Program path name"
            );
            System.exit(0);
        }

        new Program(args);
    }

    public Program(String[] args)
    {
        String path = args[0],
               name = args[1];

        String fullName = path + "/" + name;

        File file = new File(fullName);

        if (!file.exists()) {
            System.out.println(
                "Path " + fullName + " does not exist"
            );
            System.exit(0);
        }

        String msg = fullName + " is a ";
        if (file.isFile()) {        // plik
            if (file.canRead())
                msg += "readable";
            else
                msg += "not readable";
            msg += " file\n";
            msg += "Length = " + file.length();
        } else  {                   // katalog
            msg += "directory\n";
            msg += "Size = " + dirSize(file);
        }
        println(msg);
        showConsole();

    }

    public long dirSize(File file)
    {
        if (file.isFile())
            return file.length();
        else {
            long size = 0;
```

```
            String[] list = file.list();
            String path = file.getAbsolutePath();
            for (int i = 0; i < list.length ; i++) {
                size += dirSize(
                            new File(path + "/" + list[i])
                        );
            }
            return size;
        }
    }
}
```

Znaki i bajty

Wyprowadzenie znaku w ***kodzie jedno-bajtowym*** (np. **Cp1250**) powoduje umieszczenie w strumieniu tylko jednego bajtu. Wprowadzenie znaku w **takim samym** kodzie jednobajtowym, powoduje odtworzenie jego *Unikodu*. W szczególności wyprowadzenie znaku **'ą'** o *Unikodzie* **0x105** w domyślnym kodzie **Cp1250**, powoduje umieszczenie w strumieniu bajtu o wartości **0x05**, a wprowadzenie go w tym samym kodzie dostarcza znak o *Unikodzie* **0x105**.

Wyprowadzenie znaku w ***kodzie*** *UTF8* powoduje umieszczenie w strumieniu **1**, **2** albo **3** bajtów (dla znaku **'ą'** bajtów **0x0c2** i **0x105)**. Jedno-bajtowo są wyprowadzane znaki o *Unikodach* **\u0001 .. \u00ff**. Dwu-bajtowo jest m.in. wyprowadzany znak o *Unikodzie* **\u0000**, wszystkie **małe** polskie litery oraz litery **Ź** i **Ż**. Wprowadzenie w kodzie *UTF8* znaku zapisanego w kodzie *UTF8* powoduje odtworzenie jego pierwowzoru.

W tabeli ***Kody polskich liter*** przytoczono *Unikody*, kody ***Cp1250*** i kody *UTF8* polskich liter narodowych.

Tabela Kody polskich liter

ą	ć	ę	ł	ń	ó	ś	ź	ż	
105	**107**	**119**	**142**	**144**	**0f3**	**15b**	**17a**	**17c**	**Unikod (2 bajty)**
b9	e6	ea	b3	f1	f3	9c	9f	bf	Cp1250 (1 bajt)
c2	102	102	c2	102	102	c2	c2	c2	UTF8 (pierwszy bajt)
105	0a6	15e	142	0b1	142	15b	17a	17c	(drugi bajt)
Ą	Ć	Ę	Ł	Ń	Ó	Ś	Ź	Ż	
104	**106**	**118**	**141**	**143**	**0d3**	**15a**	**179**	**17b**	**Unikod (2 bajty)**
a5	c6	ca	a3	d1	d3	8c	8f	af	Cp1250 (1 bajt)
41	43	45	4c	4e	4f	53	c2	c2	UTF8 (pierwszy bajt)
							179	17b	(drugi bajt)

Strumienie znakowe-wejściowe

Strumieniem znakowym-wejściowym jest strumień znakowy otwarty ***do-wprowadzania***. W celu wprowadzenia wierszy strumienia należy utworzyć obiekt klasy **BufferedReader**, a następnie wydawać mu polecenia **read** i **readLine**.

Przetwarzanie strumienia odbywa się zgodnie ze schematem zamieszczonym w tabelach ***Wprowadzanie znaków i wierszy***. Przytoczone sekwencje instrukcji należy umieścić w bloku instrukcji *try*.

Tabela Wprowadzanie znaków i wierszy

```
                  // wprowadzanie znaków
FileReader fr =
    new FileReader(fileName);
BufferedReader br =
    new BufferedReader(fr);

int chr;
while ((chr = br.read()) != -1) {
    // ...
}

                  // wprowadzanie wierszy
String line;
while ((line = br.readLine()) != null) {
    // ...
}
```

klasa FileReader

FileReader(String fileName)
FileReader(File file)
Konstruuje obiekt klasy **FileReader** reprezentujący znakowy strumień wejściowy zakodowany domyślnie, oparty na pliku ***fileName*** albo na obiekcie ***file***.
np. FileReader fr = new FileReader("c:/config.sys");

klasa BufferedReader

BufferedReader(Reader r)
Konstruuje obiekt klasy **BufferedReader** reprezentujący znakowy-buforowany strumień wejściowy opisany przez ***r*** (w szczególności **System.in**). Argumentem może być m.in. odnośnik do dowolnego obiektu klasy **FileReader**.

```
np. BufferedReader br =
        new BufferedReader(
            new FileReader("Data.txt")
        );
```

```
int read()
    throws IOException
Wprowadza 1 znak. Jeśli w strumieniu wejściowym już nie ma bajtów,
dostarcza -1.
np. int chr = br.read();

String readLine()
    throws IOException
Wprowadza jeden wiersz. Dostarcza go bez znaków końca ('\n', '\r',
"\r\n"). Jeśli w strumieniu już nie ma znaków, dostarcza null.
np. String line = br.readLine();

int read(char[] buf, int off, int len)
    throws IOException
Wprowadza znaki do tablicy buf, od pozycji off, w liczbie nie
większej niż len. Dostarcza liczbę wprowadzonych znaków. Jeśli
w strumieniu już nie ma znaków, dostarcza -1.
np.  br.read(new char[100], 0, 100);

void close()
    throws IOException
Zamyka strumień.
np. br.close();
```

Następująca aplikacja wprowadza kolejne wiersze pliku **c:\config.sys**, a następnie wyprowadza je na konsolę.

```
package jbPack;

import java.io.*;

public
class Program {

    private String fileName = "c:/config.sys";

    public static void main(String[] args)
    {
        new Program();
    }

    public Program()
    {
        try {
            FileReader fr = new FileReader(fileName);
            BufferedReader br = new BufferedReader(fr);

            int i = 0;
            String line;
            while ((line = br.readLine()) != null)
                System.out.println(
                    i++ + ": " + line
                );
        }
        catch (Exception e) { e.printStackTrace(); }
    }
}
```

Strumienie znakowe-wyjściowe

Strumieniem znakowym-wyjściowym jest strumień znakowy otwarty ***do-zapisu***. W celu wyprowadzenia wierszy do strumienia, należy utworzyć obiekt klasy **BufferedWriter**, a następnie wydawać mu polecenia **write** i **newLine**.

Przetwarzanie strumienia odbywa się zgodnie ze schematem zamieszczonym w tabeli ***Wyprowadzanie danych***. Przytoczoną sekwencję instrukcji należy umieścić w bloku instrukcji *try*.

Tabela Wyprowadzanie danych

```
FileWriter fw =
    new FileWriter(fileName);
BufferedWriter bw =
    new BufferedWriter(fw);

    // ...
bw.write(exp);
    // ...
```

klasa FileWriter

FileWriter(String fileName)
FileWriter(File file)
Konstruuje obiekt klasy **FileWriter** reprezentujący znakowy strumień wyjściowy zakodowany domyślnie, oparty na pliku ***fileName*** albo na obiekcie ***file***.
np. FileWriter fr = new FileWriter("c:/config.sys");

klasa BufferedWriter

BufferedWriter(Writer w)
Konstruuje obiekt klasy **BufferedWriter** reprezentujący znakowy-buforowany strumień wyjściowy, opisany przez ***w*** (w szczególności **System.out**). Argumentem może być m.in. odnośnik do dowolnego obiektu klasy **FileWriter**.

```
np. BufferedWriter bw =
        new BufferedWriter(
            new FileWriter("Data.txt")
        );
```

void **write**(int code)
    throws IOException
Wyprowadza znak o *Unikodzie* ***code***.
np. bw.write('ą');

```
void write(char[] buf, int off, int len)
    throws IOException
Wyprowadza znaki z tablicy buf, od pozycji off, w liczbie len.
np. bw.write(new char[] { 'ą', 'ś' }, 0, 2 );

void write(String str)
    throws IOException
Wyprowadza łańcuch str.
np. bw.write("Łódź");

void newLine()
    throws IOException
Wyprowadza znak końca wiersza.
np. bw.newLine();

void flush()
    throws IOException
Wymiata (wyprowadza do pliku) bufor strumienia.
np. bw.flush();

void close()
    throws IOException
Wymiata i zamyka strumień.
np. bw.close();
```

Zastosowanie buforowania powoduje kompletowanie znaków w pamięci operacyjnej, a nie natychmiastowe ich wyprowadzanie. Dlatego przed zakończeniem wykonania programu należy użyć poleceń **flush** albo **close** wymiatających bufor.

Następująca aplikacja wyprowadza do pliku **Data2.txt** zawartość pliku **c:\config.sys**.

```
package jbPack;

import java.io.*;
import java.util.*;

public
class Program {

    private String fileName = "Data2.txt";

    public static void main(String[] args)
    {
        new Program();
    }

    public Program()
    {
        try {
            FileReader fr =
                new FileReader("c:/config.sys");
            BufferedReader br =
                new BufferedReader(fr);

            FileWriter fw =
                new FileWriter(fileName);
```

```
            BufferedWriter bw =
                new BufferedWriter(fw);

            String line;
            while ((line = br.readLine()) != null) {
                bw.write(line);
                bw.newLine();
            }

            bw.close();  // niezbędne!

            System.out.println("Copy done");
        }
        catch (Exception e) { e.printStackTrace(); }
    }
}
```

Strumienie bajtowe-wejściowe

Strumień bajtowy-wejściowy jest interpretacją zawartości pliku jako sekwencji bajtów. Przetwarzanie strumieni bajtowych wejściowych odbywa się najczęściej za pośrednictwem obiektów klas **BufferedInputStream**.

Wprowadzenie bajtu powoduje dostarczenie znaku z wyzerowanym bajtem bardziej znaczącym. Z tych powodów strumienie bajtowe **nie nadają się** do wyprowadzania/wprowadzania znaków o kodach większych niż **255**.

klasa FileInputStream

```
FileInputStream(String fileName)
    throws FileNotFoundException
FileInputStream(File file)
    throws FileNotFoundException
```

Konstruuje obiekt klasy **FileInputStream** reprezentujący bajtowy strumień wejściowy związany z plikiem ***fileName*** albo z plikiem opisanym przez obiekt plikowy ***file***.

```
np. FileInputStream fis =
        new FileInputStream("c:/autoexec.bat");
```

klasa BufferedInputStream

```
BufferedInputStream(FileInputStream fis)
```

Konstruuje obiekt klasy **BufferedInputStream** reprezentujący bajtowy, buforowany strumień wejściowy.

```
np. BufferedInputStream bis =
        new BufferedInputStream(fis);
```

```
int read()
    throws IOException
Wprowadza 1 bajt. Dostarcza go w mniej znaczącej części wyzerowane-
go rezultatu. Jeśli w strumieniu już nie ma bajtów, dostarcza -1.
np. int chr = bis.read();

int read(byte[] buf, int off, int len)
    throws IOException
Wprowadza bajty do tablicy buf, od pozycji off, w liczbie nie
większej niż len. Dostarcza liczbę wprowadzonych bajtów. Jeśli
w strumieniu już nie ma bajtów, dostarcza -1.
np. int count = bis.read(buffer, 0, 10);

void close()
    throws IOException
Wymiata i zamyka strumień.
np. bis.close();
```

Następująca aplikacja zlicza bajty pliku, którego nazwę określa jej pierwszy argument (np. **c:/autoexec.bat**).

```
package jbPack;

import java.io.*;

public
class Program {

    public static void main(String[] args)
    {
        if (args.length != 1) {
            System.out.println(
                "Usage is: Program source"
            );
        }

        new Program(args);
    }

    public Program(String[] args)
    {
        String source = args[0];
        int count = 0;

        try {
            FileInputStream fis =
                new FileInputStream(source);
            BufferedInputStream bis =
                new BufferedInputStream(fis);

            while (bis.read() != -1)
                count++;
            bis.close();

        } catch (IOException e) { e.printStackTrace(); }
```

```
        System.out.println(
            "File " + source + " contains " +
            count + " characters"
        );
    }
}
```

Strumienie bajtowe-wyjściowe

Strumień bajtowy-wyjściowy jest sekwencją bajtów. Przetwarzanie strumieni bajtowych wyjściowych odbywa się najczęściej za pośrednictwem obiektów klas **BufferedOutputStream**.

Wyprowadzenie **znaku** powoduje umieszczenie w strumieniu tylko jego mniej znaczącego **bajtu**. W szczególności dla znaku **'ą'** o *Unikodzie* **0x105** wyprowadza się bajt o wartości **0x05**.

klasa FileOutputStream

```
FileOutputStream(String fileName)
    throws IOException
FileOutputStream(File file)
    throws IOException
Konstruuje obiekt klasy FileOutputStream reprezentujący bajtowy
strumień wyjściowy związany z plikiem fileName albo z plikiem opi-
sanym przez obiekt plikowy file.
np. FileOutputStream fos =
        new FileOutputStream("c:/MainFrame/autoexec.bat");
```

klasa BufferedOutputStream

```
BufferedOutputStream(FileOutputStream fos)
Konstruuje obiekt klasy BufferedOutputStream reprezentujący bajto-
wy, buforowany strumień wyjściowy.
np. BufferedOutputStream bos =
        new BufferedOutputStream(fos);

void write(int code)
    throws IOException
Wyprowadza do strumienia wyjściowego bajt o wartości code & 0xff.
np. bos.write("\u1234");

void write(byte[] buf, int off, int len)
    throws IOException
Wyprowadza do strumienia wyjściowego bajty z tablicy buf, od pozy-
cji off, w liczbie len.
np. bos.write(new byte[] { \u1234, \u4321 }, 0, 2);

void flush()
    throws IOException
Wymiata bufor strumienia.
```

```
np. bos.flush();

void close()
    throws IOException
Wymiata i zamyka strumień.
np. bos.close();
```

Następująca aplikacja pobiera znaki z pliku określonego przez pierwszy argument (np. **c:/autoexec.bat**), a następnie wyprowadza je do pliku określonego przez drugi argument (np. **d:/auto.txt**).

```
package jbPack;

import java.io.*;

public
class Program {

    public static void main(String[] args)
    {
        if (args.length != 2) {
            System.out.println(
                "Usage is: Program source target"
            );
        }

        new Program(args);
    }

    public Program(String[] args)
    {
        String source = args[0],
               target = args[1];

        File src = new File(source);

        if (!src.exists() ||
           !src.isFile() ||
           !src.canRead()) {
            System.out.println(
                "Can not copy " + source
            );
            return;
        }

        try {
            FileInputStream fis =
                new FileInputStream(src);
            BufferedInputStream bis =
                new BufferedInputStream(fis);

            FileOutputStream fos =
                new FileOutputStream(target);
            BufferedOutputStream bos =
                new BufferedOutputStream(fos);
```

```
            int chr = 0;
            while ((chr = bis.read()) != -1)
                bos.write(chr);
            bis.close();
            bos.close();
        } catch (IOException e) { e.printStackTrace(); }

        System.out.println(
            "Copying of " + source + " done!"
        );
    }
}
```

Strumienie słów

Strumień słów jest interpretacją pliku jako sekwencji ***słów***. Przetwarzanie strumienia słów odbywa się za pomocą obiektów klasy **StreamTokenizer**.

Słowem jest **spójny** ciąg znaków różnych od ***odstępu*** (m.in. spacji). Po każdej stronie słowa może wystąpić dowolna liczba odstępów.

klasa StreamTokenizer

StreamTokenizer(Reader r)
Konstruuje obiekt klasy **StreamTokenizer** reprezentujący zestaw słów zawartych w pliku opisanym przez ***r***.

```
np. StreamTokenizer st =
        new StreamTokenizer(
            new FileReader("c:/config.sys")
        );
```

void **quoteChar**(int quote)
Określa ***kod ogranicznika***, umieszczonego po obu stronach łańcucha zawierającego odstępy (domyślnie **nie ma** ogranicznika).

```
np. st.quoteChar("=");
```

void **eolIsSignificant**(boolean itIs)
Poleca rozpoznawanie i dostarczanie znaków **końca wiersza** (domyślnie znak końca wiersza jest traktowany tak jak inne odstępy).

```
np. st.eolIsSignificant(true);
```

int **nextToken**()
throws IOException

Dostarcza informacji o rodzaju wprowadzonego słowa

1. Jeśli wprowadzono **liczbę**, to dostarcza **TT_NUMBER**, a liczbę umieszcza w **nval**.
2. Jeśli wprowadzono **nie-liczbę**, to dostarcza **TT_WORD**, a odniesienie do słowa umieszcza w **sval**.
3. Jeśli wprowadzono **łańcuch**, to dostarcza **kod ogranicznika**, a odniesienie do niego umieszcza w **sval**.

W pozostałych przypadkach dostarcza **kod znaku.**

Dostarczenie znaku o kodzie **TT_EOL** oznacza napotkanie końca wiersza.
Dostarczenie znaku o kodzie **TT_EOF** oznacza koniec pliku.
np. st.nextToken();

Zaleca się stosowanie następującego schematu przetwarzania

```
int EOF    = StreamTokenizer.TT_EOF,
    EOL    = StreamTokenizer.TT_EOL,
    WORD   = StreamTokenizer.TT_WORD,
    NUMBER = StreamTokenizer.TT_NUMBER;

FileReader fr =
    new FileReader(name);

StreamTokenizer st =
    new StreamTokenizer(fr);

int what;
while ((what = st.nextToken()) != EOF) {
    switch (what) {
        case WORD:
            // użycie st.sval
            break;
        case NUMBER:
            // użycie st.nval
            break;
        case EOL:
            // koniec wiersza
            break;
        default:
            // użycie what
    }
}
```

Następująca aplikacja analizuje plik, określony przez jej parametr (np. **c:\jbJava\Data.txt**), wyprowadzając zawarte w nim słowa, w tym łańcuchy zawarte między parami znaków **#** (*hash*).

Jeśli plik zawiera napis

```
12   Isa   -4.2e2   #John -4.2e2 Mary# 127
Isa#bell#13 $%&
```

to nastąpi wyprowadzenie napisu

```
12.0 Isa -4.2 e2 John -4.2e2 Mary 127.0
End of line #1

Isa bell 13.0 $ % &
End of line #2
```

Na uwagę zasługuje sposób potraktowania „liczby” **-4.2e2** (inaczej w łańcuchu, niż poza nim).

```
package jbPack;

import java.io.*;

public
class Program {

    public static void main(String[] args)
    {
        if (args.length != 1) {
            System.out.println(
                "Usage is: Program source"
            );
            System.exit(0);
        }

        new Program(args);
    }

    static final
    int EOF    = StreamTokenizer.TT_EOF,
        WORD   = StreamTokenizer.TT_WORD,
        NUMBER = StreamTokenizer.TT_NUMBER,
        EOL    = StreamTokenizer.TT_EOL;

    public Program(String[] args)
    {
        String source = args[0];

        FileReader fr;
        try {
            fr = new FileReader(source);

            StreamTokenizer st =
                new StreamTokenizer(fr);

            int what;
            st.eolIsSignificant(true);
            st.quoteChar('#');
            int lineNo = 1;

            while ((what = st.nextToken()) != EOF) {
                switch (what) {
                    case EOL:
                        System.out.print(
                            "\nEnd of line #" +
                            lineNo++ + "\n\n"
                        );
                        break;
                    case NUMBER:
                        System.out.print(st.nval + " ");
                        break;
```

```
                    case WORD:
                        System.out.print(st.sval + " ");
                        break;
                    case '#':
                        System.out.print(st.sval + " ");
                        break;
                    default:
                        System.out.print((char)what + " ");
                }
            }
        }
        catch (FileNotFoundException e) {
            System.out.println("File " + source +
                               " not found");
            return;
        }
        catch (IOException e) { e.printStackTrace(); }
    }
}
```

Strumienie danych

Strumień danych jest interpretacją pliku jako sekwencji ***danych***. Strumienie danych służą do **przenośnego** wyprowadzania i wprowadzania danych typów pierwotnych (np. **long**, ale nie **Long**). Przetwarzanie strumienia danych odbywa się za pomocą obiektów klas **DataInputStream i DataOutputStream**.

klasa DataInputStream

DataInputStream(FileInputStream fis)
Konstruuje obiekt klasy **DataInputStream** reprezentujący wejściowy strumień danych oparty na obiekcie ***fis***.

```
np. DataInputStream dis =
        new DataInputStream(
            new FileInputStream("c:/Data.raw")
        );
```

int **read**(byte[] buf)
throws IOException
Pobiera z pliku i umieszcza w tablicy ***buf***, ciąg kolejnych bajtów. Dostarcza liczbę bajtów. W przypadku napotkania końca pliku dostarcza **-1**.

```
np. int count = dis.read(vec = new byte [3]);
```

type **read*Type***() // np. ***int*** read***Int***()
throws IOException, EOFException
Pobiera z pliku i dostarcza reprezentację danej typu ***Type***: **Boolean, Byte, String, Char, Double, Float, Int, Long, Short.** W przypadku napotkania końca pliku dostarcza **-1**.

```
np. int val = dis.readInt()
```

```
int skipBytes(int drop)
    throws IOException, EOFException
Pobiera z pliku i pomija podaną liczbę bajtów. Dostarcza faktycznie
pominiętą liczbę bajtów. W przypadku napotkania końca pliku dostar-
cza -1.
np. dis.skipBytes(3);
```

klasa DataOutputStream

```
DataOutputStream(FileOutputStream fos)
Konstruuje obiekt klasy DataOutputStream reprezentujący wyjściowy
strumień danych oparty na fos.
np. DataOutputStream dos =
        new DataOutputStream(
            new FileOutputStream("c:/Data.raw")
        );

void write(int val)
    throws IOException
Wyprowadza bajt o wartości val.
np. dos.write('a');

void write(byte[] buf)
    throws IOException
Wyprowadza wszystkie bajty tablicy buf.
np. dos.write(new byte [3] { 'E', 'w', 'a' });

void writeType(type var)  // np. void writeInt(int var)
    throws IOException
Wyprowadza reprezentację zmiennej var typu Type: Boolean, Byte,
String, Char, Double, Float, Int, Long, Short.
np. dos.writeDouble(12.8);

void writeChars(String str)
     throws IOException
Wyprowadza reprezentację wszystkich znaków łańcucha str.
np. dos.write("Hello");
```

Następująca aplikacja **wyprowadza**, a następnie **wprowadza** i **porównuje** z oryginałem, obiekt klasy **Child**, wraz z jego "przyległościami".

```
package jbPack;

import java.io.*;

public
class Program {

    private Child isa = new Child("Isabel", 17);
    private File temp = new File("Temporary");
```

```
    public static void main(String[] args)
    {
        new Program();
    }

    public Program()
    {
        try {
            FileOutputStream fos =
                new FileOutputStream(temp);
            DataOutputStream dos =
                new DataOutputStream(fos);
            isa.write(dos);
            dos.close();

            FileInputStream fis =
                new FileInputStream(temp);
            DataInputStream dis =
                new DataInputStream(fis);
            Child isa2 = new Child();
            isa2.read(dis);

            System.out.println(
                isa2.equals(isa)  // true
            );

            temp.delete();
        } catch (Exception e) { e.printStackTrace(); }
    }
}

class Child {

    private String name;
    private int age;

    public Child(String name, int age)
    {
        this.name = name;
        this.age  = age;
    }

    public Child()
    {
    }

    public boolean equals(Child child)
    {
        return name.equals(child.name) &&
               age == child.age;
    }

    public void write(DataOutputStream dos)
        throws IOException
    {
        dos.writeInt(name.length());
```

```
        dos.writeChars(name);
        dos.writeInt(age);
    }

    public void read(DataInputStream dis)
        throws IOException
    {
        StringBuffer name = new StringBuffer();
        int len = dis.readInt();
        for (int i = 0; i < len ; i++)
            name.append(dis.readChar());
        this.name = new String(name);
        age = dis.readInt();
    }
}
```

Pliki wyrywkowe

Dostęp do pliku jest ***wyrywkowy***, jeśli przetwarzanie jego elementów może się odbywać w **dowolnej** kolejności. Przetwarzanie plików wyrywkowych odbywa się za pomocą obiektów klasy **RandomAccessFile**.

klasa RandomAccessFile

```
RandomAccessFile(String fileName, String mode)
    throws IOException
RandomAccessFile(File file, String mode)
    throws IOException
Konstruuje obiekt klasy RandomAccessFile reprezentujący plik file-
Name albo plik reprezentowany przez obiekt file, otwarty z dostępem
wyrywkowym w trybie mode: "r" tylko do odczytu, "rw" do odczytu
i zapisu.
np. RandomAccessFile raf =
        new RandomAccessFile("VirtualArray", "rw");

long getFilePointer()
    throws IOException
Dostarcza pozycję pliku (pozycją początkową jest 0).
np. long pos = raf.getFilePointer();

long length()
    throws IOException
Dostarcza liczbę bajtów pliku.
np. long fileSize = raf.length();

void seek(long pos)
    throws IOException
Ustawia plik w pozycji pos.
np. raf.seek(raf.length() / 2);
```

Następująca aplikacja tworzy w pliku **Array.raf** "tablicę" obiektów typu **double**.

```
package jbPack;

import java.io.*;

public
class Program {

    private String fileName = "C:/Array.raf";
    private final int Last = 10;

    public static void main(String[] args)
        throws IOException
    {
        new Program();
    }

    public Program()
    {
        try {
            Array array = new Array(fileName, 100);
            for (int i = 0; i < Last ; i++)
                array.write(i * i, i);
            for (int i = Last-1 ; i >= 0 ; i--)
                System.out.println(array.read(i));
        }
        catch (Exception e) { e.printStackTrace(); }
    }
}

class Array {

    private RandomAccessFile raf;

    public Array(String name, int size)
        throws IOException
    {
        raf = new RandomAccessFile(name, "rw");
        for (int i = 0; i < size ; i++)
            raf.writeDouble(0);
        raf.seek(0);
    }
    public void write(double num, long pos)
        throws IOException
    {
        raf.seek(8 * pos);
        raf.writeDouble(num);
    }
    public double read(long pos)
        throws IOException
    {
        raf.seek(8 * pos);
        return raf.readDouble();
    }
}
```

Klasy i obiekty

Klasa jest opisem ***rodziny obiektów***. Opracowanie fabrykatora z nazwą klasy, na przykład

```
new Point(10, 20)
```

powoduje utworzenie **obiektu** klasy. W miejscu opracowania fabrykatora jest dostarczane odniesienie do właśnie sfabrykowanego obiektu.

```
Point p;
p = new Point(10, 20);
```

Hermetyzacja

W deklaracji składnika klasy może wystąpić ***specyfikator dostępu*** do składnika: **private**, **public**, **protected**.

Składnik zadeklarowany ze specyfikatorem **private** jest dostępny tylko z ciała jego klasy.

Składnik zadeklarowany ze specyfikatorem **public** jest dostępny wszędzie.

Składnik zadeklarowany ze specyfikatorem **protected** jest dostępny tylko z ciała jego klasy oraz z ciała jego klasy pochodnej.

Składnik zadeklarowany **bez specyfikatora** dostępu jest dostępny w całym **pakiecie**, do którego należy jego klasa.

Zaleca się, aby **pola** były deklarowane jako ***prywatne*** (**private**) albo ***chronione*** (**protected**), a funkcje jako ***publiczne*** (**public**).

```
package jbPack;

class Primary {

    private   int a;  // pole prywatne
    protected int b;  // pole chronione
    public    int c;  // pole publiczne
              int d;  // pole pakietowe

    // ...
}
```

```
class Derived
    extends Primary {

    public void fun()
    {
       int aa = a,  // błąd
           bb = b,
           cc = c,
           dd = d;
    }

    // ...
}
```

Pola obiektowe i klasowe

Polem ***obiektowym*** jest pole zadeklarowane bez specyfikatora **static**. Każdemu polu obiektowemu odpowiada jeden element obiektu klasy.

Polem ***klasowym*** jest pole zadeklarowane ze specyfikatorem **static**. Pole klasowe opisuje zmienną, która jest **wspólna** dla wszystkich obiektów klasy i istnieje nawet wówczas, gdy nie utworzono **ani jednego** obiektu klasy.

Zmienne opisane przez pola klasowe są tworzone w chwili pierwszego odwołania do składnika klasy. W odróżnieniu od zmiennych lokalnych, są **niejawnie** inicjowane wartościami **0**, **0.0**, **false** albo **null**.

```
class Point {

    private int x, y;            // pola obiektowe

    public static int count;     // pole klasowe

    public Point(int x, int y)
    {
        // ...
    }
}
```

Funkcje obiektowe i klasowe

Funkcją ***konstruktorową*** (konstruktorem) jest funkcja, której nazwa jest identyczna z nazwą klasy i w której definicji nie występuje określenie typu rezultatu.

Funkcją ***obiektową*** jest funkcja nie-konstruktorowa zadeklarowana **bez** specyfikatora **static**.

Funkcją ***klasową*** funkcja zadeklarowana ze specyfikatorem **static**.

Funkcje obiektowe są ***metodami***, a funkcje klasowe są ***sposobami***. Nazwa metody albo sposobu może być identyczna z nazwą pola klasy. Polecenia wydawane obiektom można definiować tylko za pomocą **metod**.

```
class Point {

    private int x, y;
    public int static count;

    public Point(int x, int y)     // konstruktor
    {
        // ...
    }

    public void setX(int x)        // metoda
    {
        this.x = x;
    }

    public static int getCount()   // sposób
    {
        return count;
    }
}
```

Funkcje klasowe

Funkcje ***klasowe*** są opisami **czynności** do wykonania na dostarczonych im **argumentach**.

Funkcje klasowe mogą się odwoływać tylko do **klasowych** pól swojej klasy.

```
System.out.println(
    Math.sqrt(144)     // 12
);
```

Funkcje obiektowe

Funkcje ***obiektowe*** są opisami **poleceń** wydawanych obiektom identyfikowanym przez odnośniki (do czego mogą wykorzystywać argumenty). Jeśli funkcja obiektowa zostanie zadeklarowana ze specyfikatorem **final**, to **nie może** być ***przedefiniowana*** w klasie pochodnej (dotyczy to m.in. klasy **String**).

Przedefiniowanie polega na dostarczeniu w klasie pochodnej, funkcji o identycznej sygnaturze i identycznym typie rezultatu, jakie ma funkcja widoczna w klasie bazowej.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        Value val = new Value();

            // polecenie ustawienia wartości
        val.setValue(127);

        System.out.println(
                // polecenie dostarczenia wartości
            val.getValue()   // 127
        );
    }
}

class Value {

    public int value;

    public int getValue()
    {
        return value;
    }

    public void setValue(int value)
    {
        this.value = value;
    }
}
```

Kwalifikowanie nazw

Nie-kwalifikowane odwołania do składników klasy mogą występować zarówno w ciele metody, jak i w ciele sposobu.

Nie-kwalifikowane odwołanie do składnika **obiektowego** jest niejawnie poprzedzane kwalifikatorem **this** (np. **this.x**), a nie-kwalifikowane odwołanie do składnika **klasowego** jest niejawnie poprzedzane nazwą **klasy** (np. **Point.count**).

Nie-kwalifikowane odwołania mogą występować także w inicjatorach pól, inicjatorach klasowych i w inicjatorach obiektowych.

```
class Point {

    private int x, y;
```

```
    public static int count = 0;

    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
        count++;        // Point.count++;
    }

    public int clearXY_getCount()
    {
        x = y = 0;      // this.x = this.y = 0;
        return count;   // return Point.count;
    }
}
```

Rozpoznawanie klas

Klasą ***zewnętrzną*** jest klasa **nie zawarta** w żadnej innej klasie. Klasa zewnętrzna może być tylko **publiczna** albo **pakietowa**.

```
package jbPack;

class Outer {          // klasa zewnętrzna (pakietowa)

    // ...

}
```

Klasą ***wewnętrzną*** jest klasa zawarta w innej klasie, zadeklarowana bez specyfikatora **static** i zdefiniowana w miejscu, w którym mogłyby wystąpić składniki klasy zawierającej. Klasa wewnętrzna może być **prywatna**, **publiczna**, **chroniona** albo **pakietowa**, ale nie może zawierać składników klasowych (w tym interfejsów!).

```
class Outer {

    private int count = 10;

    public
    class Inner {      // klasa wewnętrzna

        public int getCount()
        {
            return count;
        }
    }
}
```

Klasą ***zanurzoną*** jest klasa zawarta w innej klasie, zadeklarowana ze specyfikatorem **static** i zdefiniowana w miejscu, w którym mogłyby wystąpić składniki klasy zawierającej. Klasa zanurzona może być **prywatna**, **publiczna**, **chroniona** albo **pakietowa**, a w odróżnieniu od klasy wewnętrznej, może zawierać składniki klasowe.

```
class Outer {

    private static int count = 10;

    private static
    class Immersed {     // klasa zanurzona

        public static int count = 20;

        public int getCount()
        {
            return Outer.count + count;
        }
    }
}
```

Klasą ***lokalną*** jest klasa zawarta w ciele funkcji. Klasa lokalna nie może być zadeklarowana ze specyfikatorem ochrony, ani ze specyfikatorem **static**, a fabrykator jej obiektu może wystąpić dopiero **po** jej definicji.

```
class Outer {

    // ...
    public void fun()
    {
        class Local {  // klasa lokalna

            // ...
        }

        Local loc = new Local();

        // ...
    }
}
```

Klasą ***anonimową*** jest klasa niejawnie zdefiniowana tuż **za fabrykatorem**. Definicja klasy anonimowej nie zawiera słowa kluczowego **class** ani fraz **extends** i **implements**. W następującym przykładzie, do wykonania takich samych czynności, użyto klasy **jawnej** i **anonimowej**.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
```

```
            // klasa jawna
            // definicję klasy pogrubiono
        class Value
            extends Object {

            public int val = 127;

            public int getVal()
            {
                return val;
            }
        }

        System.out.println(
            new Value().getVal()
        );

            // klasa anonimowa
            // definicję klasy pogrubiono
        System.out.println(
            new Object() {

                public int val = 127;

                public int getVal()
                {
                    return val;
                }
            }.getVal()
        );
    }
}
```

Nazywanie składników

Nazwą ***obiektowego*** składnika klasy, opatrzonego identyfikatorem ***id***, jest w tej klasie ***this.id***.

```
class Point {

    private int x, y;

    public Point(int x, int y)
    {
        this.x = x;  // pole = parametr;
        this.y = y;  // pole = parametr;
    }

    // ...
}
```

Nazwą ***klasowego*** składnika klasy ***Name***, opatrzonego identyfikatorem ***id***, jest ***Name.id***.

```
class Point {

    private int x, y;
    public static int count;

    public Point(int x, int y)
    {
        // ...
        Point.count = 0;
    }

    // ...
}
```

Nazwą ***elementu*** obiektu opisanego przez pole obiektowe ***field***, należące do obiektu identyfikowanego przez odnośnik ***ref*** jest ***ref.field***.

```
class Point {

    private int x, y;

    public Point(int x, int y)
    {
        // ...
    }

    public void setXY(Point p)
    {
        this.x = p.x;
        this.y = p.y;
    }

    // ...
}
```

Nazwą ***zmiennej*** opisanej przez pole klasowe ***field*** klasy ***Name*** jest ***Name.field***.

```
class Point {

    private int x, y;
    public static int count;

    public Point(int x, int y)
    {
        // ...
    }
```

```
    public void setCount(int c)
    {
        Point.count = c;
    }

    // ...
}
```

Nazwą ***składnika*** widocznego w bezpośredniej klasie bazowej danej klasy, opatrzonego identyfikatorem ***id***, jest **super.*id***.

```
class Point {         // klasa bazowa

    protected static int count;

    // ...
}

class Point3D         // klasa pochodna
    extends Point {

    public static int count;

    public void setCount()
    {
        super.count = count;
    }

    // ...
}
```

Dziedziczenie

Jeśli dysponuje się definicją klasy ***bazowej***, to na tej podstawie można utworzyć definicję jej klasy ***pochodnej***.

Jeśli w deklaracji klasy pochodnej nie użyje się frazy **extends** określającej jej klasę bazową, to domyślnie przyjmie się frazę

```
   extends Object
```

A zatem każda klasa, z wyjątkiem **Object**, jest bezpośrednio albo pośrednio pochodną klasy **Object**.

```
class Primary {          // klasa bazowa
                         // domyślnie pochodna od Object
    // ...
}

class Derived            // klasa pochodna
    extends Primary {    // klasy bazowej Primary

    // ...
}
```

Obiekt klasy pochodnej składa się z takich samych elementów z jakich składa się obiekt klasy bazowej, a **ponadto** zawiera elementy opisane przez pola **obiektowe** zadeklarowane w klasie pochodnej.

W następującym zestawie klas, **Person** jest klasą bazową (niejawnie pochodną od **Object**), a **Woman** jest klasą pochodną od **Person**. Obiekty klasy **Person** składają się z elementów typu **String** i **int**, a obiekty klasy **Woman** składają się z elementów typu **String**, **int** i **boolean**.

```
class Person {

    private String name;
    private int age;

    // ...
}
```

```
class Woman
    extends Person {

    private boolean isMarried;

    // ...
}
```

Odwołania odnośnikowe

Poprzez odnośnik do obiektu klasy pochodnej można się odwoływać do wszystkich jej **widocznych** i **dostępnych** składników, w tym do składników odziedziczonych z jej bezpośredniej oraz z pośrednich klas bazowych.

Składnik, który w klasie pochodnej jest **widoczny** nie musi być w niej **zadeklarowany**. Jeśli nie jest w niej zadeklarowany, to poszukiwanie go odbywa się wzdłuż linii dziedziczenia, w kierunku klasy **Object**.

```
class Person {

    private String name;
    protected int age;

    // ...
}

class Woman
    extends Person {

    // ...

    public void fun()
    {
        name = "Ewa";  // błąd (brak dostępu)
        age = 24;
    }
}
```

Odwołania przez this i super

Poprzez kwalifikator **this** można się odwoływać do składników **widocznych** w danej klasie, a poprzez kwalifikator **super** można się odwoływać do składników widocznych w jej bezpośredniej klasie bazowej. W obu przypadkach jest to możliwe tylko wówczas, gdy składniki są **dostępne**.

```
class One {

    protected int val = 1;
}
```

```
class Two
    extends One {

    protected int val = 2;

    public int getSum(int val)
    {
        return val + this.val + super.val;
    }
}
```

Przedefiniowanie funkcji

Jeśli w klasie pochodnej występuje funkcja, która ma sygnaturę i typ rezultatu **identyczne** z sygnaturą i typem rezultatu funkcji widocznej w klasie bazowej, to ma miejsce ***przedefiniowanie*** funkcji.

Jeśli w klasie bazowej występuje metoda **finalna**, to w klasie pochodnej nie może być ***przedefiniowana***.

Zabrania się, aby funkcja ***przedefiniowująca*** zawierała frazę **throws** wyszczególniająca klasę wyjątku znajdującą się w hierarchii klas **powyżej** klasy wyjątku wyszczególnionej we frazie **throws** funkcji ***przedefiniowywanej***.

Zakaz ten nie dotyczy klas **Error** i **RuntimeException** (oraz ich pochodnych), ponieważ ich ewentualne wystąpienia we frazach **throws** są ignorowane.

```
package jbPack;

import java.io.*;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

        // ...

    public void fun1()    // (zbędne throws)
        throws ArrayIndexOutOfBoundsException
    {
        // ...
    }

    public void fun2()
        throws IOException
    {
       // ...
    }
}
```

```
class Program2
    extends Program {

    public void fun1()    // dobrze! (zbędne throws)
        throws IndexOutOfBoundsException
    {
        // ...
    }

    public void fun2()    // błąd
        throws Exception
    {
        // ...
    }
}
```

dla dociekliwych

Jeśli w klasie bazowej występują funkcje przeciążone, a jedna z nich zostanie przedefiniowana, to pozostałe **będą** widoczne w klasie pochodnej.

```
class One {

    public void fun()
    {
        // ...
    }

    public void fun(int par)
    {
        // ...
    }
}

class Two
    extends One {

        // funkcja przedefiniowująca
    public void fun()
    {
        // ...
    }

    public void fun(String par)
    {
        fun(12);  // dobrze!

        // ...
    }
}
```

Poszukiwanie składników

Składnikami klasy są ***pola*** i ***funkcje***. Jeśli w pewnym miejscu programu występuje niekwalifikowany identyfikator składnika (np. **getAbs** w odróżnieniu od ***Name*.getAbs** albo ***ref*.getAbs**), to jego deklaracji poszukuje się w najbliższym otoczeniu odwołania, to jest kolejno

1. Wstecz, począwszy od najwęższej instrukcji grupującej, aż do początku funkcji.
2. W obrębie **całej** najwęższej klasy.
3. W obrębie całej bezpośredniej klasy bazowej oraz kolejnych pośrednich klas bazowych, aż do klasy **Object**.
4. W obrębie kolejnych klas zawierających daną klasę.

Jeśli poszukiwanie zakończy się odnalezieniem deklaracji identyfikatora, to identyfikator uznaje się za ***widoczny*** w miejscu jego użycia.

Deklaracje parametrów funkcji traktuje się tak, jakby umieszczono je na początku ciała funkcji. Zabrania się, aby definicja składnika była jednocześnie widoczna ***przez-dziedziczenie*** i ***przez-zawieranie***.

W następującym zestawie klas, w miejscu odwołania się do identyfikatorów od **val1** do **val5** są widoczne deklaracje wymienione w komentarzach.

Ponieważ definicja składnika **both** jest widoczna zarówno przez-dziedziczenie jak i przez-zawieranie, więc odwołanie do składnika jest **nie-jednoznaczne** i musi być rozstrzygnięte za pomocą **kwalifikacji**.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    public Program()
    {
        new Outer().new C().fun();
    }
}

class A {

    protected int val4,
                  val5 = 5,
                  both;

    public void show(int par)
    {
        System.out.println(par);
    }
}
```

```
class Outer {

    class B
        extends A {

        protected int val4 = 4;
    }


    class C
        extends B {

        protected int val2;

        public void fun()
        {
            int val2 = 2;

            {
                int val1;
                val1 = 1;

                show(val1);  // 1
                show(val2);  // 2
                show(val3);  // 3
                show(val4);  // 4
                show(val5);  // 5

                both++;      // błąd

                    // ujednoznacznienie
                super.both++;
                Outer.this.both++;
            }
            int val3 = 0;

        }

        int val3 = 3;
    }

    private int both;
}
```

Identyfikowanie pól

Deklaracje pól są opisami zmiennych. Pola ***obiektowe*** opisują zmienne wchodzące w skład obiektów, a pola ***klasowe*** opisują zmienne występujące **poza** obiektami.

Odwołanie do pola obiektowego poprzedza się nazwą odnośnika do obiektu, a odwołanie do pola klasowego poprzedza się nazwą klasy.

Jeśli zrezygnuje się z takiej kwalifikacji, to nazwę pola klasowego niejawnie poprzedza się nazwą jego klasy, a nazwę pola obiektowego (ale tylko wówczas, gdy odwołanie nie występuje w sposobie!) niejawnie poprzedza się kwalifikatorem **this** (wraz z kropką).

Identyfikowanie przy dziedziczeniu

Jeśli w pewnym miejscu klasy nie jest widoczna deklaracja pola obiektowego ***field***, widocznego w jej klasie **bazowej**, to do tego pola można się odwołać za pomocą nazwy **super.*field***. Odwołanie takie nie może wystąpić w ciele sposobu.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    public Program()
    {
        new C().funM();
        C.funS();
    }
}

class A {

    protected int val = 2;

    public static void show(int val)
    {
        System.out.println(val);
    }
}

class B
    extends A {

}

class C
    extends B {

    public void funM()
    {
        show(val);        // 1
        show(this.val);   // 1
        show(super.val);  // 2
    }
```

```
    public static void funS()
    {
        show(val);        // błąd
        show(this.val);   // błąd (brak this!)
        show(super.val);  // błąd (pole obiektowe)
    }

    private int val = 1;
}
```

Identyfikowanie przy zawieraniu

Jeśli w pewnym miejscu klasy nie jest widoczna deklaracja pola obiektowego ***field***, znajdującego się w klasie ją **zawierającej** o nazwie ***Name***, to do tego pola można się odwołać za pomocą nazwy ***Name*.this.*field***.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    public Program()
    {
        new A().new B().fun();
    }
}

class A {

    private        int valM = 1;
    private static int valS = 2;

    class B {

        public void fun()
        {
                // trzy równoważne fabrykatory
            new C();      // B.this.new C();
            new B.C();    // B.this.new B.C();
            new A.B.C();  // B.this.new A.B.C();
        }

        class C {

            public C()
            {
                System.out.println("Here");
            }
```

```
            public void funM()
            {
                show(valM);          // 1
                show(A.this.valM);   // 1
                show(B.this.valM);   // błąd

                show(valS);          // 2
                show(A.valS);        // 2
            }
        }
    }

    public static void funS()
    {
        show(valS);                  // 2
        show(A.valS);                // 2
        show(B.valS);                // błąd
    }

    public static void show(int val)
    {
        System.out.println(val);
    }
}
```

Identyfikowanie funkcji

Wywołanie **metody** poprzedza się zazwyczaj nazwą odnośnika do obiektu, a wywołanie **sposobu** poprzedza się nazwą klasy, w której jest on **widoczny**.

Jeśli zrezygnuje się z takiej kwalifikacji, to nazwa sposobu zostanie niejawnie poprzedzona nazwą jego klasy, a nazwa metody zostanie niejawnie poprzedzona kwalifikatorem **this**.

Jeśli w pewnym miejscu klasy nie jest widoczna deklaracja funkcji ***fun*** widocznej w jej klasie bazowej, to do tej funkcji można się odwołać za pomocą nazwy **super.*fun***.

W następującej klasie zastosowano m.in. nie-kwalifikowane wywołanie metody **getMin** z wnętrza sposobu **funStatic**. Takie wywołanie powinno być niejawnie poprzedzone kwalifikatorem **this** (z kropką). Ponieważ jednak podczas wykonywania sposobu nie istnieje zmienna **this**, więc wywołanie to jest **błędne**.

```
class Functions {

    public int getMin(int one, int two)
    {
        return one < two ? one : two;
    }

    public static int getMax(int one, int two)
    {
        return one > two ? one : two;
    }
```

```
    public void funMember(int par)
    {
        int min = getMin(10, par),  // this.getMin
            max = getMax(20, par);  // Main.getMax
        // ...
    }

    public static void funStatic(int par)
    {
        int min = getMin(10, par),  // błąd
            max = getMax(20, par);  // Main.getMax
        // ...
    }

    // ...
}
```

dla dociekliwych

Jeśli w programie występuje wywołanie nie-kwalifikowane przez ***ref*** ani przez ***Name***, to przyjmuje się, że chodzi o wywołanie funkcji **widocznej** w miejscu, w którym wystąpiło to wywołanie. W takim wypadku zabrania się, aby były widoczne jednocześnie **dwie** deklaracje: jedna przez **dziedziczenie** i druga przez **zawieranie**.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    public Program()
    {
        new Outer().new Lowest();
    }
}

class Highest {

    public void fun()
    {
        System.out.println("Highest");
    }
}

class Middle
    extends Highest {

}
```

```
class Outer {

    public void fun()
    {
        System.out.println("Outer");
    }

    class Lowest
        extends Middle {

        public Lowest()
        {
            fun();              // błąd
            this.fun();         // Highest
            super.fun();        // Highest
            Outer.fun();        // błąd
            Outer.this.fun();   // Outer
        }
    }
}
```

Inicjowanie klas i obiektów

Inicjowanie ***klasy*** odbywa się tuż przed jej pierwszym ***efektywnym użyciem***. Tworzony jest wówczas unikalny obiekt klasy **Class**, opisujący klasę.

Inicjowanie ***obiektu*** odbywa się tuż po wykonaniu czynności opisanych przez konstruktor klasy bazowej. Składa się ono, w podanej kolejności, z zainicjowania elementów obiektu, wykonania inicjatorów obiektowych oraz wykonania konstruktora klasy.

Inicjatory klasowe

Inicjator klasowy ma postać

```
static {
    // .... instrukcje inicjatora
}
```

Inicjator klasowy umieszcza się w takim miejscu, gdzie mogłaby wystąpić deklaracja pola klasy.

Instrukcje inicjatora klasowego są wykonywane **jednokrotnie**, bezpośrednio po załadowaniu klasy. Jeśli klasa zawiera więcej niż jeden inicjator klasowy, to wykonuje się je **wszystkie**, w kolejności ich wystąpienia w klasie.

W następującej aplikacji klasa **Program** jest efektywnie użyta, ponieważ jest wywoływana jej funkcja **main**. Natomiast klasa **Outer** nie jest efektywnie użyta i dlatego wśród liczb

wyprowadzonych w kolejności

```
3 5 2 1 4
```

nie pojawia się liczba **6**.

```
package jbPack;

public
class Program {

    private int val = getVal(1);

    public static void main(String[] args)
    {
        int val = getVal(2);

            // opracowanie poniższej deklaracji
            // nie jest efektywnym użyciem klasy
        Outer outer;

        new Program();
    }

    static {    // inicjator klasowy

        int val = getVal(3);
    }

    public Program()
    {
        int val = getVal(4);
    }

    static {    // inicjator klasowy
        int val = getVal(5);
    }

    public static int getVal(int val)
    {
        System.out.println(val);
        return val;
    }
}

class Outer {

    static {

        int val = Program.getVal(6);
    }
}
```

Inicjatory obiektowe

Inicjator obiektowy ma postać

```
{
    // ....
}
```

Umieszcza się go w takim samym miejscu, w którym może być umieszczony inicjator **klasy**.

Inicjator obiektowy jest wykonywany bezpośrednio po wykonaniu czynności opisanych przez konstruktor klasy bazowej i po wstępnym zainicjowaniu elementów obiektu. Jeśli klasa zawiera więcej niż jeden inicjator obiektowy, to wykonuje się je **wszystkie**, w kolejności ich wystąpienia w klasie.

Następująca aplikacja wyprowadza liczby w kolejności

```
5 1 3 4 2
```

```
public
class Program
    extends Outer {

    public static void main(String[] args)
    {
        new Program();
    }

    private int val = getVal(1);

    public static int getVal(int val)
    {
        System.out.println(val);
        return val;
    }

    public Program()
    {
        val = getVal(2);
    }

    {       // inicjator obiektowy
        val = getVal(3);
    }

    {       // inicjator obiektowy
        val = getVal(4);
    }
}
```

```
class Outer {

    public Outer()
    {
        int val = Program.getVal(5);
    }
}
```

Interfejsy

Klasa może dziedziczyć co najwyżej **jedną** klasę, ale może implementować dowolnie wiele ***interfejsów***.

Interfejs można uznać za statyczną ***klasę abstrakcyjną***, która zawiera co najwyżej **deklaracje** funkcji i **definicje** zmiennych ustalonych. Funkcje interfejsu są domyślnie **publiczne** i **abstrakcyjne**, a jego zmienne są domyślnie **publiczne**, **statyczne** i **ustalone**.

W szczególności następująca definicja

```
interface Drawable {

    double Pi = 3.141592;

    void draw();
}
```

jest równoważna definicji

```
static abstract
interface Drawable {

    public static final double Pi = 3.141592;

    public abstract void draw();
}
```

Implementowanie interfejsów

Fraza wyrażająca implementowanie interfejsów ma postać

```
   implements Name, Name, ... , Name
```

w której każde ***Name*** jest nazwą interfejsu.

```
class Derived
    extends Primary                  // dziedziczenie
    implements Drawable, Movable {   // implementowanie

    // ...
}
```

W klasie implementującej interfejs dostarcza się zazwyczaj definicje wszystkich jego funkcji. Jeśli się tego nie uczyni, to klasa staje się **abstrakcyjna**, a więc w fabrykatorze nie może wystąpić odwołanie do jej konstruktora.

Klasa abstrakcyjna może być klasą bazową innej klasy. Umożliwia to jej klasie pochodnej dostarczenie brakujących definicji metod abstrakcyjnych odziedziczonych po klasie bazowej.

Dziedziczenie interfejsów

Interfejs może **dziedziczyć** dowolnie wiele interfejsów.

Fraza wyrażająca dziedziczenie interfejsów ma postać

```
   extends Name, Name, ... , Name
```

w której każde *Name* jest nazwą interfejsu.

```
interface Drawable {

    double Pi = 3.14;

    void draw();
}

interface Movable {

    void moveTo(int x, int y);
    void moveBack();
}

interface Paintable_And_Movable
    extends Drawable, Movable {

    // ...
}
```

Fabrykatory

Obiektem jest **zmienna** składająca się z zestawu ***elementów***.

Każdy element jest zmienną typu **numerycznego**, **orzecznikowego** albo **odnośnikowego**. W odróżnieniu od tablic, poszczególne elementy obiektu nie muszą być takiego samego typu. Elementami obiektów nie mogą być tablice ani obiekty.

Elementy obiektów są opisane przez ***pola obiektowe*** zadeklarowane w ***klasie*** obiektu i są tworzone dopiero po sfabrykowaniu obiektu.

W szczególności deklaracja

```
Point origin;
```

nie tworzy obiektu klasy **Point**, a jedynie odnośnik do dowolnego obiektu tej klasy.

Dopiero po opracowaniu fabrykatora, takiego jak na przykład

```
new Point(10, 20)
```

nastąpi utworzenie obiektu składającego się z **2** elementów typu **int** opisanych przez pola **obiektowe** klasy.

```
class Point {

    private int x, y;          // pola obiektowe
    public static int count;   // pole klasowe

    public Point(int x, int y)
    {
        // ...
    }

    // ...
}
```

Składnia fabrykatorów

Fabrykator obiektu klasy ***zewnętrznej***, ***zanurzonej*** oraz ***lokalnej Name*** ma postać

```
new Name(arg, arg, ... , arg)
```

w której ***Name*(*arg*, *arg*, *... , arg*)** jest wywołaniem konstruktora tej klasy.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        Outer outer = new Outer(10, 20);

        // ...
    }
}

class Outer {

    private int x, y;

    public Outer(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

Fabrykator obiektu klasy ***wewnętrznej Name*** ma postać

```
ref.new Name(arg, arg, ... , arg)
```

albo

```
new Name(arg, arg, ... , arg)
```

w której ***Name*(*arg*, *arg*, *... , arg*)** jest wywołaniem konstruktora tej klasy, a ***ref*** jest odnośnikiem do obiektu klasy zawierającej jej definicję.

Jeśli nie użyto kwalifikatora ***ref***, a ***Name*** **jest** nazwą klasy wewnętrznej, to jako kwalifikator domniemywa się **this**.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        // ...
    }
```

```
    public Program()
    {
        Inner inner1 = Program.this.new Inner(10, 20),
              inner2 = this.new Inner(10, 20),
              inner3 = new Inner(10, 20);
    }

    class Inner {

        private int x, y;

        public Inner(int x, int y)
        {
            this.x = x;
            this.y = y;
        }
    }
}
```

Fabrykator obiektu klasy ***anonimowej*** ma postać

```
   ref.new Name(arg, arg, ... , arg) {
       // ...
   }
```

albo

```
   new Name(arg, arg, ... , arg) {
       // ...
   }
```

w której ***Name*(*arg*, *arg*, … , *arg*)** jest wywołaniem konstruktora **klasy bazowej** klasy anonimowej, a ***ref*** jest odnośnikiem do obiektu klasy zawierającej definicję klasy wewnętrznej ***Name***.

Jeśli nie użyto kwalifikatora ***ref***, a ***Name*** **jest** nazwą klasy wewnętrznej, to jako kwalifikator domniemywa się **this**.

Przyjmuje się, że zdefiniowana w taki sposób klasa anonimowa jest pochodną klasy ***Name*** oraz że zainicjowanie pod-obiektu jej klasy bazowej zostanie powierzone konstruktorowi użytemu w fabrykatorze.

Ciało klasy anonimowej **nie może** zawierać definicji konstruktorów.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        // ...
    }
```

```
    public Program()
    {
        Inner inner =
            new Inner(10, 20) {

                private int z = 13;

                public int getSum()
                {
                    return x + y + z;
                }
            };
    }

    class Inner {

        protected int x, y;

        public Inner(int x, int y)
        {
            this.x = x;
            this.y = y;
        }
    }
}
```

Fabrykator obiektu klasy ***anonimowej*** implementującej ***interfejs*** może mieć **tylko** postać

```
new Name() {
    // ...
}
```

w której *Name* jest nazwą interfejsu.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Printable() {

            public void print(String par)
            {
                System.out.println(par);
            }
        }.print("Hello");
    }

    interface Printable {

        public void print(String par);
    }
}
```

Anatomia fabrykacji

Wykonanie operacji fabrykowania obiektu składa się z wykonania następujących czynności

1. Utworzenia ***zmiennej obiektowej*** składającej się z elementów opisanych przez pola obiektowe klasy.
2. Utworzenia odnośnika **this** identyfikującego zmienną obiektową.
3. Zainicjowania pól obiektowych (domyślnie wartościami **0**, **false**, **null**).
4. Wykonania inicjatorów obiektów.
5. Wywołania **konstruktora** klasy w celu przekształcenia zmiennej obiektowej w **obiekt**.
6. Dostarczenia **odniesienia** do właśnie utworzonego obiektu.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        Point p = new Point(10, 20);
        System.out.println(
            "p(" + p.x + ',' + p.y + ')'  // p(10,20)
        );
    }
}

class Point {

    int x, y;

    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

Konstruowanie obiektów

Podczas opracowywania fabrykatora jest wywoływany konstruktor najlepiej pasujący do użytej listy argumentów. Jeśli jego pierwszą instrukcją jest

```
this(arg, arg, ... , arg);
```

to następuje wywołanie innego konstruktora **tej samej** klasy, a jeśli jest nią

```
super(arg, arg, ... , arg);
```

to następuje wywołanie konstruktora klasy **bazowej** danej klasy.

Każda z wymienionych instrukcji może być tylko **pierwszą** instrukcją dowolnego konstruktora. Jeśli nie użyto żadnej z nich, to jako pierwszą instrukcję konstruktora domniemywa się instrukcję

```
super();
```

Zabrania się użycia takiego zestawu wywołań konstruktorów, które mogłoby spowodować **zapętlenie** wykonania programu.

```
class Point {

    private int x, y;

    public Point()
    {
        this(0);
    }

    public Point(int x)
    {
        this(x, 0);
    }

    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}

class Point3D
    extends Point {

    private int z;

    public Point3D(int x, int y, int z)
    {
        super(x, y);
        this.z = z;
    }
}
```

Konwersje

Konwersją ***odnośnikową*** jest przekształcenie ***odnośnika-w-odnośnik***. Konwersja odnośnikowa jest poprawna tylko wówczas, gdy jest poprawna **statycznie** i **dynamicznie**.

Poprawność statyczna

Poprawna ***statycznie*** jest

1. Konwersja wzdłuż takiej linii dziedziczenia, która od klasy odnośnika poddawanego konwersji do klasy odnośnika docelowego wiedzie **tylko w górę** albo **tylko w dół**.
2. Konwersja z dowolnego typu klasowego albo interfejsowego na dowolny typ interfejsowy.
3. Niejawna konwersja z typu klasowego na jego bazowy typ klasowy.
4. Niejawna konwersja z typu klasowego na implementowany przezeń typ interfejsowy.
5. Niejawna konwersja z typu interfejsowego na jego bazowy typ interfejsowy.

Następująca aplikacja ilustruje sposób rozstrzygania o poprawności konwersji statycznych. Liczby podane w komentarzach nawiązują do przytoczonych powyżej konwersji.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    private Woman woman = new Woman();
    private Animal animal = new Animal();
    private Object object;
    private Face face;
    private Snout snout;
    private Look look;
    private Man man;
```

```
    public Program()
    {
        object = (Object)woman;       // ad. 1
        woman = (Woman)object;        // ad. 1
        man = woman;                  //       błąd statyczny
        man = (Man)(Object) woman;    //       błąd dynamiczny
        snout = (Snout)woman;         //       błąd dynamiczny
        snout = (Snout)face;          // ad. 2
        object = woman;               // ad. 3
        snout = woman;                //       błąd statyczny
        face = woman;                 // ad. 4
        snout = face;                 //       błąd statyczny
        look = snout;                 // ad. 5
    }
}

// ========================================================

interface Face {
    // ...
}

class Person
    implements Face {

    // ...
}

class Woman
    extends Person {

    // ...
}

class Man
    extends Person {

}

interface Look {

}

interface Snout
    extends Look {
    // ...
}

class Animal
    implements Snout {

    // ...
}
```

Poprawność dynamiczna

Poprawna ***dynamicznie*** jest konwersja, w której

1. Odnośnik docelowy jest typu **klasowego**, a jego klasa znajduje się w hierarchii klas **powyżej** i **na linii** łączącej klasę **odniesienia** przypisanego temu odnośnikowi z klasą **Object**.
2. Odnośnik docelowy jest typu **interfejsowego**, a klasa przypisanego mu **odniesienia** implementuje ten interfejs.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    private Woman woman = new Woman();
    private Person person;
    private Face face;
    private Snout snout;
    private Man man;

    public Program()
    {
        person = woman;              // ad. 1
        face = woman;                // ad. 2
        man = (Man)(Person)woman;    //       błąd dynamiczny
        snout = (Snout)woman;        //       błąd dynamiczny
    }
}

//  ========================================================

interface Face {
    // ...
}

class Person
    implements Face {

    // ...
}

class Woman
    extends Person {

    // ...
}

class Man
    extends Person {

}
```

```
interface Look {

}

interface Snout
    extends Look {
    // ...
}

class Animal
    implements Snout {

    // ...
}
```

Przypisania odnośników

Podczas wykonania programu

1. Odnośnikowi **klasowemu** typu ***Name*** można przypisać tylko odniesienie do obiektu klasy ***Name*** albo odniesienie do obiektu klasy pochodnej od ***Name***.
2. Odnośnikowi **interfejsowemu** klasy ***Name*** można przypisać tylko odniesienie do obiektu klasy implementującej interfejs ***Name***.

Wykonanie takiego przypisania nie wymaga jawnego zastosowania konwersji.

Przykład

```
class Person {

    // ...
}

class Woman
    extends Person {

    // ...
}

class Program {

    public Program()
    {
        Person person;
        Woman woman =
            new Woman("Mary Taylor", 30, true);
        person = woman;          // person = (Person)woman;
        woman  = (Woman)person;  // dobrze!
        woman  = person;         // błąd statyczny
        // ...
    }
}
```

Polimorfizm

Odnośnikowi klasowemu można przypisać odniesienie nie tylko do obiektu jego klasy, ale również odniesienie do obiektu jego klasy **pochodnej**, a odnośnikowi interfejsowemu można przypisać odniesienie do obiektu klasy **implementującej** jego interfejs. Dlatego podczas opracowania wywołania

```
ref.fun(arg, arg, ... , arg)
```

typ odniesienia przypisanego odnośnikowi ***ref*** może być **inny** niż typ odnośnika.

Istota polimorfizmu

Niezależnie od tego, czy wystąpi taka rozbieżność czy nie, zawsze wywołuje się funkcję ***fun*** widoczną w klasie **odniesienia** przypisanego odnośnikowi ***ref***, a nie funkcję ***fun*** widoczną w klasie (albo w interfejsie) tego odnośnika. Dlatego to samo wywołanie może dotyczyć **różnych** funkcji, a więc jest ***polimorficzne*** (wielopostaciowe).

Wywołanie jest ***nie-polimorficzne*** tylko wówczas, gdy odnośnik ***ref*** jest typu klasowego, a w jego klasie metoda ***fun*** jest **prywatna** albo **finalna** albo gdy ***ref*** jest słowem kluczowym **super**.

W następującej aplikacji, w miejscu wystąpienia wywołania **person.getInfo**, odnośnikowi **person** jest przypisane odniesienie do obiektu klasy **Woman**. Dlatego, mimo iż odnośnik ten jest typu **Person**, nastąpi wywołanie funkcji **Woman.getInfo**.

W miejscu wystąpienia wywołania **who()** domyślnym ***ref*** jest **this**. Dlatego, jeśli argument funkcji **showInfo** identyfikuje obiekt klasy **Woman**, to zostanie wywołana funkcja **Woman.who**, a jeśli identyfikuje obiekt klasy **Person**, to zostanie wywołana funkcja **Person.who**.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }
```

Wyniki

```
Woman:  Iza 17 false
Person: Ewa 40
```

```
    public Program()
    {
            // Woman:  Iza 17 false
        showInfo(
            new Woman("Iza", 17, false)
        );

            // Person: Ewa 40
        showInfo(
            new Person("Ewa", 40)
        );
    }

    public void showInfo(Person person)
    {
        System.out.println(
            person.getInfo()
        );
    }
}

class Person {

    protected String name;
    protected int age;

    public Person(String name, int age)
    {
        this.name = name;
        this.age = age;
    }

    public String getInfo()
    {
        return who() + name + ' ' + age;
    }

    public String who()
    {
        return "Person: ";
    }
}

class Woman
    extends Person {

    private boolean isMarried;

    public Woman(String name, int age, boolean isMarried)
    {
        super(name, age);
        this.isMarried = isMarried;
    }
```

```
    public String getInfo()
    {
            // wywołanie funkcji Person.getInfo
        return super.getInfo() +
               ' ' + isMarried;
    }

    public String who()
    {
        return "Woman:  ";
    }
}
```

Operator instanceof

Sprawdzianem biegłego posługiwania się polimorfizmem jest ograniczenie do minimum posługiwania się operatorem **instanceof**.

Ilustruje to rozwiązanie następującego problemu

> Dysponując klasami opisującymi ***obcych*** (**Alien**) i ***ludzi*** (**Human**) oraz przy założeniu, że w klasę **Human** nie można ingerować, bo na przykład dostarczono ją w pliku **Human.class**, utworzyć kolekcję **obcych** oraz **ludzi-*matek*** (**Mother**) i **ludzi-*ojców*** (**Father**) oraz umożliwić rozpoznawanie i zliczanie jej elementów.

Z operatorem instanceof

Zastosowano dziedziczenie klas i wielokrotnie posłużono się operatorem **instanceof**. Zliczanie elementów zrealizowano poza kolekcją.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    public Program()
    {
        Collector creatures = new Collector();

        creatures.add(new Father("John", 40));
        creatures.add(new Alien("Zork"));
        creatures.add(new Mother("Mary", 24));
        creatures.add(new Mother("Tess", 30));

        int aliens  = 0,
```

Wyniki

```
Father John
Alien  Zork
Mother Mary
Mother Tess
============
aliens: 1
humans: 3
   mothers: 2
   fathers: 1
```

```
            humans  = 0,
            mothers = 0,
            fathers = 0;

        for (int i = 0; i < creatures.size() ; i++) {
            Object creature = creatures.get(i);
            if (creature instanceof Alien) {
                aliens++;
                System.out.println(
                        // wywołanie nie-polimorficzne
                    ((Alien)creature).who()
                );
            } else if (creature instanceof Parent) {
                humans++;
                System.out.println(
                        // wywołanie polimorficzne
                    ((Parent)creature).who()
                );
                if (creature instanceof Mother)
                    mothers++;
                else
                    fathers++;
            } else {
                System.out.println(
                    "Not an Alien nor Human"
                );
                System.exit(0);
            }
        }

            // podsumowanie
        System.out.println(
            "=============\n" +
            "aliens: " + aliens + '\n' +
            "humans: " + humans + '\n' +
            "   mothers: " + mothers + '\n' +
            "   fathers: " + fathers + '\n'
        );
    }
}

    // obcy
class Alien {

    private String name;

    public Alien(String name)
    {
        this.name = name;
    }

    public String who()
    {
        return "Alien  " + name;
    }
}
```

```
    // człowiek
class Human {

    protected String name;

    public Human(String name)
    {
        this.name = name;
    }
}

    // rodzic
abstract
class Parent
    extends Human {

    public Parent(String name)
    {
        super(name);
    }

    public abstract String who();
}

    // matka
class Mother
    extends Parent {

    private int age;

    public Mother(String name, int age)
    {
        super(name);

        this.age = age;
    }

    public String who()
    {
        return "Mother " + super.name;
    }
}

    // ojciec
class Father
    extends Parent {

    private int age;

    public Father(String name, int age)
    {
        super(name);

        this.age = age;
    }
```

```
    public String who()
    {
        return "Father " + super.name;
    }
}

    // klasa kolekcyjna
class Collector {

    private Object[] pObject = new Object [10];
    private int count = 0;

    public void add(Object pObject)
    {
        if (count < 10)
            this.pObject[count++] = pObject;
        else {
            System.out.println("Too many objects");
            System.exit(0);
        }
    }

    public Object get(int i)
    {
        return pObject[i];
    }

    public int size()
    {
        return count;
    }
}
```

Bez operatora instanceof

Zastosowano dziedziczenie klas, implementowanie interfejsów oraz polimorfizm. Zliczanie elementów przeniesiono do **wyspecjalizowanej** kolekcji. Pozbyto się operatora **instanceof**.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    public Program()
    {
        Collector creatures = new Collector();
```

Wyniki

```
Father John
Alien  Zork
Mother Mary
Mother Tess
============
aliens: 1
humans: 3
   mothers: 2
   fathers: 1
```

```
        creatures.add(new Father("John", 40));
        creatures.add(new Alien("Zork"));
        creatures.add(new Mother("Mary", 24));
        creatures.add(new Mother("Tess", 30));

        creatures.showAll();


/* ================= równoważne creatures.showAll();

        creatures.clearCounts();

        for (int i = 0; i < creatures.size() ; i++) {
            Creature creature = creatures.get(i);
            System.out.println(
                    // wywołanie polimorficzne
                creature.who()
            );
        }
/**/

        int aliens  = Alien.count,
            mothers = Mother.count,
            fathers = Father.count,
            humans  = mothers + fathers;

            // podsumowanie
        System.out.println(
            "=============\n" +
            "aliens: " + aliens + '\n' +
            "humans: " + humans + '\n' +
            "   mothers: " + mothers + '\n' +
            "   fathers: " + fathers + '\n'
        );
    }
}

interface Creature {

    String who();
    void clearCount();
}

    // obcy
class Alien
    implements Creature {

    private String name;
    public static int count;

    public Alien(String name)
    {
        this.name = name;
    }
```

```
    public String who()
    {
        count++;
        return "Alien  " + name;
    }

    public void clearCount()
    {
        count = 0;
    }
}

    // człowiek
class Human {

    protected String name;

    public Human(String name)
    {
        this.name = name;
    }
}

    // rodzic
abstract
class Parent
    extends Human
    implements Creature {

    public Parent(String name)
    {
        super(name);
    }

    public abstract String who();
}

    // matka
class Mother
    extends Parent {

    private int age;
    public static int count;

    public Mother(String name, int age)
    {
        super(name);

        this.age = age;
    }

    public String who()
    {
        count++;
        return "Mother " + super.name;
    }
```

```
    public void clearCount()
    {
        count = 0;
    }
}

    // ojciec
class Father
    extends Parent {

    private int age;
    public static int count;

    public Father(String name, int age)
    {
        super(name);

        this.age = age;
    }

    public String who()
    {
        count++;
        return "Father " + super.name;
    }

    public void clearCount()
    {
        count = 0;
    }
}

    // klasa kolekcyjna
class Collector {

    private Creature[] pCreature = new Creature [10];
    private int count = 0;

    public void add(Creature pCreature)
    {
        if (count < 10)
            this.pCreature[count++] = pCreature;
        else {
            System.out.println("Too many creatures");
            System.exit(0);
        }
    }

    public Creature get(int i)
    {
        return pCreature[i];
    }

    public int size()
    {
        return count;
    }
```

```
    public void clearCounts()
    {
        for (int i = 0; i < count ; i++)
            pCreature[i].clearCount();
    }

    public void showAll()
    {
        clearCounts();

        for (int i = 0; i < count ; i++) {
            System.out.println(
                    // wywołanie polimorficzne
                pCreature[i].who()
            );
        }
    }
}
```

Przetwarzanie list

Listą jest zestaw obiektów powiązanych ***łącznikami***. Łączniki implementuje się zazwyczaj za pomocą **odnośników**.

Następująca aplikacja ilustruje istotę powiązań obiektów.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    class Item {

        public Item pNext;
        public int value;

        public Item(Item pNext, int value)
        {
            this.pNext = pNext;
            this.value = value;
        }
    }

    public Program()
    {
            // utworzenie listy
        Item fifth  = new Item(null,   50),
             fourth = new Item(fifth,  40),
             third  = new Item(fourth, 30),
             second = new Item(third,  20),
             first  = new Item(second, 10),
             pHead  = first;

            // przejrzenie listy
        Item pItem = pHead;
        while (pItem != null) {
            System.out.println(
               pItem.value    // 10 20 30 40 50
            );
```

```
            pItem = pItem.pNext;
        }
    }
}
```

Listy jedno-kierunkowe

Następująca aplikacja tworzy listę jedno-kierunkową, a następnie **wstawia** do niej i **usuwa** z niej jeden element.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    class Item {

        public Item pNext;
        public int value;

        public Item(int value)
        {
            this.value = value;
        }
    }

    public Program()
    {
            // utworzenie listy
        Item pHead,
             first  = new Item(10),
             second = new Item(20);
        pHead = first;
        pHead.pNext = second;

            // pokazanie listy
        showList(pHead);  // 10 20

            // nastawienie pItem na element
        Item pItem = first;

            // utworzenie nowego elementu
        Item pNew  = new Item(99);

            // wstawienie nowego elementu
            // za pokazywanym przez pItem
        Item pNext = pItem.pNext;
```

```
        if (pNext == null)
            pItem.pNext = pNew;
        else {
            pItem.pNext = pNew;
            pNew.pNext  = pNext;
        }

        showList(first);  // 10 99 20

            // nastawienie pItem na element
        pItem = first;  // zbyteczne

            // usunięcie następnego elementu
            // po pokazywanym przez pItem
        pNext = pItem.pNext;
        if (pNext != null) {
            if (pNext.pNext == null)
                pItem.pNext = null;
            else
                pItem.pNext = pNext.pNext;
        }

        showList(first);  // 10 20
    }

        // przeglądanie listy
    public void showList(Item pHead)
    {
        System.out.println();

        while (pHead != null) {
            System.out.println(
               pHead.value
            );
            pHead = pHead.pNext;
        }
    }
}
```

Listy dwu-kierunkowe

Następująca aplikacja tworzy listę dwu-kierunkową, a następnie **wstawia** do niej i **usuwa** z niej jeden element.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }
```

```
class Item {

    public Item pNext,
                pBack;
    public int value;

    public Item(int value)
    {
        this.value = value;
    }
}

public Program()
{
        // utworzenie listy
    Item pHead,
         first  = new Item(10),
         second = new Item(20);
    pHead = first;
    pHead.pNext = second;
    pHead.pNext.pBack = first;

        // pokazanie listy
    showList(pHead);  // 10 20

        // nastawienie pItem na element
    Item pItem = first;

        // utworzenie nowego elementu
    Item pNew  = new Item(99);

        // wstawienie nowego elementu
        // za pokazywanym przez pItem
    Item pNext = pItem.pNext;
    if (pNext == null) {
        pItem.pNext = pNew;
        pNew.pBack  = pItem;
    } else {
        pItem.pNext = pNew;
        pNew.pBack  = pItem;
        pNew.pNext  = pNext;
        pNext.pBack = pNew;
    }

    showList(first);  // 10 99 20

        // nastawienie pItem na element
    pItem = first;  // zbyteczne

        // usunięcie następnego elementu
        // po pokazywanym przez pItem
    pNext = pItem.pNext;
    if (pNext != null) {
        if (pNext.pNext == null)
            pItem.pNext = null;
```

```
            else {
                pItem.pNext = pNext.pNext;
                pNext.pNext.pBack = pItem;
            }
        }

        showList(first);  // 10 20
    }

        // przeglądanie listy
    public void showList(Item pHead)
    {
        System.out.println();

        while (pHead != null) {
            System.out.println(
               pHead.value
            );
            pHead = pHead.pNext;
        }
    }
}
```

Listy uporządkowane

Następująca aplikacja tworzy listę składającą się z obiektów reprezentujących imiona dostarczone w dialogu. Nowe elementy są tak związywane z listą, aby w każdej chwili lista była **posortowana**.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    private String[] names = { "Iza", "Ewa", "Jan", "Iza" };

    public Program()
    {
            // utworzenie listy
        List list = new List();

            // dokładanie elementów
        for(int i = 0; i < names.length ; i++)
            list.add(names[i]);

            // pokazanie listy
        list.show();  // Ewa Iza Iza Jan
    }
```

```
    class Item {

        public Item pNext;
        public String pName;

        public Item(String name)
        {
            pName = name;
        }

        public Item(Item pNext, String name)
        {
            this.pNext = pNext;
            pName = name;
        }
    }

    class List {

        private Item pHead;

        public void add(String name)
        {
            Item pName = new Item(name);

            if (pHead == null)
                pHead = pName;
            else if (name.compareTo(pHead.pName) < 0) {
                    pName.pNext = pHead;
                    pHead = pName;
            } else  {
                Item pItem = pHead;
                while (pItem.pNext != null &&
                        name.compareTo(pItem.pNext.pName) >= 0)
                    pItem = pItem.pNext;

                if (pItem.pNext == null) {
                    pItem.pNext = pName;
                } else {
                    pName.pNext = pItem.pNext;
                    pItem.pNext = pName;
                }
            }
        }

        public void show()
        {
            System.out.println();

            Item pItem = pHead;
            while (pItem != null) {
                System.out.println(pItem.pName);
                pItem = pItem.pNext;
            }
        }
    }
}
```

Kolekcjonowanie

Kolekcją jest zestaw odnośników do obiektów. Najczęściej używane kolekcje są oparte na klasach **Vector**, **Stack** i **Hashtable**.

Wszystkie odnośniki są typu **Object**, ale każdy z nich może identyfikować obiekt dowolnej klasy. Ponadto, ponieważ istnieje niejawna konwersja z dowolnego typu odnośnikowego do typu **Object**, więc jeśli parametr funkcji kolekcyjnej (np. ***add***) jest typu **Object**, to można z nim skojarzyć argument dowolnego typu tablicowego albo obiektowego (np. typu **String**, albo **int[]**).

```
int[] numbers = { 10, 17 };
String name = "Iza";
Vector vector = new Vector();
vector.add(numbers);
vector.add(name);
for (int i = 0; i < vector.size() ; i++) {
    Object object = vector.get(i);
    if (object instanceof int[]) {
        int[] num = (int [])object;
        System.out.println(
            num[1]  // 17
        );
    } else if (object instanceof String) {
        String str = (String)object;
        System.out.println(
            str     // Iza
        );
    }
}
```

Kolekcja wektorowa

Kolekcja **wektorowa** jest oparta na klasie kolekcyjnej **Vector**. Z każdym odnośnikiem do obiektu znajdującym się w kolekcji jest związany ***indeks*** (od **0** w górę). Odnośniki do obiektów można dokładać w dowolnym miejscu kolekcji.

Wszystkie metody klasy **Vector** są ***synchronizowane***. Dzięki temu **nie może** dojść do sytuacji, kiedy dwa wątki **jednocześnie** modyfikują zawartość kolekcji.

klasa Vector

Vector()
Konstruuje obiekt klasy **Vector** reprezentujący pustą kolekcję tablicową o potencjalnie nieograniczonej pojemności.
np. Vector vector = new Vector();

boolean **add**(Object obj)
Dołącza na końcu kolekcji odnośnik do obiektu ***obj***. Zawsze dostarcza **true**.
np. vector.add(new Person("Mary Taylor", 25));

Object **get**(int index)
Dostarcza odniesienie zawarte w odnośniku, który w kolekcji znajduje się na pozycji ***index***. Jeśli kolekcja nie zawiera odnośnika o podanym indeksie, to wysyła wyjątek klasy **ArrayIndexOutOfBoundsException**.
np. Person man = (Person)vector.get(3);

Object **set**(int index, Object obj)
Zastępuje odnośnik do obiektu, który znajduje się w kolekcji na pozycji ***index***, odnośnikiem do obiektu ***obj*** oraz dostarcza odniesienie przypisane zastępowanemu odnośnikowi. Jeśli na podanej pozycji kolekcja nie zawiera odnośnika do obiektu, to wysyła wyjątek klasy **ArrayIndexOutOfBoundsException**.
np. vector.set(3, "Ewa");

boolean **contains**(Object obj)
Liniowo i za pomocą funkcji **equals**, porównuje obiekty identyfikowane przez odnośniki kolekcji z obiektem identyfikowanym przez ***obj***, dostarczając orzeczenie o wartości: "*kolekcja zawiera odnośnik do obiektu* ***obj***".
np. boolean contains = vector.contains("Jan");

boolean **remove**(Object obj)
Usuwa z kolekcji odnośnik do obiektu ***obj***. Dostarcza **true**, jeśli kolekcja zawierała taki odnośnik.
np. boolean wasThere = vector.remove("Hello World");

int **size**()
Dostarcza liczbę odnośników do obiektów kolekcji.
np. int length = vector.size();

Object **clone**()
Dostarcza odniesienie do klonu kolekcji. Klonowanie jest **płytkie**, to jest obejmuje tylko odnośniki.
np. Vector vector2 = (Vector)vector.clone();

Object[] **toArray**()
Tworzy tablicę odnośników zainicjowaną odniesieniami do obiektów kolekcji. Dostarcza odniesienie do tej tablicy.
np. Object[] array = vector.toArray();

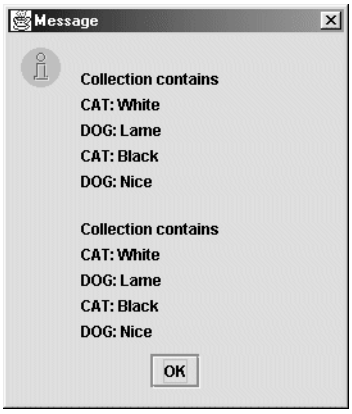
```
void clear()
Usuwa z kolekcji wszystkie znajdujące się w niej odnośniki do
obiektów.
np. vector.clear();
```

dla dociekliwych

Funkcja **contains(*obj*)** dostarcza poprawnej odpowiedzi tylko wówczas, gdy w klasie obiektu ***obj*** zdefiniowano własną funkcję **equals** (a nie odziedziczono jej na przykład po klasie **Object**, kiedy to porównanie dotyczyłoby tylko odnośników do obiektów).

Następująca aplikacja, z wynikami na ekranie ***Kolekcja wektorowa***, tworzy kolekcję obiektów reprezentujących "zwierzaki", a następnie informuje o nich.

Ekran
Kolekcja wektorowa




```
package jbPack;

import java.util.*;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    public Program()
    {
        Cat whiteCat = new Cat("White"),
            blackCat = new Cat("Black");

        Dog lameDog = new Dog("Lame"),
            niceDog = new Dog("Nice");

        Vector pets = new Vector();

        pets.add(whiteCat);
        pets.add(lameDog);
```

```
        pets.add(blackCat);
        pets.add(niceDog);

        int count = pets.size();

                            // rozwiązanie niepolimorficzne

        System.out.println("\nCollection contains");
        for (int i = 0; i < count ; i++) {
            Object object = pets.get(i);
            Pet pet = (Pet)object;

            if (pet instanceof Dog) {
                Dog dog = (Dog)pet;
                System.out.println(
                    "DOG: " + dog.getName()
                );
            }

            if (pet instanceof Cat) {
                Cat cat = (Cat)pet;
                System.out.println(
                    "CAT: " + cat.getName()
                );
            }
        }

                            // rozwiązanie polimorficzne

        System.out.println("\nCollection contains");
        for (int i = 0; i < count ; i++) {
            Object object = pets.get(i);
            Pet pet = (Pet)object;

            System.out.println(
                pet.getKind() + ": " + pet.getName()
            );
        }
    }
}

abstract
class Pet {

    private String name;

    public Pet(String name)
    {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }
```

```
    public abstract String getKind();
}

class Dog
    extends Pet {

    public Dog(String name)
    {
        super(name);
    }

    public String getKind()
    {
        return "DOG";
    }
}

class Cat
    extends Pet {

    public Cat(String name)
    {
        super(name);
    }

    public String getKind()
    {
        return "CAT";
    }
}
```

Kolekcja stosowa

Kolekcja **stosowa** jest oparta na klasie kolekcyjnej **Stack**. Obiekty można dokładać ***na-końcu*** kolekcji, ale można je z niej wyjmować tylko ***od-końca***.

Klasa **Stack** jest pochodną klasy **Vector**. Wszystkie metody klasy **Stack** są **synchronizowane**.

klasa Stack

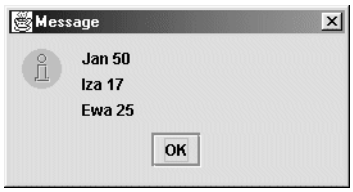
Stack()
Konstruuje obiekt klasy **Stack** reprezentujący pustą kolekcję stosową.
np. Stack stack = new Stack();

Object **push**(Object obj)
Umieszcza na końcu kolekcji odnośnik do obiektu ***obj***. Dostarcza odniesienie do ***obj***.
np. stack.push("Ewa");

```
Object pop()
Usuwa z kolekcji ostatni odnośnik do obiektu. Dostarcza odniesienie
zawarte w tym odnośniku. Jeśli kolekcja była pusta, wysyła wyjatek
klasy EmptyStackException.
np. String name = (String)stack.pop();

boolean empty()
Dostarcza orzeczenie o wartości: "kolekcja jest pusta".
np. boolean isEmpty = stack.empty();
```

Następująca aplikacja, z wynikami na ekranie ***Kolekcja stosowa***, tworzy kolekcję obiektów reprezentujących osoby, a następnie informuje o nich.

Ekran
Kolekcja stosowa




```
package jbPack;

import java.util.*;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    public Program()
    {
        Stack stack = new Stack();

        stack.push(new Person("Ewa", 25));
        stack.push(new Person("Iza", 17));
        stack.push(new Person("Jan", 50));

        while (!stack.empty()) {
            Person name = (Person)stack.pop();
            System.out.println(name.getInfo());
        }
    }
}

class Person {

    private String name;
    private int age;
```

```
    public Person(String name, int age)
    {
        this.name = name;
        this.age = age;
    }

    public String getInfo()
    {
        return name + ' ' + age;
    }
}
```

Kolekcja mieszająca

Kolekcja **mieszająca** jest oparta na klasie kolekcyjnej **Hashtable**. Wszystkie metody kolekcji są **synchronizowane**.

Każdy obiekt znajdujący się w kolekcji jest opatrzony ***kluczem***. Obranemu kluczowi może odpowiadać co najwyżej **jeden** obiekt kolekcji. Nic nie stoi na przeszkodzie, aby **różnym** kluczom odpowiadał ten sam obiekt.

Wyszukiwanie obiektów na podstawie kluczy jest bardzo szybkie i przy współczynniku załadowania kolekcji rzędu **0.75** sprowadza się do około **2** porównań (sic!).

klasa Hashtable

Hashtable()
Konstruuje obiekt klasy **Hashtable** reprezentujący pustą kolekcję mieszającą, o domyślnym rozmiarze **101** pojemników na odnośniki i współczynniku załadowania **0.75**.
np. `Hashtable hash = new Hashtable();`

`Object` **put**`(Object key, Object obj)`
`    throws NullPointerException`
Umieszcza w kolekcji odnośnik do obiektu ***obj*** związując z tym obiektem klucz ***key***. Jeśli kolekcja nie zawierała podanego klucza, to dostarcza **null**. W przeciwnym razie stan kolekcji się nie zmienia, a funkcja dostarcza odniesienie do obiektu opatrzonego podanym kluczem. Jeśli ***obj*** albo ***key*** ma wartość **null**, to wysyła wyjątek klasy **NullPointerException**.
np. `hash.put("Mary", person);`

`Object` **get**`(Object key)`
Sprawdza, czy w kolekcji znajduje się odnośnik do obiektu z którym związano klucz ***key***, a następnie dostarcza zawarte w nim odniesienie. Dostarcza **null** jeśli nie ma takiego odnośnika.
np. `Person person = (Person)hash.get("Ewa");`

```
boolean containsKey(Object key)
Dostarcza orzeczenie o wartości: "kolekcja zawiera odnośnik do
obiektu  z którym związano klucz key".
np. boolean isKey = hash.containsKey("Ewa");

boolean containsValue(Object obj)
Liniowo i za pomocą funkcji equals, porównuje obiekty identyfikowa-
ne przez odnośniki kolekcji z obiektem identyfikowanym przez obj,
dostarczając orzeczenie o wartości: "kolekcja zawiera odnośnik do
obiektu obj ".
np. boolean isElement = hash.containsValue(person);

Object remove(Object key)
Usuwa z kolekcji odnośnik do obiektu z którym związano klucz key.
Dostarcza odniesienie zawarte w tym odnośniku. Jeśli w kolekcji nie
było takiego odnośnika, to dostarcza null.
np. Person oldPerson = hash.remove(person);

int size()
Dostarcza liczbę kluczy kolekcji.
np. int size = hash.size();

Object clone()
Dostarcza odniesienie do klonu kolekcji. Klonowanie jest płytkie,
to jest obejmuje tylko odnośniki.
np. Hashtable hash2 = (Hashtable)hash.clone();

void clear()
Usuwa z kolekcji wszystkie znajdujące się w niej odnośniki do
obiektów.
np. hash.clear();
```

dla dociekliwych

Funkcje **containsKey(*key*)** i **containsValue(*obj*)** dostarczają poprawnej odpowiedzi tylko wówczas, gdy w klasie klucza ***key*** albo obiektu ***obj*** zdefiniowano własną funkcję **equals** (a nie odziedziczono jej na przykład po klasie **Object**, kiedy to porównanie dotyczyłoby tylko odnośników).

Następująca aplikacja, z wynikami na ekranie ***Kolekcja mieszająca***, tworzy kolekcję odnośników do obiektów reprezentujących osoby, a następnie informuje o nich.

Ekran
Kolekcja mieszająca


```
package jbPack;

import java.util.*;
```

```
public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    private Hashtable persons = new Hashtable();

    public Program()
    {
        persons.put("Ewa", new Person("Ewa", 25));
        persons.put("Iza", new Person("Iza", 17));
        persons.put("Jan", new Person("Jan", 50));

        System.out.println(
            getAge("Iza")  // 17
        );

        System.out.println(
            getAge("Ewa")  // 25
        );
    }

    public int getAge(String key)
    {
        Object object = persons.get(key);
        Person person = (Person)object;
        return person.getAge();
    }
}

class Person {

    private String name;
    private int age;

    public Person(String name, int age)
    {
        this.name = name;
        this.age = age;
    }

    public int getAge()
    {
        return age;
    }
}
```

Projektowanie kolekcji

Kolekcją jest klasa opisująca ***kolektory***, to jest obiekty umożliwiające **kolekcjonowanie** odnośników do obiektów. Typowy kolektor powinien umożliwiać umieszczanie obiektów w kolekcji oraz dostarczanie informacji o ich liczbie.

Wyniki

```
x = 10, y = 10
x = 20, y = 20, z = 20
x = 30, y = 30
```

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        Collector points = new Collector();  // kolektor

        points.add(new Point(10, 10));
        points.add(new Point3D(20, 20, 20));
        points.add(new Point(30, 30));

        for (int i = 0; i < points.size() ; i++) {
            Object object = points.get(i);
            ((Point)object).show();
            System.out.println();
        }
    }
}

    // klasa kolekcyjna
class Collector {

    private int slots = 1;
    private Object[] pObject = new Object [slots];
    private int size;

    public void add(Object pObject)
    {
        if (size == slots) {
            Object[] pObject2 = new Object [2 * slots];
            System.arraycopy(this.pObject, 0, pObject2, 0, slots);
            this.pObject = pObject2;
            slots *= 2;
        }
        this.pObject[size++] = pObject;
    }
```

```
    public Object get(int i)
    {
        return pObject[i];
    }

    public int size()
    {
        return size;
    }
}

class Point {

    protected int x, y;

    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public void show()
    {
        System.out.print(
            "x = " + x + ", y = " + y
        );
    }
}


class Point3D
    extends Point {

    protected int z;

    public Point3D(int x, int y, int z)
    {
        super(x, y);

        this.z = z;
    }

    public void show()
    {
        super.show();
        System.out.print(
            ", z = " + z
        );
    }
}
```

Projektowanie iteracji

Iteracją jest klasa opisująca ***iteratory***, to jest obiekty umożliwiające ***przeglądanie*** kolekcji. Typowy iterator powinien umożliwić udzielanie odpowiedzi na pytanie, czy kolekcja zawiera jeszcze nie-przeiterowane elementy oraz zapewnić dostarczanie odniesień zawartych w kolejnych odnośnikach kolekcji.

Wyniki

```
x = 10, y = 10
x = 20, y = 20, z = 20
x = 30, y = 30
```

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        Collector points = new Collector();  // kolektor

        points.add(new Point(10, 10));
        points.add(new Point3D(20, 20, 20));
        points.add(new Point(30, 30));

        Iterator scanner = new Iterator(points);

        while (scanner.hasNext()) {
            Object object = scanner.getNext();
            ((Point)object).show();
            System.out.println();
        }
    }
}

    // klasa kolekcyjna
class Collector {

    private int slots = 1;
    private Object[] pObject = new Object [slots];
    private int size;

    public void add(Object pObject)
    {
        if (size == slots) {
            Object[] pObject2 = new Object [2 * slots];
            System.arraycopy(this.pObject, 0, pObject2, 0, slots);
            this.pObject = pObject2;
            slots *= 2;
        }
```

```
        this.pObject[size++] = pObject;
    }

    public Object get(int i)
    {
        return pObject[i];
    }

    public int size()
    {
        return size;
    }
}


    // klasa iteracyjna
class Iterator {

    private Collector objects;
    private int pos = 0, count;

    public Iterator(Collector objects)
    {
        this.objects = objects;
        count = objects.size();
    }

    public boolean hasNext()
    {
        return pos < count;
    }

    public Object getNext()
    {
        return objects.get(pos++);
    }
}


class Point {

    protected int x, y;

    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public void show()
    {
        System.out.print(
            "x = " + x + ", y = " + y
        );
    }
}
```

```
class Point3D
    extends Point {

    protected int z;

    public Point3D(int x, int y, int z)
    {
        super(x, y);

        this.z = z;
    }

    public void show()
    {
        super.show();
        System.out.print(
            ", z = " + z
        );
    }
}
```

dla dociekliwych

Zaprojektowane powyżej klasy są oparte na strukturach ***tablicowych***. Nic jednak nie stoi na przeszkodzie oparcia ich na strukturach ***listowych***.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        Collector points = new Collector();  // kolektor

        points.add(new Point(10, 10));
        points.add(new Point(30, 30));

        Collector.Iterator scanner =
            points.new Iterator(points);

        while (scanner.hasNext()) {
            Object object = scanner.getNext();
            ((Point)object).show();
            System.out.println();
        }
    }
}

    // klasa kolekcyjna
class Collector {

    private Pair pHead, pTail;
```

```
    private int size;

    class Pair {

        public Pair pNext;
        public Object pObject;

        public Pair(Object pObject)
        {
            this.pObject = pObject;
        }
    }

    public void add(Object pObject)
    {
        Pair pNew = new Pair(pObject);
        if (pHead == null)
            pHead = pNew;
        else
            pTail.pNext = pNew;
        pTail = pNew;
        size++;
    }

    public int size()
    {
        return size;
    }

        // klasa iteracyjna
    class Iterator {

        private Collector objects;
        private Pair pNext;

        public Iterator(Collector objects)
        {
            this.objects = objects;
            pNext = objects.pHead;
        }

        public boolean hasNext()
        {
            return pNext != null;
        }

        public Object getNext()
        {
            Pair pNext2 = pNext;
            pNext = pNext2.pNext;
            return pNext2.pObject;
        }
    }
}
```

```
class Point {

    protected int x, y;

    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public void show()
    {
        System.out.print(
            "x = " + x + ", y = " + y
        );
    }
}
```

Grafika

Na pulpicie można wykreślać ***napisy***, ***linie***, ***obszary*** i ***obrazy***. W celu wykreślenia obiektu graficznego należy podać jego ***współrzędne*** i wybrać ***narzędzia*** do wykreślania. Wykreślanie może się odbywać w ***kolorach***, a napisy można wykreślać ***czcionkami*** o dowolnie obranym ***kroju***, ***stylu*** i ***rozmiarze***.

Linie rysuje się ***piórami***, a napisy i obszary maluje ***pędzlami***. Pióra mogą ***rysować*** linie ciągłe i przerywane, w dowolnym kolorze i o dowolnej szerokości. Pędzle mogą ***malować*** obszary ***farbami***, ***gradientami*** i ***teksturami***.

Wykreślacz

Wykreślenie napisu, linii, obszaru albo obrazu odbywa się przez wydanie polecenia ***wykreślaczowi***. Odnośnik do własnego wykreślacza dostarczają funkcje **getGraphics** albo **create**, a odniesienie do wykreślacza systemowego otrzymuje się w parametrze funkcji wykreślającej.

Nic nie stoi na przeszkodzie jednoczesnego posługiwania się więcej niż jednym wykreślaczem. Jeśli **własny** wykreślacz nie jest już potrzebny, to zaleca się wydać mu polecenie **dispose**.

```
public void paintComponent(Graphics gDC)
{
    super.paintComponent(gDC);

        // własny wykreślacz
    Graphics pDC = getGraphics();
    // ...
}
```

klasa Graphics

```
Graphics create()
Na podstawie już istniejącego wykreślacza tworzy jego własną kopię,
a odniesienie do niej dostarcza w odnośniku typu Graphics.
np. Graphics pDC = gDC.create();
```

klasa Component

```
Graphics getGraphics()
Tworzy własny wykreślacz klasy Graphics2D, a odniesienie do niego
dostarcza w odnośniku typu Graphics.
np. Graphics pDC = getGraphics();

void dispose()
Zwalnia zasoby systemowe przydzielone własnemu wykreślaczowi.
np. gDC.dispose();
```

```
public
class ContentPane
    extends Content {

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        gDC.drawString("Hello", 100, 100);

        Graphics pDC = getGraphics();
        pDC.drawString("World", 200, 200);

        pDC.dispose();   // ok (własny wykreślacz)
        gDC.dispose();   // błąd (obcy wykreślacz)
    }
}
```

Jeśli wykreślanie jest **buforowane**, to wykreślacz systemowy służy do wykreślania w buforze. Dopiero po zakończeniu wykonywania funkcji **paintComponent**, zawartość bufora jest przenoszona na ekran. Natomiast własny wykreślacz służy do wykreślania na ekranie albo w buforze utworzonym za pomocą funkcji **createImage**.

Współrzędne

Układ współrzędnych ***pojemnika*** jest zamocowany w jego **lewym-górnym** narożniku.

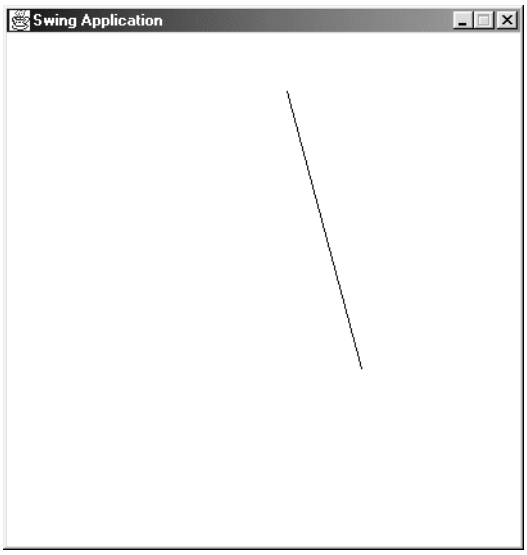
1. W celu wykreślenia **napisu** podaje się współrzędne lewego końca domyślnego ***odcinka bazowego***, na którym napis ***spoczywa***.
   `gDC.drawString("Hello", x, y);`

2. W celu wykreślenia **odcinka**, podaje się współrzędne jego końców.
   `gDC.drawLine(xA, yA, xZ, yZ);`

3. W celu wykreślenia **obszaru** i **obrazu** podaje się współrzędne lewego-górnego wierzchołka domyślnego prostokąta, w który wpisano ten obszar albo obraz.
   `gDC.drawRect(x, y, width, height);`

Następująca pod-aplikacja, pokazana na ekranie ***Wykreślanie odcinków***, wykreśla odcinek łączący **2** przypadkowe punkty pulpitu.

Ekran
Wykreślanie odcinków




```
public
class ContentPane
    extends Content {

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        int w = getWidth(),
            h = getHeight();

        Point p1 = getPoint(w, h),
              p2 = getPoint(w, h);
        gDC.drawLine(p1.x, p1.y, p2.x, p2.y);
    }

    public Point getPoint(int w, int h)
    {
        int x = (int)(Math.random() * w),
            y = (int)(Math.random() * h);
        return new Point(x, y);
    }
}
```

Obiekty graficzne

Najprostszymi obiektami graficznymi są: ***napis***, ***odcinek***, ***owal*** i ***prostokąt***. Napisy oraz **wypełnione** owale i prostokąty są **obszarami**, natomiast odcinki i pozostałe owale i prostokąty są **liniami**.

W wielu następujących opisach jest mowa o **rozmiarach** obiektów graficznych. W tym kontekście należy pamiętać, a wykreślając **wypełniony** prostokąt, łuk i owal (**fillRect**, **clearRect**, **fillArc**, **fillOval**) podaje się jego **pełne** rozmiary, natomiast wykreślając **obrzeże** prostokąta, łuku i owalu (**drawRect**, **drawArc**, **drawOval**) podaje się jego rozmiary **zmniejszone** o **1**. Fakt ten podkreślono w przykładach pisząc np. **200-1** zamiast po prostu **199**.

klasa Graphics

```
void translate(int x, int y)
Przemieszcza układ współrzędnych do punktu (x, y) bieżącego układu
odniesienia.
np. gDC.translate(100, 100);

void drawString(String string, int x, int y)
Wykreśla napis string, spoczywający na odcinku bazowym o lewym
końcu w (x, y).
np. gDC.drawString("Hello", 100, 100);

void drawLine(int xA, int yA, int xZ, int yZ)
Wykreśla odcinek łączący punkty (xA, yA) i (xZ, yZ). Jeśli xA==xZ
oraz yA==yZ, to wykreśla punkt.
np. gDC.drawLine(10, 10, 50, 50);

void drawOval(int x, int y, int w, int h)
Wykreśla owal wpisany w domyślny prostokąt o lewym-górnym wierz-
chołku w (x, y) i rozmiarach określonych przez w x h.
np. gDC.drawOval(0, 0, 200-1, 200-1);

void drawRect(int x, int y, int w, int h)
Wykreśla prostokąt o lewym-górnym wierzchołku w (x, y) i rozmiarach
określonych przez w x h.
np. gDC.drawRect(100, 100, 200-1, 200-1);

void drawRoundRect(int x, int y, int w, int h,
                   int arcW, int arcH)
Wykreśla prostokąt o lewym-górnym wierzchołku w (x, y) i rozmiarach
określonych przez w x h. Narożniki prostokąta są owalami o średni-
cach określonych przez arcW i arcH.
np. gDC.drawRoundRect(100, 100, 200-1, 200-1, 8, 8);

void drawPolygon(int x[], int y[], int n)
Wykreśla łamaną łączącą n punktów o współrzędnych (x[i], y[i]) (dla
i = 0, 1, ... , n-1).
np. gDC.drawPolygon(
        new int[] { 10, 40, 30, 10, },
```

```
        new int[] { 10, 20, 60, 50, },
        4
    );
```

void **drawArc**(int x, int y, int w, int h, int from, int span)
Wykreśla łuk wpisany w domyślny prostokąt o lewym-górnym narożniku w (***x***, ***y***) i rozmiarach określonych przez ***w*** *x* ***h***, od kąta początkowego ***from*** (liczonego od dodatniej półosi *x*, w kierunku przeciwnym do ruchu wskazówek zegara) i rozpiętości ***span*** (wyrażonych **w stopniach**!).
np. gDC.drawArc(100, 100, 50-1, 50-1, -90, 135);

void **fillOval**(int x, int y, int w, int h)
Wykreśla wypełniony owal wpisany w domyślny prostokąt o lewym-górnym wierzchołku w (***x***, ***y***) i rozmiarach ***w*** *x* ***h***.
np. gDC.fillOval(0, 0, 100, 100);

void **fillRect**(int x, int y, int w, int h)
Wykreśla wypełniony prostokąt o lewym-górnym wierzchołku w (***x***, ***y***) i rozmiarach ***w*** *x* ***h***.
np. gDC.fillRect(100, 100, 200, 200);

void **fillRoundRect**(int x, int y, int w, int h,
int arcW, int arcH)
Wykreśla wypełniony prostokąt o lewym-górnym wierzchołku w (***x***, ***y***) i rozmiarach ***w*** *x* ***h***. Narożniki prostokąta są owalami o średnicach ***arcW*** i ***arcH***.
np. gDC.fillRoundRect(100, 100, 200, 200, 8, 8);

void **fillPolygon**(int x[], int y[], int n)
Wykreśla wypełniony wielokąt utworzony przez łamaną łączącą ***n*** punktów o współrzędnych (***x[i]***, ***y[i]***) (dla ***i*** = **0**, **1**, ... , ***n-1***). Jeśli punkt początkowy nie jest identyczny z końcowym, to przed wypełnieniem punkty te łączy się ze sobą.

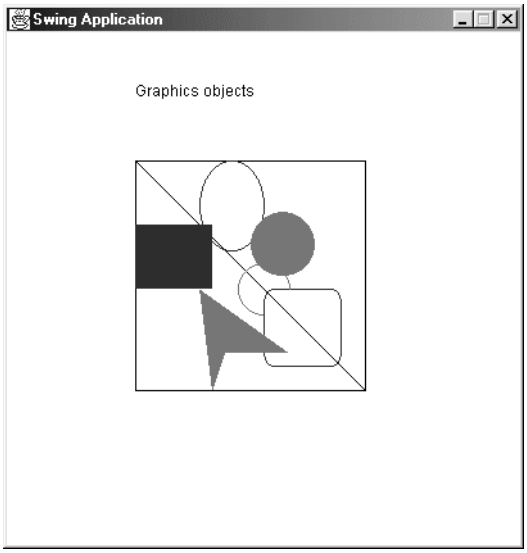
```
np. gDC.fillPolygon(
        new int[] { 10, 40, 30, 10, },
        new int[] { 10, 20, 60, 50, },
        4
    );
```

void **fillArc**(int x, int y, int w, int h, int from, int span)
Wykreśla wypełniony łuk wpisany w domyślny prostokąt o lewym-górnym narożniku w (***x***, ***y***) i rozmiarach ***w*** *x* ***h*** pikseli, od kąta początkowego ***from*** (liczonego od dodatniej półosi *x*, w kierunku przeciwnym do ruchu wskazówek zegara) i rozpiętości ***span*** (wyrażonych **w stopniach**!).
np. gDC.fillArc(100, 100, 50, 50, 45, 90);

void **clearRect**(int x, int y, int w, int h)
Wyczyszcza **kolorem tła** prostokątny obszar o lewym-górnym narożniku w (***x***, ***y***) i rozmiarach ***w*** *x* ***h*** pikseli.
np. gDC.clearRect(100, 200, 20, 40);

Następująca pod-aplikacja, pokazana na ekranie ***Obiekty graficzne***, wykreśla obszerny zestaw obszarów i linii.

Ekran
Obiekty graficzne



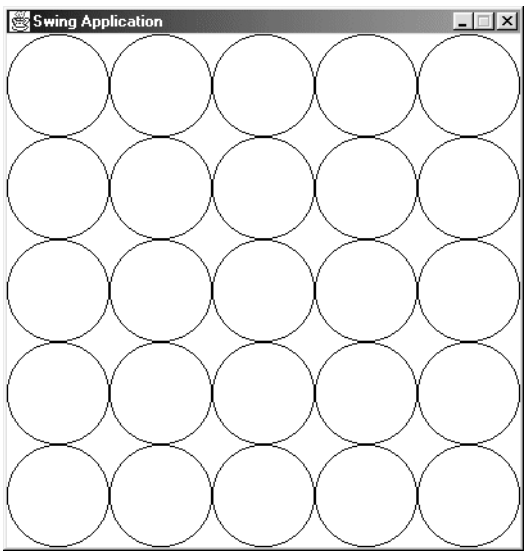

```
public
class ContentPane
    extends Content {

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        // przemieszczenie układu współrzędnych
        gDC.translate(100, 100);

            // wykreślanie linii i obszarów
        gDC.drawLine(0, 0, 180-1, 180-1);
        gDC.drawRect(0, 0, 180-1, 180-1);

        gDC.setColor(Color.blue);
        gDC.fillRect(0, 50, 60, 50);
        gDC.drawRoundRect(100, 100, 60, 60, 15, 25);
        gDC.drawOval(50, 0, 50-1, 70-1);

        gDC.setColor(Color.magenta);
        gDC.fillOval(90, 40, 50, 50);

        gDC.drawArc(80, 80, 40-1, 40-1, 0, 270);

        int[] xVec = {  50, 120,  70,  60, },
              yVec = { 100, 150, 150, 180, };
        gDC.fillPolygon(xVec, yVec, 4);

            // przemieszczenie układu współrzędnych
        gDC.translate(-100, -100);
```

```
            // wykreślenie napisu
        gDC.setColor(Color.black);
        gDC.drawString("Graphics objects", 100, 50);
    }
}
```

Zadanie A

Wykreślić prostokątny układ owali pokrywających pulpit (ekran Zadanie A)

Ekran
Zadanie A

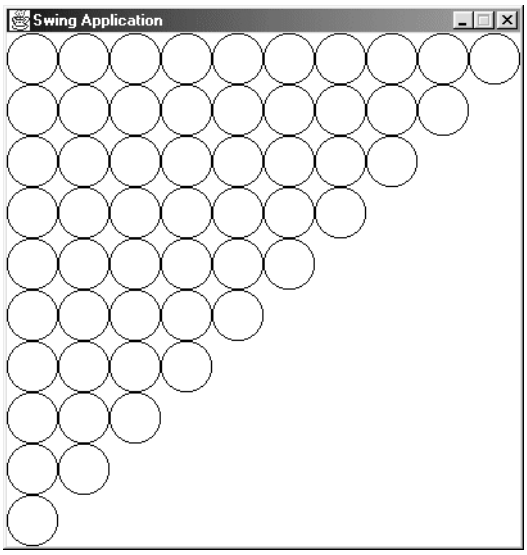



```
public
class ContentPane
    extends Content {

    private final int Count = 5;

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        int w = getWidth()  / Count,
            h = getHeight() / Count;

        for (int y = 0; y < Count; y++) {
            int yLoc = y * h;
            for (int x = 0; x < Count ; x++) {
                int xLoc = x * w;
                gDC.drawOval(xLoc, yLoc, w-1, h-1);
            }
        }
    }
}
```

Zadanie B

Wykreślić trójkątny układ owali dosuniętych do narożnika pulpitu (ekran Zadanie B)

Ekran
Zadanie B

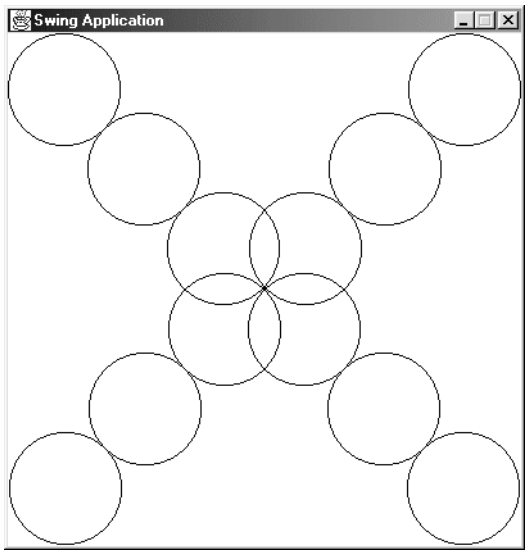



```
public
class ContentPane
    extends Content {

    private final int Count = 10;

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        int w = getWidth()  / Count,
            h = getHeight() / Count;

        for (int y = 0; y < Count; y++) {
            int yLoc = y * h;
            for (int x = 0; x < Count-y ; x++) {
                int xLoc = x * w;
                gDC.drawOval(xLoc, yLoc, w-1, h-1);
            }
        }
    }
}
```

Zadanie C

Wykreślić układ okręgów o środkach położonych na przekątnych pulpitu (ekran Zadanie C)

Ekran
Zadanie C

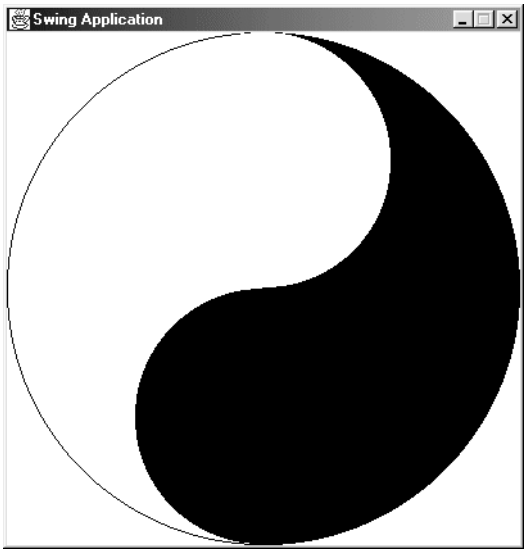



```
public
class ContentPane
    extends Content {

    private final int Count = 6;
    private final double root2 = Math.sqrt(2);

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        int w = getWidth(),
            h = getHeight();

        if (w != h) {
            System.out.println("ContentPane is not square!");
            System.exit(0);
        }

        int r = (int)(w * root2 /
                (Count-1 + root2) / 2 + 0.5);

        for (int i = 0; i < Count; i++) {
            int xyLoc = (int)(i * r * root2 + 0.5);
            gDC.drawOval(xyLoc, xyLoc, 2*r-1, 2*r-1);
            int xLoc = w - xyLoc - 2*r;
            gDC.drawOval(xLoc, xyLoc, 2*r-1, 2*r-1);
        }
    }
}
```

Zadanie D

Wykreślić chiński symbol przenikania sprzeczności (ekran Zadanie D)

Ekran *Zadanie D*




```
public
class ContentPane
    extends Content {

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        int w = getWidth(),
            h = getHeight();

        gDC.drawOval(0, 0, w-1, h-1);

        gDC.fillArc(0, 0, w-1, h-1, 270, 180);
        gDC.fillArc(w/4, h/2, w/2-1, h/2-1, 0, 360);

        gDC.setColor(Color.white);
        gDC.fillArc(w/4, 0, w/2-1, h/2-1, 0, 360);
    }
}
```

Położenie i rozmiary

Obiekty graficzne mają określone ***położenie*** i ***rozmiar***. Położenie jest reprezentowane w obiekcie klasy **Point**, a rozmiary w obiekcie klasy **Dimension**.

Wszystkie pola klas **Point** i **Dimension** są **publiczne**.

klasa Point

Point(int x, int y)
Konstruuje obiekt klasy **Point** reprezentujący punkt o współrzędnych (***x***, ***y***).
np. Point point = new Point(100, 100);

x
Dostarcza współrzędną ***x***.
np. int x = point.x;

y
Dostarcza współrzędną ***y***.
np. int y = point.y;

void **setLocation**(int x, int y)
Zmienia współrzędne punktu na (***x***, ***y***).
np. point.setLocation(200, 200);

klasa Dimension

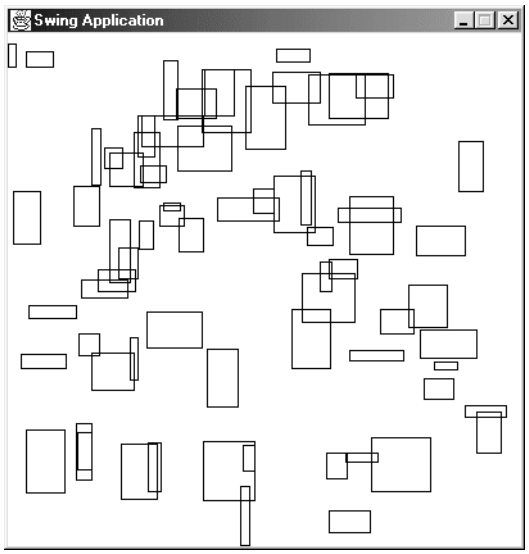
Dimension(int w, int h)
Konstruuje obiekt klasy **Dimension** reprezentujący rozmiary ***w*** *x* ***h***.
np. Dimension dim = new Dimension(50, 50);

width
Dostarcza szerokość.
np. int w = dim.width;

height
Dostarcza wysokość.
np. int h = dim.height;

Następująca pod-aplikacja, pokazana na ekranie ***Położenie i rozmiary***, wykreśla zestaw prostokątów o przypadkowym położeniu i rozmiarach. W celu jej uproszczenia zrezygnowano z odtwarzania pulpitu.

Ekran
Położenie i rozmiary




```
public
class ContentPane
    extends Content {

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        int w = getWidth(),
            h = getHeight(),
            sw, sh;

        for (int i = 0; i < 100 ; i++) {
            Point point =
                new Point(
                    getRandom(w), getRandom(h)
                );
            Dimension dim =
                new Dimension(
                   sw = getRandom(50),
                   sh = getRandom(50)
                );

            if (sw > 5 && sh > 5 &&
               point.x + sw < w &&
               point.y + sh < h) {
                gDC.drawRect(
                    point.x, point.y,
                    dim.width, dim.height
                );
            }
        }
    }
```

```
    public int getRandom(int span)
    {
        return (int)(Math.random() * span);
    }
}
```

Kolory

Najczęściej używanym ***modelem koloru*** jest ***ARGB***. W modelu *ARGB* każdy kolor jest opisany przez **4** liczby z domkniętego przedziału **0** .. **255**, z których trzy określają stopień nasycenia kolorem ***czerwonym***, ***zielonym*** i ***niebieskim***, a czwarta określa stopień ***nieprzezroczystości*** koloru. Trzynaście kolorów całkowicie nieprzezroczystych jest reprezentowanych przez symbole wymienione w tabeli ***Kolory podstawowe***.

Tabela Kolory podstawowe

black	**blue**	**cyan**
darkGray	**gray**	**green**
lightGray	**magenta**	**orange**
pink	**red**	**white**
	yellow	

Do utworzenia obiektu reprezentującego kolor używa się fabrykatora. Jego argumentami są zazwyczaj składowe koloru; np. zestaw **255,0,0** definiuje kolor **czerwony** (**Color.red**), a **255,255,0** definiuje kolor żółty (**Color.yellow**).

klasa Color

Color(int red, int green, int blue)
Color(int red, int green, int blue, int alpha)
Konstruuje obiekt klasy **Color** reprezentujący nieprzezroczysty kolor o składowych ***red***, ***green***, ***blue*** albo kolor o stopniu **nieprzezroczystości** określonym przez ***alpha*** (dla **0** całkowicie przezroczysty, a dla **255** całkowicie nieprzezroczysty).

```
np. Color color = new Color(255, 255, 0);
```

Color(int aRGB)
Color(int aRGB, boolean isAlpha)
Konstruuje obiekt klasy **Color** reprezentujący nieprzezroczysty kolor o składowych określonych przez ***aRGB*** albo kolor ze składową nieprzezroczystości określoną przez najbardziej znaczący bajt ***aRGB*** (jeśli ***isAlpha*** ma wartość **true**).

```
np. Color color =
        new Color((128 << 24) + (255<<16), true);
```

klasa Graphics

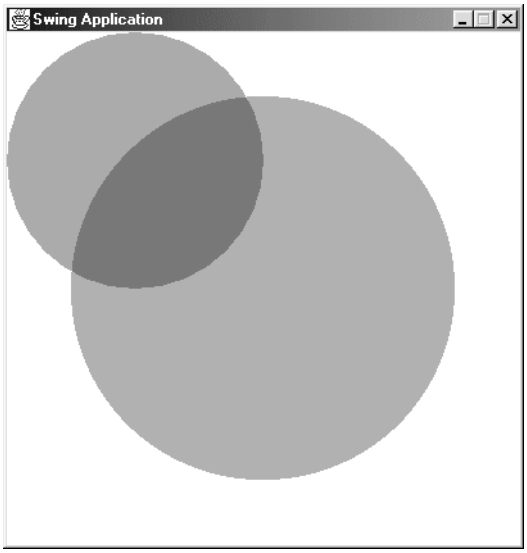
```
void setColor(Color color)
Wstawia do wykreślacza kolor color. Staje się on jednocześnie kolo-
rem aktualnie używanego pióra i pędzla.
np. gDC.setColor(Color.cyan);

Color getColor()
Dostarcza kolor wstawiony do wykreślacza.
np. Color color = gDC.getColor();
```

klasa Color

```
int getRed()
Dostarcza wartość składowej czerwonej koloru.
np. int red = color.getRed();

int getGreen()
Dostarcza wartość składowej zielonej koloru.
np. int green = color.getGreen();

int getBlue()
Dostarcza wartość składowej niebieskiej koloru.
np. int blue = color.getBlue();

int getAlpha()
Dostarcza wartość składowej nieprzezroczystości koloru.
np. int alpha = color.getAlpha();
```

Następująca pod-aplikacja, pokazana na ekranie ***Dobieranie kolorów***, wykreśla elipsę w kolorze przypadkowym, a na konsoli podaje jej składowe *RGB*. Ponadto wykreśla dodatkowe koło w kolorze półprzezroczystym.

Ekran
Dobieranie kolorów



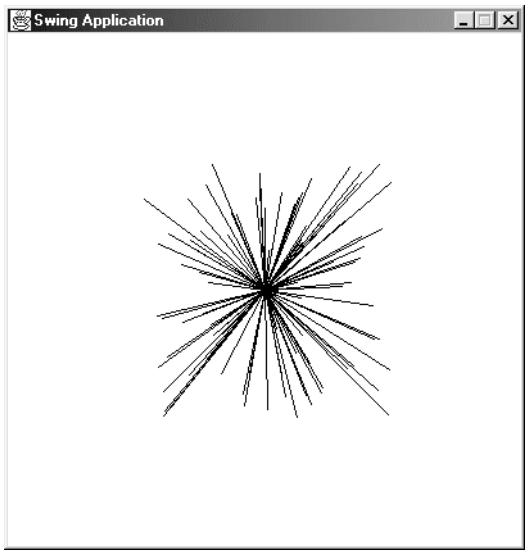
```
public
class ContentPane
    extends Content {

    private int w, h;

    public void initPane()
    {
        w = getWidth();
        h = getHeight();
    }

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        Color color = getColor();
        gDC.setColor(color);

        gDC.fillOval(w/8, h/8, 3*w/4-1, 3*h/4-1);

        int red   = color.getRed(),
            green = color.getGreen(),
            blue  = color.getBlue();

        System.out.println(
            "" + red + ' ' + green + ' ' + blue
        );

        gDC.setColor(
            new Color(0, 0, 255, 128)
        );
        gDC.fillOval(0, 0, w/2, h/2);
    }

    public Color getColor()
    {
        int r = (int)(Math.random() * 256),
            g = (int)(Math.random() * 256),
            b = (int)(Math.random() * 256);
        return new Color(r, g, b);
    }
}
```

Pióra

Wstawienie do wykreślacza koloru określa jednocześnie kolor ***pióra*** używanego do wykreślania linii.

Następująca pod-aplikacja, pokazana na ekranie ***Użycie pióra***, ilustruje użycie pióra do wykreślania linii.

Ekran
Użycie pióra



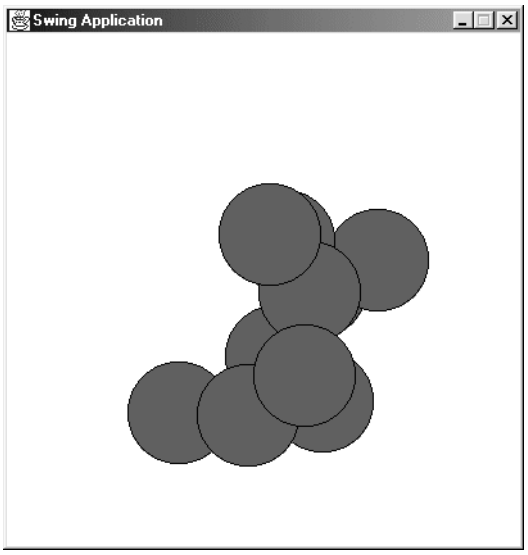

```
public
class ContentPane
    extends Content {

    private int Count = 100;
    private int w2, h2;

    public void initPane()
    {
        w2 = getWidth()  / 2;
        h2 = getHeight() / 2;
    }

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        for (int i = 0; i < Count ; i++) {
            int x = getRand(w2),
                y = getRand(h2);
            gDC.drawLine(w2, h2, w2+x, h2+y);
        }
    }

    public int getRand(int lim)
    {
        double random = Math.random();
        return (int)(random * lim) - lim/2;
    }
}
```

Pędzle

Wstawienie do wykreślacza koloru określa jednocześnie kolor ***pędzla*** używanego do malowania obszarów.

Następująca pod-aplikacja, pokazana na ekranie ***Użycie pędzla***, ilustruje użycie pędzla do malowania obszarów.

Ekran
Użycie pędzla




```
public
class ContentPane
    extends Content {

    private int Count = 10,
                r = 40, d = 2*r;
    private int w2, h2;

    public void initPane()
    {
        w2 = getWidth()  / 2;
        h2 = getHeight() / 2;
    }

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        for (int i = 0; i < Count ; i++) {
            int x = getRand(w2),
                y = getRand(h2);
            gDC.setColor(Color.red);
            gDC.fillOval(w2+x-r, h2+y-r, d, d);
```

```
            gDC.setColor(Color.black);
            gDC.drawOval(w2+x-r, h2+y-r, d-1, d-1);
        }
    }

    public int getRand(int lim)
    {
        double random = Math.random();
        return (int)(random * lim) - lim/2;
    }
}
```

Czcionki

Czcionkę opisuje ***krój***, ***styl*** i ***rozmiar***. Wypełnienie znaków napisu wykreślanego wybraną czcionką odbywa się za pomocą **pędzla**.

Przed przystąpieniem do wykreślania znaków, utworzony za pomocą fabrykatora klasy **Font** obiekt czcionki należy wstawić do wykreślacza.

Następująca pod-aplikacja wykreśla na pulpicie, pogrubioną **64** punktową czcionką, napis **Hello**.

```
public
class ContentPane
    extends Content {

    private Font font = new Font("Lucida Sans", Font.BOLD, 64);

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        gDC.setFont(font);  // wstawienie czcionki do wykreślacza
        gDC.drawString("Hello", 20, 100);
    }
}
```

W każdym *Systemie* można się posługiwać czcionkami o krojach

Serif, **Sansserif**, **Monospaced**, **Dialog**, **DialogInput**
Lucida Sans, Lucida Bright, Lucida Sans Typewriter

o stylach

BOLD, **ITALIC**, **BOLD** i **ITALIC**, **PLAIN**

oraz o rozmiarach wyrażonych nie w pikselach, ale w ***punktach*** (**1 pt = 1/72 cala**).

Wykreślanie polskich liter jest możliwe

1. We wszystkich *Systemach*: za pomocą czcionek rodziny ***Lucida*** (z wyjątkiem **Lucida Sans** PLAIN).
2. W systemach ***spolonizowanych***: za pomocą czcionek **Serif**, **SansSerif**, **Monospaced** i **Dialog**.

Czcionki krojów ***Lucida*** umożliwiają także wykreślanie znaków języków zachodnio-europejskich, a ponadto znaków arabskich, hebrajskich oraz cyrylicy.

klasa Font

```
Font(String face, int style, int size)
Konstruuje obiekt klasy Font reprezentujący czcionkę o kroju face,
stylu style i rozmiarze size.
np. Font font =
        new Font(
            "Serif",                    // krój
            Font.BOLD | Font.ITALIC,    // styl
            60                          // rozmiar
        );
```

klasa Graphics

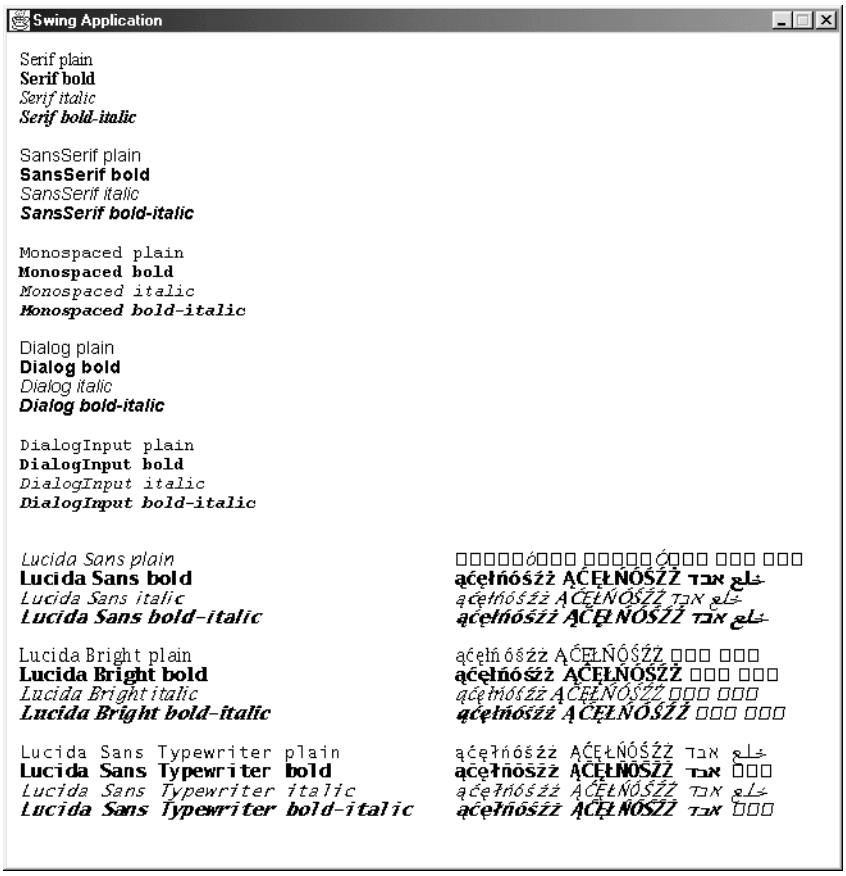
```
void setFont(Font font)
Wstawia do wykreślacza czcionkę font.
np. gDC.setFont(
        new Font("Monospaced", Font.PLAIN, 10)
    );

Font getFont()
Dostarcza czcionkę wstawioną do wykreślacza.
np. Font font = gDC.getFont();
```

Następująca pod-aplikacja, pokazana na ekranie ***Dobieranie czcionek***, wykreśla zestaw napisów, posługując się czcionkami o różnych **krojach**, **rozmiarach** i **stylach**. Dla wszystkich krojów wykreślono ***polskie litery***, a dla krojów ***Lucida*** kilka znaków ***arabskich*** i ***hebrajskich***. W celu uwidocznienia wszystkich znaków nadano pulpitowi rozmiary **650** x **650** pikseli.

Mimo iż znaki arabskie są wymienione w środku napisu, zgodnie z oczekiwaniami są wyprowadzane od końca wiersza (sic!).

Ekran

Dobieranie czcionek




```
public
class ContentPane
    extends Content {

    private String string;

    private String typeFace[] =
        {
            "Serif",
            "SansSerif",
            "Monospaced",
            "Dialog",
            "DialogInput",

            "Lucida Sans",
            "Lucida Bright",
            "Lucida Sans Typewriter",
        };
```

```
    private int style[] =
        {
            Font.PLAIN,
            Font.BOLD,
            Font.ITALIC,
            Font.BOLD + Font.ITALIC
        };

    private String styleName[] =
        {
            " plain",
            " bold",
            " italic",
            " bold-italic"
        };

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        for (int n = 1, i = 0; i < typeFace.length ; i++) {
            if (i == 5)
                n++;

            String fontName = typeFace[i];

            for (int j = 0; j < style.length ; j++) {
                int fontStyle = style[j];
                Font font =
                    new Font(fontName, fontStyle, 14);
                gDC.setFont(font);

                string = fontName + styleName[j];
                gDC.drawString(string, 10, 10 + 15 * n);

                String polish = "ąćęłńóśźż ĄĆĘŁŃÓŚ•Ż",
                       arabic = "\u062e\u0644\u0639",
                       hebrew = "\u05d0\u05d1\u05d3";

                int offset = 350;
                if (i > 4) {
                    gDC.drawString(
                        polish + ' ' + arabic + ' ' + hebrew,
                        offset, 10 + 15 * n
                    );
                }
                n++;
            }
            n++;
        }
    }
}
```

Metryki

Metryka opisuje takie właściwości czcionki jak ***uniesienie*** i ***obniżenie*** jej znaków względem linii bazowej (*ascent* i *descent*), optymalny ***odstęp*** między wierszami napisów (*leading*) oraz ***wysokość*** czcionki (*height*).

klasa Graphics

```
FontMetrics getFontMetrics()
Dostarcza metrykę czcionki aktualnie wstawionej do wykreślacza.
np. FontMetrics mtx = gDC.getFontMetrics();
```

klasa Component

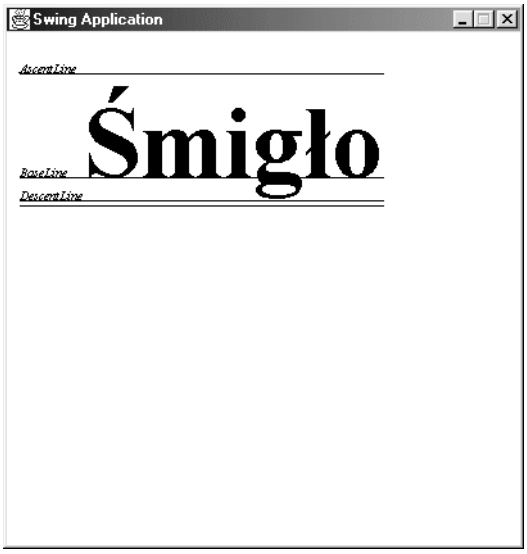
```
FontMetrics getFontMetrics(Font font)
Dostarcza metrykę czcionki font komponentu.
np. FontMetrics mtx =
        getFontMetrics(new Font("Serif", 0, 12));
```

klasa FontMetrics

```
int getAscent()
Dostarcza uniesienie znaków czcionki względem odcinka bazowego.
np. int a = mtx.getAscent();

int getDescent()
Dostarcza obniżenie znaków czcionki względem odcinka bazowego.
np. int d = mtx.getDescent();

int getLeading()
Dostarcza optymalny odstęp między kolejnymi rzędami napisów wykre-
ślonych daną czcionką.
np. int l = mtx.getLeading();

int getHeight()
Dostarcza wysokość czcionki (sumę a+d+l).
np. int h = mtx.getHeight();

int stringWidth(String string)
Dostarcza szerokość napisu string wyrażoną w pikselach.
np. int w = mtx.stringWidth("Kaja");
```

Następująca pod-aplikacja, pokazana na ekranie ***Metryka czcionki***, wykreśla podany napis wraz z jego linią bazową oraz liniami uniesienia, obniżenia i odstępu.

Ekran
Metryka czcionki




```
public
class ContentPane
    extends Content {

    private String string = "Śmigło";
    private Font font = new Font("Serif", Font.BOLD, 80);
    private FontMetrics mtx = getFontMetrics(font);
    private int strWidth = mtx.stringWidth(string);

    private int xBase = 10, yBase,
                strAscent, strDescent,
                strHeight, strLeading;

    public void initPane()
    {
        strAscent  = mtx.getAscent();
        strDescent = mtx.getDescent();
        strHeight  = mtx.getHeight();

        strLeading = strHeight - (strAscent + strDescent);
        yBase = (int)strHeight + 10;
    }

    void draw(Graphics gDC, String caption, int h)
    {
        gDC.drawLine(xBase, yBase-h,
                     xBase + 50 + strWidth, yBase-h);

        Font oldFont = gDC.getFont(),
             newFont = new Font("Serif", Font.ITALIC, 10);
```

```
        gDC.setFont(newFont);
        gDC.drawString(caption, xBase, yBase-h);

        gDC.setFont(oldFont);
    }

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        gDC.setFont(font);
        gDC.drawString(string, xBase + 50, yBase);

        draw(gDC, "BaseLine", 0);
        draw(gDC, "AscentLine", strAscent);
        draw(gDC, "DescentLine", -strDescent);
        draw(gDC, "", -strDescent-strLeading);
    }
}
```

Wykreślanie

Wykreślaczowi można narzucić ***tryb wykreślania***. Ponieważ dwukrotne wykreślenie obiektu w tym samym ***trybie XOR*** powoduje **przywrócenie** pierwotnego tła wykresu, więc użycie takiego trybu umożliwia sporządzanie wykresów ***próbnych***.

klasa Graphics

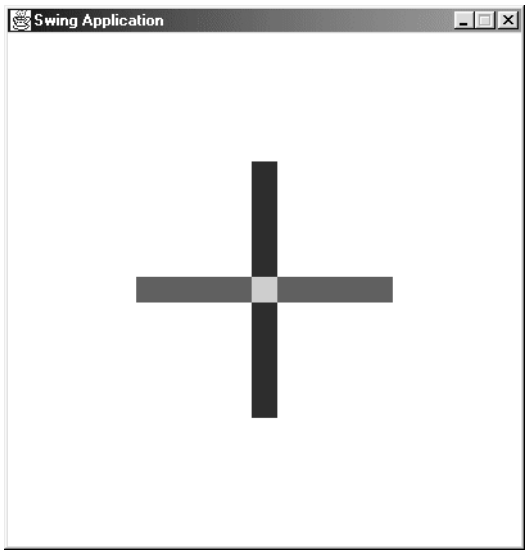
`void` **`setXORMode`**`(Color color)`
Dostosowuje wykreślacz do sporządzania wykresów w trybie *XOR* z kolorem ***color*** (najlepiej z kolorem tła). Kolor wynikowy tworzy się z **podanego koloru**, **koloru bieżącego** i **koloru tła**. Reprezentacje tych **3** kolorów sumuje się zazwyczaj *modulo-2*.
np. `gDC.setXORMode(Color.red);`

`void` **`setPaintMode`**`()`
Dostosowuje wykreślacz do wykreślania w trybie **zwykłym**.
np. `gDC.setPaintMode();`

Następująca pod-aplikacja, pokazana na ekranie ***Użycie trybu XOR***, wykreśla poziomy prostokąt **czerwony**, a po **0.5 s**, w trybie *XOR*, przecinający go pionowy prostokąt **niebieski**. Powoduje to, że obszar przecięcia staje się **zielony**. Po upływie **0.5 s** prostokąt pionowy jest wykreślany **ponownie**. Powoduje to przywrócenie wyglądu poziomego prostokąta.

W celu ujawnienia poszczególnych faz wykreślania, **wyłączono** buforowanie wykreślania.

Ekran
Użycie trybu XOR




```
public
class ContentPane
    extends Content {

    public void initPane()
    {
        RepaintManager.currentManager(this).
            setDoubleBufferingEnabled(false);
    }

    private int s = 20, s2 = s/2;

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        int w = getWidth(),
            h = getHeight();

        gDC.setColor(Color.red);
        gDC.fillRect(w/4, h/2-s2, w/2, s);

            // wstrzymanie wykonania
        try {
            Thread.sleep(500);
        }
        catch (InterruptedException e) {}

        gDC.setColor(Color.blue);

        gDC.setXORMode(Color.white);

        gDC.fillRect(w/2-s2, h/4, s, h/2);
```

```
            // wstrzymanie wykonania
        try {

            Thread.sleep(500);
        }
        catch (InterruptedException e) {}

        gDC.fillRect(w/2-s2, h/4, s, h/2);
    }
}
```

Obcinanie

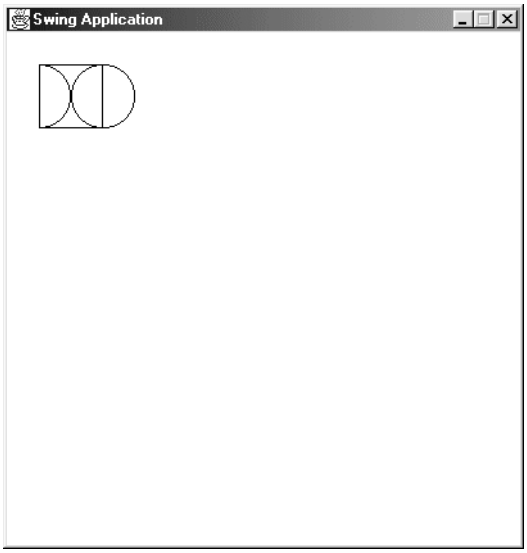
Do wykreślacza można wstawić ***obszar obcinania***. Polecenie wykreślenia piksela poza obszarem obcinania jest **ignorowane**. Do określenia obszaru obcinania związanego z wykreślaczem komponentu służy metoda **setClip**.

klasa Graphics

```
void setClip(int x, int y, int w, int h)
Definiuje jako obszar obcinania, prostokąt o współrzędnych narożni-
ka (x,y) i rozmiarach w x h.
np. gDC.setClip(10, 10, 100, 100);
```

Następująca pod-aplikacja, pokazana na ekranie ***Obszar obcinania***, wykreśla prostokąt, a następnie zajęty przezeń obszar definiuje jako obszar obcinania. Dlatego wykres jednego z okręgów jest **obcięty** do wybranego prostokąta.

Ekran
Obszar obcinania



```
public
class ContentPane
    extends Content {

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        gDC.drawRect(25, 25, 50-1, 50-1);

            // zmiana domniemanego obszaru obcinania
        gDC.setClip(25, 25, 50, 50);

        gDC.drawOval(0, 25, 50-1, 50-1);

        Dimension dim = getSize();
        int w = dim.width,
            h = dim.height;

            // przywrócenie obszaru obcinania
        gDC.setClip(0, 0, w-1, h-1);
        gDC.drawOval(50, 25, 50-1, 50-1);
    }
}
```

Zasoby

Zasobem programu jest ***plik*** z danymi, ***klip*** dźwiękowy, ***obraz***, itp. Zasoby dzielą się na ***lokalne*** i ***odległe***. Zasób lokalny znajduje się w obrębie lokalnego komputera, a zasób odległy znajduje się **w sieci**.

Położenie zasobu określa jego ***lokalizator***. W ogólnym przypadku ma on postać

protocol`://`***host***`:`***port***`/`***path***`/`***file***`#`***ref***

w której ***protocol*** jest nazwą protokołu komunikacyjnego (np. **file**, **http**, **ftp**, **telnet**), ***host*** jest nazwą komputera-gospodarza (np. **www.sun.com**), ***port*** jest numerem portu (dla **http** domyślnie **80**), ***path*** jest ścieżką (np. **~jbl**), ***file*** jest nazwą pliku albo katalogu (np. **Duke.gif**), a ***ref*** jest zakładką w pliku *HTML* (np. **Chapter2**). Nazwę katalogu kończy w lokalizatorze ***skośnik*** (/).

W szczególności, jeśli zasobem jest plik **Duke.gif**, to jego lokalizatorem może być

```
file:/C:/jbJava/jbGifs/Duke.gif
```

albo

```
http://bolek.ii.pw.edu.pl/~janb/jbGifs/Duke.gif
```

Po napisie **file:** występuje jeden skośnik, a po napisie **http:** dwa.

Obiekty lokalizujące

Obiektem ***lokalizującym*** jest obiekt reprezentujący lokalizator.

klasa URL

URL(String locator)
URL(String protocol, String host, String file)
Konstruuje obiekt klasy **URL** reprezentujący lokalizator w pełni opisany przez ***locator*** albo lokalizator opisany przez składowe ***protocol***, ***host*** i ***file***.
np. URL duke = new URL("http://www.sun.com/Duke.gif");

Poszukiwanie zasobów

W celu dostarczenia lokalizatora zasobu należy użyć funkcji **getResource** klasy **Class** albo funkcji **toURL** klasy **File**.

Funkcja **getResource** sięga do katalogu, z którego załadowano plik **.class** klasy wywołującej. Z tego powodu nadaje się do stosowania w programach dystrybuowanych za pomocą plików ***.class**.

Name.class
Dostarcza odniesienie do unikalnego obiektu klasy **Class** reprezentującego klasę ***Name***.
np. Class classObj = Program.class;

klasa Class

Class **forName**(String name)
Dostarcza odniesienie do unikalnego obiektu klasy **Class** reprezentującego klasę ***name***.
np. Class classObj = Class.forName("Program");

Class **getClass**()
Dostarcza odniesienie do obiektu reprezentującego unikalny obiekt klasy, z której wywołano tę funkcję.
np. Class classObj = getClass();

URL **getResource**(String fileName)
Dostarcza lokalizator zasobu znajdującego się w pliku ***fileName***. Poszukuje pliku w tym samym katalogu, z którego załadowano *B*-kod klasy reprezentowanej przez obiekt, któremu wydano to polecenie. Dostarcza **null** jeśli zasobu nie znaleziono.
np. URL duke = Program.class.getResource("Duke.gif");

W ciele klasy *Name*, zarówno ***Name*.class** jak i **getClass()** jest nazwą anonimowego odnośnika do obiektu reprezentującego klasę *Name*. Argumentem funkcji **getResource** może być tylko nazwa względna (np. **jbImages/Duke.gif**, ale nie **d:/jbImages/Duke.gif**).

Następująca aplikacja sięga po zasób **Duke.gif**. Jeśli zasób znajduje się **w tym samym** katalogu co plik **Program.class**, to aplikacja podaje jego lokalizator. W przeciwnym razie informuje, że zasób nie istnieje.

```
package jbPack;

import java.net.*;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    private String fileName = "Duke.gif";

    public Program()
    {
        URL url = Program.class.getResource(fileName);
        if (url != null)
            System.out.println(
                url   // file:/D:/jbKawa40/jbPack/Duke.gif
            );
        else
            System.out.println("Resource does not exist");
    }
}
```

Następująca aplikacja sprawdza, czy istnieje plik o podanej nazwie i podaje jego lokalizator. Poszukiwanie pliku odbywa się w katalogu **projektowym**.

```
package jbPack;

import java.io.*;
import java.net.*;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    private String fileName = "Duke.gif";
```

```
    public Program()
    {
        File file = new File(fileName);
        try {
            if (file.exists() && file.isFile()) {
                URL url = file.toURL();
                System.out.println(
                    url   // file:/D:/jbKawa40/Duke.gif
                );
            } else
                System.out.println(
                    fileName + " does not exist!"
                );
        }
        catch (MalformedURLException e) {}
    }
}
```

Przybornik

Osadzenie *Maszyny Wirtualnej* na nowej platformie wymaga implementowania funkcji zadeklarowanych w klasach pakietu **java.awt.peer**. Ich definicji dostarcza się w ***klasie przybornika***.

Wśród przyborników platformy jeden jest ***przybornikiem domyślnym***. Odnośnik do przybornika domyślnego otrzymuje się za pomocą funkcji **getDefaultToolkit**. Odnośniki do przyborników związanych z komponentami otrzymuje się za pomocą funkcji **getToolkit** klasy **Component**.

klasa Toolkit

```
Toolkit getDefaultToolkit()
Dostarcza odniesienie do przybornika.
np. Toolkit kit = Toolkit.getDefaultToolkit();

void beep()
Wydaje sygnał dźwiękowy.
np. kit.beep();

Dimension getScreenSize()
Dostarcza rozmiary ekranu wyrażone w pikselach.
np. Dimension d = kit.getScreenSize();

int getScreenResolution()
Dostarcza rozdzielczość ekranu (w pikselach na cal).
np. int res = kit.getScreenResolution();

String getProperty(String key, String defValue)
Dostarcza właściwość key. Jeśli jej nie ma, to dostarcza defValue.
np. String dir = kit.getProperty("user.dir", "d:\jbKawa40");
```

W tabeli ***Właściwości systemowe*** wyszczególniono wybrane właściwości wyprowadzone na konsolę za pomocą instrukcji

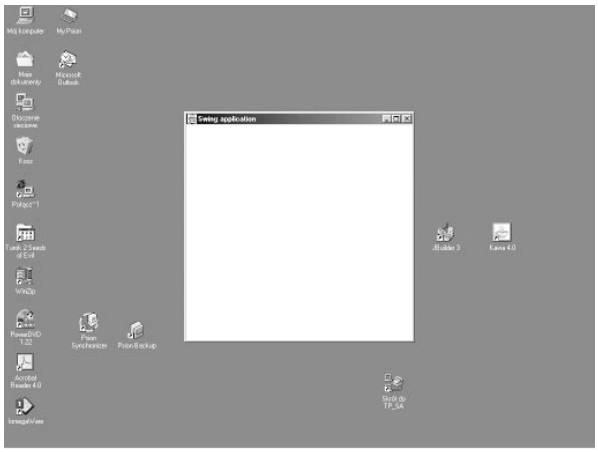
```
System.getProperties().list(System.out);
```

Tabela Właściwości systemowe

```
java.runtime.name=Java(TM) 2 Runtime Environment, Stand...
sun.boot.library.path=D:\JDK1.3\JRE\bin
java.vm.version=1.3.0-C
java.vm.vendor=Sun Microsystems Inc.
java.vm.name=Java HotSpot(TM) Client VM
user.dir=D:\jbKawa\jbApplication
java.runtime.version=1.3.0-C
line.separator=

os.name=Windows 98
java.library.path=D:\JDK1.3\BIN;.;C:\WINDOWS\SYSTEM;C:\...
user.home=C:\WINDOWS
file.encoding=Cp1250
java.specification.version=1.3
user.name=Jan Bielecki
java.class.path=d:\jbKawa\jbApplication;.;D:\KawaEnt5...
java.vm.specification.version=1.0
java.home=D:\JDK1.3\JRE
user.language=pl
java.version=1.3.0
java.ext.dirs=D:\JDK1.3\JRE\lib\ext
file.separator=\
sun.cpu.endian=little
user.region=PL
```

Następująca aplikacja, pokazana na ekranie ***Przyborniki platformy***, tworzy puste okno o rozmiarach **400** *x* **400** pikseli i umieszcza je na środku ekranu.

Ekran
Przyborniki platformy



```
package jbPack;

import javax.swing.*;
import java.awt.*;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    private int w = 400, h = 400;
    private Toolkit kit =
        Toolkit.getDefaultToolkit();

    public Program()
    {
        final JFrame frame =
            new JFrame("Swing application");

            // rozmiary ekranu
        Dimension d = kit.getScreenSize();
        int scrW = d.width,
            scrH = d.height;

            // kolor tła pulpitu
        Container pane = frame.getContentPane();
        pane.setBackground(Color.white);

            // położenie okna
        frame.setLocation(
            (scrW - w) / 2,
            (scrH - h) / 2
        );

            // rozmiary okna
        frame.setSize(w, h);

        frame.show();
    }
}
```

Obrazy

Wykreślenie obrazu zapamiętanego w pliku w formacie *GIF*, *JPG* albo *PNG*, należy poprzedzić jego ***załadowaniem***. Czynność tę wykonuje się fabrykując obiekt klasy **ImageIcon**. Bezpośrednio po sfabrykowaniu obiektu, obraz znajduje się w pamięci operacyjnej i jest gotowy do wykreślenia.

Nie należy stosować funkcji **getImage** klasy **Component**, ponieważ mimo iż dostarcza odniesienie do obiektu reprezentującego obraz, to ładuje go zazwyczaj dopiero tuż przed wykreśleniem, a więc o wiele za późno.

klasa ImageIcon

```
ImageIcon(String fileName)
ImageIcon(URL url)
Konstruuje obiekt klasy ImageIcon i ładuje go ikoną z pliku fileNa-
me albo z pliku zlokalizowanego przez url. Po powrocie z konstruk-
tora obraz (o ile istnieje) jest już załadowany.
np. ImageIcon icon = new ImageIcon("Duke.gif");

int getIconWidth()
Dostarcza szerokość obrazu. Dostarcza -1 jeśli obraz nie istnieje.
np. int w = icon.getIconWidth();

int getIconHeight()
Dostarcza wysokość obrazu. Dostarcza -1 jeśli obraz nie istnieje.
np. int h = icon.getIconHeight();

Image getImage()
Dostarcza odniesienie do obiektu klasy Image reprezentującego ob-
raz. Nawet jeśli nie udało się załadować obrazu, odniesienie to ma
wartość różną od null.
np. Image img = icon.getImage();
```

Uwaga: Nazwa pliku o postaci

```
relPath/Duke.gif
```

jest nazwą ***względną***, odniesioną do katalogu ***bieżącego*** (w *Kawie* jest nim katalog projektowy), nazwa o postaci

```
file:/c:/jbGifs/Duke.gif
```

jest nazwą ***bezwzględną*** dotyczącą **lokalnego** komputera, a nazwa o postaci

```
http://bolek.ii.pw.edu.pl/~janb/jbGifs/Duke.gif
```

jest nazwą bezwzględną dotyczącą ***odległego*** komputera.

klasa Graphics

```
void drawImage(Image image, int x, int y, ImageObserver obs)
void drawImage(Image image, int x, int y,
               int w, int h, ImageObserver obs)
Obraz reprezentowany przez image wykreśla w taki sposób, że współ-
rzędne jego lewego-górnego narożnika umieszcza w punkcie (x, y),
a jeśli podano parametry w i h, to przeskalowuje go i wykreśla
```

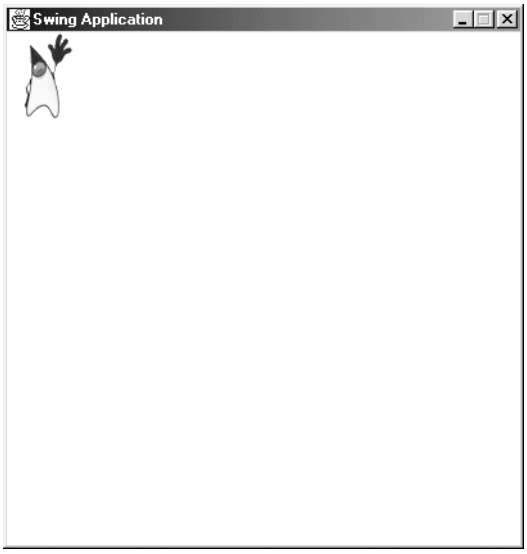
w prostokącie o rozmiarach ***w*** x ***h***. Wykreślanie odbywa się pod nadzorem obserwatora ładowania ***obs***. Może być nim dowolny komponent.
np. gDC.drawImage(img, 100, 100, null);

```
void drawImage(Image image,
               int x1d, int y1d, int x2d, int y2d,
               int x1s, int y1s, int x2s, int y2s,
               ImageObserver obs)
```

Obraz reprezentowany przez ***image*** w prostokącie (***x1s***, ***y1s***), (***x2s***, ***y2s***) wykreśla w prostokącie (***x1d***, ***y1d***), (***x2d***, ***y2d***). Wykreślanie odbywa się pod nadzorem obserwatora ładowania ***obs***. Może być nim dowolny komponent.
np. gDC.drawImage(img, 10, 10, 20, 20, 50, 50, 70, 70, null);

Następująca pod-aplikacja, pokazana na ekranie ***Wykreślenie obrazu***, wykreśla obraz albo informuje o jego braku. Poszukiwanie obrazu odbywa się w ***katalogu bieżącym***, a badanie czy plik **istnieje** przeprowadza za pomocą funkcji **getIconWidth** i **getIconHeight**.

Ekran
Wykreślenie obrazu




```
public
class ContentPane
    extends Content {

    private String fileName = "Duke.gif";

    private Image image;

    public void initPane()
    {
            // pobranie obrazu
        ImageIcon icon = new ImageIcon(fileName);
        image = icon.getImage();

            // czy obraz istnieje
        int w = icon.getIconWidth(),
            h = icon.getIconHeight();
```

```
        if (w == -1 || h == -1) {
            System.out.println("Image does not exist");
            System.exit(0);
        }
    }

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        gDC.drawImage(image, 0, 0, null);
    }
}
```

Buforowanie

Wykreślanie grafiki może się odbywać nie tylko na pulpicie, ale również w **buforze** pamięci operacyjnej. Po utworzeniu bufora można utworzyć wykreślacz zarządzający buforem i wydawać mu polecenia wykreślania (np. **drawLine**) albo wprost umieszczać piksele w buforze (np. **setRGB**).

Skomplikowany obraz można przygotować w buforze, a następnie wykreślić na pulpicie. Dzięki temu, w komputerach wyposażonych w słabe procesory i wolne karty graficzne unika się ujawnienia anatomii konstruowania obrazu, a w przypadku animacji pozbywa niemiłego dla oczu migotania ekranu.

klasa Component

```
Image createImage(int w, int h)
Tworzy bufor obrazu o rozmiarach w x h. W odnośniku typu Image
dostarcza odniesienie do obiektu klasy BufferedImage reprezentują-
cego obraz.
np. Image img = createImage(200, 200);
    BufferedImage img2 = (BufferedImage)img;
```

klasa Image

```
Graphics getGraphics()
Dostarcza odniesienie do wykreślacza związanego z buforem obrazu.
np. Graphics mDC = img.getGraphics();
```

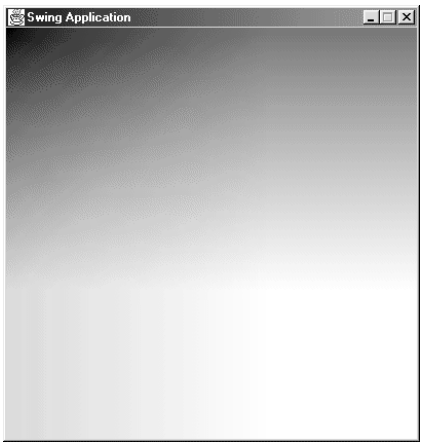
klasa BufferedImage

```
void setRGB(int x, int y, int color)
Nadaje pikselowi (x, y) obrazu kolor color, opisany przez jego
składowe ARGB.
np. img.setRGB(10, 10, 255 << 16 | 255 << 8 | 128);
```

```
int getRGB(int x, int y)
Dostarcza reprezentację koloru piksela w modelu ARGB.
np. int red = (getRGB(100, 100) >> 16) & 0xff;

void setRGB(int x, int y, int w, int h,
            int[] source, int offset, int scan)
Przenosi reprezentacje pikseli z tablicy source do prostokąta o le-
wym-górnym narożniku w (x ,y) i rozmiarach w x h (parametrowi off-
set nadaje się zazwyczaj wartość 0, a parametrowi scan wartość w).
np. img.setRGB(0, 0, w, h, pixels, 0, w);

int[] getRGB(int x, int y, int w, int h,
             int[] target, int offset, int scan)
Przenosi reprezentacje pikseli do tablicy target, pobierając je
z prostokąta o lewym-górnym narożniku w (x ,y) i rozmiarach w x h
(parametrowi offset nadaje się zazwyczaj wartość 0, a parametrowi
scan wartość w). Dostarcza odniesienie do tablicy pikseli (jeśli
target ma wartość null, to tablica jest fabrykowana niejawnie).
np. img.getRGB(0, 0, w, h, pixels, 0, w);
```

Następująca pod-aplikacja, pokazana na ekranie ***Wykreślanie pikseli*** wypełnia pulpit gradientowo zmieniającym się wzorem.

Ekran
Wykreślanie pikseli




```
public
class ContentPane
    extends Content {

    private BufferedImage image;

    public void initPane()
    {
        int w = getWidth(),
            h = getHeight();

        image = (BufferedImage)createImage(w, h);
```

```
        for (int x = 0; x < w ; x++) {
            for (int y = 0; y < h ; y++) {
                int r = Math.min(x, 255),
                    g = Math.min(y, 255),
                    b = Math.min(x+y, 255);
                int color = r << 16 | g << 8 | b;
                image.setRGB(x, y, color);
            }
        }
    }

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        gDC.drawImage(image, 0, 0, null);
    }
}
```

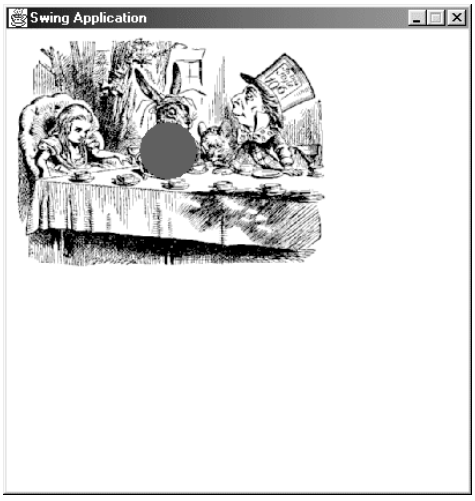
dla dociekliwych

Obraz załadowany za pomocą fabrykatora klasy **ImageIcon** jest reprezentowany w obiekcie klasy **Image**, a nie w obiekcie klasy **BufferedImage**. W celu uzyskania obrazu reprezentowanego w obiekcie klasy **BufferedImage** należy utworzyć bufor w pamięci operacyjnej, a następnie wykreślić w nim obraz.

Następująca pod-aplikacja, pokazana na ekranie ***Obrazy buforowane***, używa funkcji do załadowania obrazu z pliku o podanej nazwie, dostarczającej odniesienie do bufora klasy **BufferedImage** zawierającego ten obraz.

Dzięki temu, że obraz jest buforowany, można wykonywać na nim operacje graficzne.

Ekran
Obrazy buforowane



```
public
class ContentPane
    extends Content
    implements ImageObserver {

    private String fileName = "Hatter.gif";

    private BufferedImage img;

    public void initPane()
    {
        img = getImage(fileName);

        int w = img.getWidth(this),
            h = img.getHeight(this);

        Graphics mDC = img.getGraphics();

        mDC.setColor(Color.red);
        mDC.fillOval((w-50)/2, (h-50)/2, 50, 50);
    }

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        gDC.drawImage(img, 0, 0, null);
    }

    public BufferedImage getImage(String fileName)
    {
        try {
            ImageIcon icon = new ImageIcon(fileName);

            Image image = icon.getImage();

            int w = icon.getIconWidth(),
                h = icon.getIconHeight();

            BufferedImage img =
                (BufferedImage)createImage(w, h);

            Graphics mDC = img.getGraphics();
            mDC.drawImage(image, 0, 0, null);
            return img;
        }
        catch (Exception e) {
            return null;
        }
    }
}
```

Kursory

Z pulpitem można związać ***kursor***, zdefiniowany za pomocą ***przezroczystego*** obrazu *GIF*. Istnieje możliwość utworzenia kursora na podstawie obrazu zawartego w pliku oraz określenie jego ***gorącego punktu*** (np. w przypadku kursora strzałkowego, zakończenia jej grotu).

W celu ustawienia przezroczystości koloru wchodzącego w skład obrazu, można użyć programu ***Paint ShopPro***, wydając w nim polecenie ***Colors / Set Palette Transparency***.

klasa Toolkit

```
Cursor createCustomCursor(
          Image image, Point hotSpot, String name
       )
    throws IndexOutOfBoundsException
```

Dostarcza kursor o takim wyglądzie, jaki ma obraz ***image***. Definiuje piksel ***hotSpot*** jako gorący punkt kursora. Związuje z kursorem nazwę mnemoniczną ***name***.

```
np. Cursor arrow = kit.createCustomCursor(
                       arrow, new Point(10, 10), "Arrow"
                   );
```

klasa Container

```
void setCursor(Cursor cursor)
```

Związuje z pojemnikiem kursor ***cursor***.

```
np. pane.setCursor(arrow);
```

klasa Cursor

```
Cursor getDefaultCursor()
```

Dostarcza systemowy kursor domyślny.

```
np. Cursor defCursor = Cursor.getDefaultCursor();
```

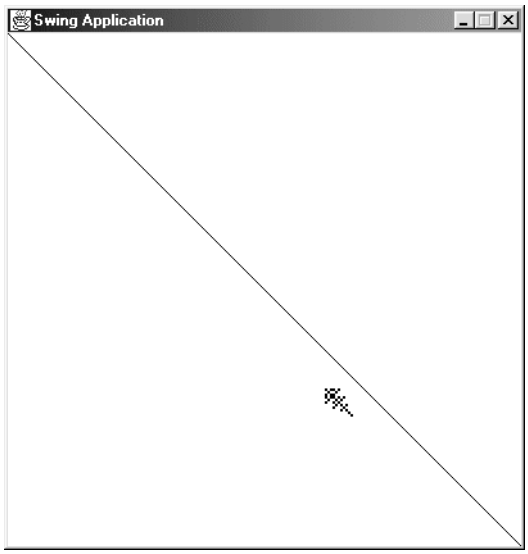
```
Cursor getPredefinedCursor(int id)
```

Dostarcza kursor o identyfikatorze ***id***, m.in. **CROSSHAIR_CURSOR**, **HAND_CURSOR**, **WAIT_CURSOR**, **DEFAULT_CURSOR**.

```
np. Cursor hand =
        Cursor.getPredefinedCursor(Cursor.HAND_CURSOR);
```

Następująca pod-aplikacja, pokazana na ekranie ***Własny kursor***, zastępuje kursor domyślny, kursorem zdefiniowanym w pliku ***Cursor.gif***.

Ekran
Własny kursor

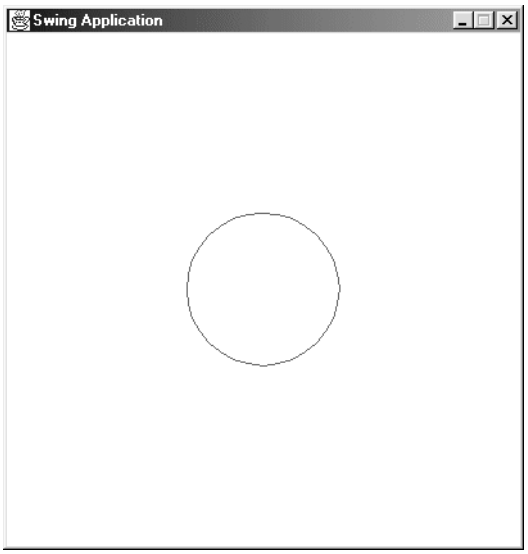



```
public
class ContentPane
    extends Content {

    private String fileName = "Cursor.gif";

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

            // wykreślenie przekątnej
        int w = getWidth(),
            h = getHeight();
        gDC.drawLine(0, 0, w-1, h-1);

            // pobranie kursora
        ImageIcon icon =
            new ImageIcon(fileName);
        Image image = icon.getImage();

            // gorący punkt kursora
        Point hotSpot = new Point(0,0);

            // utworzenie kursora
        Cursor cursor =
            Toolkit.getDefaultToolkit().
                createCustomCursor(image, hotSpot, "");

            // związanie kursora z pulpitem
        setCursor(cursor);
    }
}
```

Zadanie A

Wykreślić okrąg o podanym promieniu, wyśrodkowany na pulpicie

Ekran
Wykreślenie okręgu

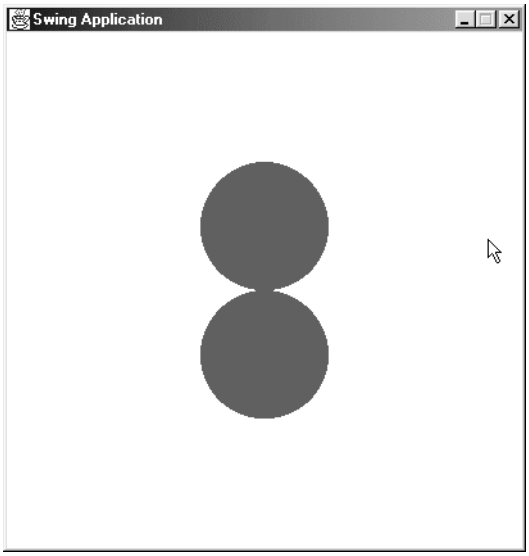



```
public
class ContentPane
    extends Content {

    private int r, d, x, y, w, h;

    public void initPane()
    {
        String string =
            JOptionPane.showInputDialog(
                "Enter radius"
            );

        r = Integer.parseInt(string);
        w = getWidth();
        h = getHeight();
    }

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        d = 2 * r;
        x = (w - d) / 2;
        y = (h - d) / 2;
        gDC.setColor(Color.red);
        gDC.drawOval(x, y, d-1, d-1);
    }
}
```

Zadanie B

Wykreślić stycznie dwa koła o podanym promieniu, położone jedno nad drugim i wyśrodkowane na pulpicie

Ekran
Wykreślenie dwóch kół

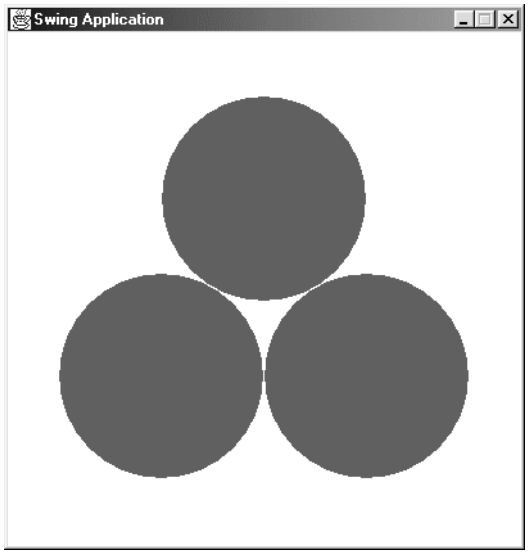



```
public
class ContentPane
    extends Content {

    private int r, w, h;

    public void initPane()
    {
        String string =
            JOptionPane.showInputDialog(
                "Enter radius"
            );

        r = Integer.parseInt(string);
        w = getWidth();
        h = getHeight();
    }

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        int d = 2*r,
            x = (w - d) / 2,
            y1 = (h - 2 * d) / 2,
            y2 = y1 + d;

        gDC.setColor(Color.red);
        gDC.fillOval(x, y1, d, d);
        gDC.fillOval(x, y2, d, d);
    }
}
```

Zadanie C

Wykreślić stycznie trzy koła o podanym promieniu, wyśrodkowane na pulpicie

Ekran
Wykreślenie trzech kół



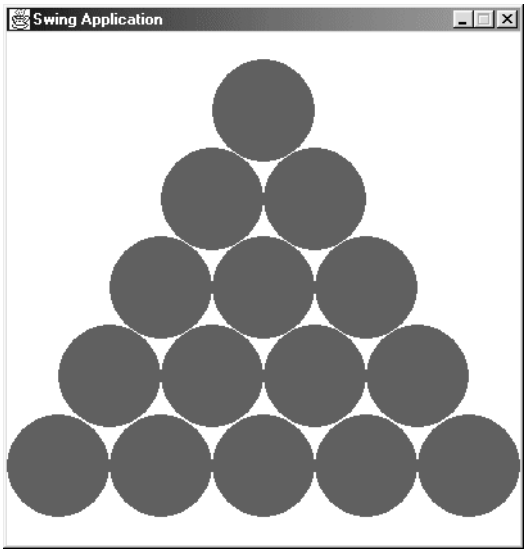

```
public
class ContentPane
    extends Content {

    private int r, w, h;
    private double cos30 = Math.cos(Math.PI / 6);

    public void initPane()
    {
        String string =
            JOptionPane.showInputDialog(
                "Enter radius"
            );

        r = Integer.parseInt(string);
        w = getWidth();
        h = getHeight();
    }

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        int d = 2 * r,
            x = (w - d) / 2,
            y = (int)((h - d * (1 + cos30)) / 2),
            x1 = x - r,
            x2 = x + r,
```

```
            y1 = (int)(y + d * cos30),
            y2 = y1;

        gDC.setColor(Color.red);

        gDC.fillOval(x, y, d, d);
        gDC.fillOval(x1, y1, d, d);
        gDC.fillOval(x2, y2, d, d);
    }
}
```

Zadanie D

Wykreślić piramidę kół, o podanej ich liczbie u podstawy, wyśrodkowaną na pulpicie

Ekran
Wykreślenie piramidy kół




```
public
class ContentPane
    extends Content {

    private int n, r, w, h;
    private double cos30 = Math.cos(
                               Math.PI * 30 / 180
                           );

    public void initPane()
    {
        String string =
            JOptionPane.showInputDialog(
                "Enter count"
            );
```

```
        n = Integer.parseInt(string);
        w = getWidth();
        h = getHeight();
    }

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        r = (int)(h/2 / (1 + (n-1) * cos30));
        if (2 * n * r  > w)
            r = w / (2 * n);

        int d = 2 * r,
            x = (w - d) / 2,
            y = (int)((h - d * (1 + (n-1) * cos30)) / 2);
        for (int i = 0; i < n ; i++) {
            double xi = x - i * r,
                   yi = y + i * d * cos30;

            gDC.setColor(Color.red);
            gDC.fillOval((int)xi, (int)yi, d, d);

            for (int j = 1; j < i+1 ; j++) {
                gDC.setColor(Color.red);
                gDC.fillOval(
                    (int)(xi + j * d), (int)yi, d, d
                );
            }
        }
    }
}
```

Zdarzenia

Program wykonuje się w ***otoczeniu***, które może być źródłem ***zdarzeń***. System rozpoznaje ***zdarzenie*** (np. naciśnięcie przycisku myszki), tworzy opisujący je ***obiekt zdarzeniowy***, identyfikuje ***obiekty nasłuchujące***, które ***oddelegowano*** do obsługi zdarzenia i wydaje im polecenia wyrażone przez ***funkcje obsługi*** (np. **mousePressed**).

Z myszką są związane zdarzenia ***mouse*** (np. naciśnięcie przycisku myszki) i ***mouseMotion*** (np. przeciągnięcie kursora), z klawiaturą zdarzenia ***key*** (np. naciśnięcie klawisza), a z **przyciskami** (**JButton**), **klatkami** (**JTextField**) i **menu** (**JMenuItem**) zdarzenia ***action*** (np. kliknięcie przycisku).

Ponadto istnieją zdarzenia związane z wydawaniem poleceń **komponentom** oblicza aplikacji (oknami, listami, itp.) oraz zdarzenia związane z **odtwarzaniem pulpitu**, wiodące do wywołania funkcji **paint** i **paintComponent**.

Obiekty nasłuchujące

Obiektem nasłuchującym zdarzenia kategorii *Kind* (np. **mouse**, **mouseMotion**, **key**) może być obiekt dowolnej klasy **implementującej** interfejs *Kind***Listener** lub **rozszerzającej** klasę adaptacyjną *Kind***Adapter**.

Jeśli klasa implementuje interfejs, to powinna dostarczyć definicje wszystkich zadeklarowanych w nim funkcji. W przeciwnym razie nie będzie można fabrykować jej obiektów.

W szczególności, w zasięgu deklaracji

```
class MouseWatcher
    extends MouseAdapter {

    // ...

}
```

i po wykonaniu instrukcji

```
MouseWatcher watcher = new MouseWatcher();
```

powstaje obiekt przystosowany do nasłuchiwania zdarzeń ***mouse***.

W tabeli ***Interfejsy nasłuchujące*** wymieniono wybrane interfejsy *Kind***Listener** oraz zadeklarowane w nich funkcje obsługi. Każdemu interfejsowi *Kind***Listener** (z wyjątkiem **ActionListener**) odpowiada klasa adaptacyjna *Kind***Adapter** implementująca interfejs *Kind***Listener**. W klasie adaptacyjnej zdefiniowano domyślne funkcje obsługi. Ciało każdej z nich jest **puste**, a więc domyślną obsługą zdarzenia jest zaniechanie wykonania jakichkolwiek czynności.

Tabela Interfejsy nasłuchujące

Interfejs	*Funkcje obsługi*
ActionListener	actionPerformed
KeyListener	keyPressed keyReleased keyTyped
MouseListener	mousePressed mouseReleased mouseClicked mouseEntered mouseExited
MouseMotionListener	mouseMoved mouseDragged

Ponieważ każda klasa może być pochodną co najwyżej **jednej** klasy, więc jeśli zamierza się użyć funkcji obsługi zdarzeń więcej niż jednej kategorii (np. ***action*** i ***key***), to klasę obsługującą należy zdefiniować jako pochodną klasy adaptacyjnej **implementującej** dodatkowe interfejsy, a to może pociągnąć za sobą konieczność zdefiniowania **nie używanych** funkcji obsługi.

W szczególności, jeśli zamierza się użyć tylko funkcji **mousePressed** (z klasy **MouseAdapter**) i **mouseDragged** (z interfejsu **MouseMotionListener**), to w klasie nasłuchującej należy zdefiniować także **nie używaną** funkcję **mouseMoved** (z interfejsu **MouseMotionListener**).

```
    // klasa obiektów nasłuchujących
    // zdarzenia mouse i mouseMotion
class Watcher
    extends MouseAdapter
    implements MouseMotionListener {

    public void mousePressed(MouseEvent evt)
    {
        // ...
    }

    public void mouseDragged(MouseEvent evt)
    {
         // ...
    }
```

```
        // nie używana funkcja obsługi
    public void mouseMoved(MouseEvent evt)
    {
    }
}
```

Delegowanie nasłuchu

W celu oddelegowania obsługi zdarzeń, należy utworzyć obiekt klasy nasłuchującej, a następnie wydać polecenie

***source*.add*Kind*Listener(*target*)**

w którym ***source*** jest odnośnikiem do obiektu wysyłającego zdarzenia, a ***target*** jest odnośnikiem do obiektu nasłuchującego.

Jeśli ***source*** jest słowem kluczowym **this**, to źródłem zdarzenia jest obiekt tej klasy z której wywołano funkcję **add*Kind*Listener**. Ponieważ w miejscu jej wywołania funkcja ta jest zazwyczaj **widoczna**, więc kwalifikator **this** z reguły **pomija się**.

klasa AbstractButton

```
void addActionListener(ActionListener listener)
Oddelegowuje obiekt listener do nasłuchiwania zdarzeń action.
np. button.addActionListener(lst);
```

klasa Component

```
void addKindListener(KindListener listener)
Oddelegowuje obiekt listener do nasłuchiwania zdarzeń Kind.
np. panel.addMouseMotionListener(lst);
```

W następującej pod-aplikacji utworzono obiekt nasłuchujący klasy **Watcher**, a następnie do obsługiwania zdarzeń ***mouse*** i ***mouseMotion*** pochodzących od pulpitu (reprezentowanego przez zbyteczne tu **this**) oddelegowano ten właśnie obiekt.

```
public
class ContentPane
    extends Content {

    public void initPane()
    {
           // utworzenie obiektu nasłuchującego
        Watcher watcher = new Watcher();

           // oddelegowanie obsługi zdarzeń
           // (zbędna kwalifikacja przez this)
        this.addMouseListener(watcher);
```

```
        this.addMouseMotionListener(watcher);
        // ...
    }

    class Watcher
        extends MouseAdapter
        implements MouseMotionListener {

        // ... funkcje obsługi
    }

    // ...
}
```

Funkcje obsługi

Funkcje obsługi zdarzeń definiuje się w klasie obiektów nasłuchujących. Jeśli klasa obiektów nasłuchujących jest **pochodną** klasy adaptacyjnej, to wystarczy zdefiniować tylko **niektóre** jej funkcje. Jeśli **implementuje** interfejs, to należy zdefiniować **wszystkie** jego funkcje (bo w przeciwnym razie będzie klasą abstrakcyjną).

Funkcje obsługi **zdarzeń** związanych z myszką (np. **mouseMoved**) są funkcjami jedno-parametrowymi, z parametrem typu **MouseEvent**. Polecenia **getX** i **getY**, wydane obiektowi identyfikowanemu przez ten parametr, dostarczają współrzędne punktu, w którym zaszło zdarzenie.

interfejs ActionListener

```
void actionPerformed(ActionEvent evt)
Wykonano akcję na komponencie (np. naciśnięto przycisk).
```

interfejs MouseMotionListener

```
void mouseMoved(MouseEvent evt)
Przemieszczono kursor, nie naciskając przycisku myszki (przesunięto kursor).

void mouseDragged(MouseEvent evt)
Przemieszczono kursor przy naciśniętym przycisku myszki (przeciągnięto kursor).
```

Jeśli **przeciąganie** zaczęło się **w obszarze** komponentu, to nawet w przypadku gdy kursor przemieści się **poza** komponent, do obiektu nasłuchującego będą dostarczane współrzędne liczone względem lewego-górnego narożnika komponentu.

interfejs MouseListener

`void` **`mousePressed`**`(MouseEvent evt)`
`Naciśnięto przycisk myszki.`

`void` **`mouseReleased`**`(MouseEvent evt)`
`Zwolniono przycisk myszki.`

`void` **`mouseClicked`**`(MouseEvent evt)`
`Naciśnięto, a następnie` **`nie przemieszczając kursora`**`, zwolniono przycisk myszki.`

`void` **`mouseEntered`**`(MouseEvent evt)`
`Przemieszczono kursor w obszar komponentu (np. pulpitu). Udostępnione współrzędne dotyczą zazwyczaj punktu` **`w obszarze`** `komponentu, a nie na jego obrzeżu.`

`void` **`mouseExited`**`(MouseEvent evt)`
`Przemieszczono kursor poza obszar komponentu (np. pulpitu). Udostępnione współrzędne dotyczą zazwyczaj punktu` **`poza obszarem`** `komponentu, a nie na jego obrzeżu.`

Jeśli przeciąganie zaczęło się **poza obszarem** komponentu, to po przejściu kursora przez jego **obrzeże** zostanie wywołana funkcja **mouseEntered**. Nie będzie natomiast wywołana funkcja **mouseDragged** ani funkcja **mouseReleased**, chyba że obiekt nasłuchujący zostanie **dodatkowo** oddelegowany do nasłuchiwania zdarzeń, które zachodzą **poza** komponentem.

interfejs KeyListener

`void` **`keyPressed`**`(KeyEvent evt)`
`Naciśnięto klawisz klawiatury.`

`void` **`keyReleased`**`(KeyEvent evt)`
`Zwolniono klawisz klawiatury.`

`void` **`keyTyped`**`(KeyEvent evt)`
`Naciśnięto, a następnie zwolniono klawisz, któremu odpowiada znak` *`Unikodu`*`.`

Obsługiwanie zdarzeń

W chwili zajścia zdarzenia identyfikuje się wszystkie obiekty nasłuchujące, a następnie **każdemu** z nich wydaje polecenie opisane przez funkcję obsługi.

Parametrem funkcji obsługi jest odnośnik do obiektu zdarzeniowego. Wydawanie poleceń obiektowi zdarzeniowemu umożliwia rozpoznanie ***kontekstu zdarzenia***.

```
public void mouseReleased(MouseEvent evt)
{
        // pobranie współrzędnych kursora
    int x = evt.getX(),
        y = evt.getY();

        // rozpoznanie dwu-kliknięcia wykonanego
        // za pomocą prawego przycisku (isMetaDown)
        // w pobliżu lewego-górnego narożnika pulpitu
    boolean rightButton = evt.isMetaDown(),
            doubleClick = evt.getClickCount() == 2;
    if (x < 50 && y < 50 && rightButton && doubleClick) {
        // ...
    }
}
```

Urządzenia

Poza ***monitorem***, którego fragmentem jest **pulpit**, typowymi urządzeniami zewnętrznymi są: ***myszka***, ***klawiatura*** i ***głośnik***. Wykonanie operacji za pomocą myszki albo klawiatury powoduje zajście **zdarzenia**. Głośnik obsługuje się przez wydawanie **poleceń**.

Obiekty zdarzeniowe

W chwili zajścia zdarzenia jest udostępniany odnośnik do ***obiektu zdarzeniowego*** opisującego zdarzenie. Obiektowi zdarzeniowemu można wydawać polecenia dostarczenia informacji o zdarzeniu.

W funkcji obsługi **myszki** odnośnik do ***obiektu zdarzeniowego*** jest typu **MouseEvent**, a w funkcji obsługi **klawiatury** jest typu **KeyEvent**. Jednak w obu tych przypadkach typ odnośnika wywodzi się pośrednio z klasy **AWTEvent** i bezpośrednio z klasy **InputEvent**. Dlatego zarówno podczas obsługiwania myszki jak i klawiatury można używać wszystkich funkcji tych klas.

klasa AWTEvent

```
Object getSource()
W odnośniku typu Object dostarcza odniesienie do obiektu, który
jest źródłem zdarzenia.
np. Object obj = evt.getSource();

int getID()
Dostarcza liczbowy identyfikator zdarzenia (m.in. MouseEvent.MO-
USE_RELEASED).
np. int id = evt.getID();
```

klasa InputEvent

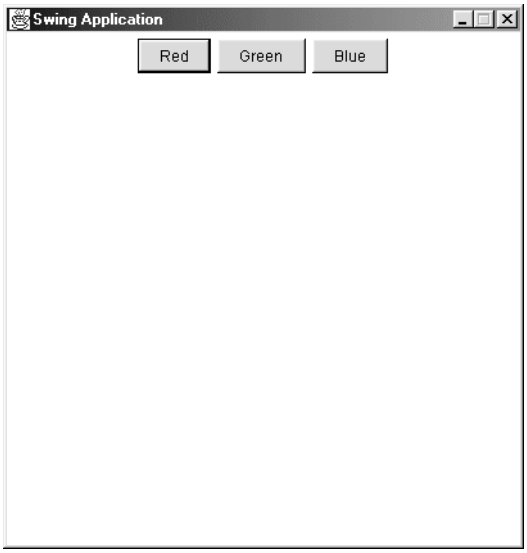
```
int getModifiers()
Dostarcza liczbę, której poszczególne bity opisują stan wciśnięcia
klawiszy Ctrl, Shift, Alt i Meta. Nazwami poszczególnych bitów są
identyfikatory: CTRL_MASK, SHIFT_MASK, ALT_MASK i META_MASK.
np. boolean isCtrlShift =
        evt.getModifiers() &
```

```
            (InputEvent.CTRL_MASK | InputEvent.SHIFT_MASK);

long getWhen()
Dostarcza liczbę milisekund od początku epoki (1 stycznia 1970) do
chwili zajścia zdarzenia.
np. long time = evt.getWhen();

boolean isControlDown()
boolean isShiftDown()
boolean isAltDown()
boolean isMetaDown()
Dostarcza orzeczenie, którego wartość określa stan naciśnięcia kla-
wiszy Ctrl, Shift, Alt i Meta.
np. boolean isCtrl = evt.isControlDown();
```

Użycie funkcji **isMetaDown** umożliwia rozpoznanie, czy naciśnięto **prawy** przycisk myszki. W systemach z myszką **jedno**-przyciskową umożliwia rozpoznanie, że **zasymulowano** naciśnięcie tego przycisku (np. naciskając dodatkowo klawisz *Alt*).

Następująca pod-aplikacja, pokazana na ekranie ***Źródła zdarzeń***, wyświetla przyciski do określania koloru tła.

Ekran
Źródła zdarzeń




```
public
class ContentPane
    extends Content
    implements ActionListener {

    private JButton red   = new JButton("Red"),
                    green = new JButton("Green"),
                    blue  = new JButton("Blue");

    public void initPane()
    {
            // umieszczenie przycisków na pulpicie
        add(red);
```

```
        add(green);
        add(blue);

            // oddelegowanie obsługi
        red.addActionListener(this);
        green.addActionListener(this);
        blue.addActionListener(this);
    }

    public void actionPerformed(ActionEvent evt)
    {
        Object source = evt.getSource();

        if (source == red)
            setBackground(Color.red);
        else if (source == green)
            setBackground(Color.green);
        else
            setBackground(Color.blue);
    }
}
```

Myszka

Wykonanie operacji za pomocą myszki powoduje zajście zdarzeń ***mouse*** i ***mouseMotion***. W funkcji obsługi zdarzenia można użyć funkcji klas **AWTEvent** (m.in. **getSource**) i **InputEvent** (m.in. **isControlDown** i **getWhen**) oraz dodatkowo funkcji klasy **MouseEvent**.

klasa MouseEvent

int **getX**()
Dostarcza ***odciętą*** punktu, w którym zaszło zdarzenie. Współrzędną mierzy się względem **lewego** obrzeża komponentu.
np. `int x = evt.getX();`

int **getY**()
Dostarcza ***rzędną*** punktu, w którym zaszło zdarzenie. Współrzędną mierzy się względem **górnego** obrzeża komponentu.
np. `int y = evt.getY();`

Point **getPoint**()
Dostarcza odniesienie do obiektu opisującego ***punkt***, w którym zaszło zdarzenie. Współrzędne punktu podaje względem lewego-górnego narożnika komponentu.
np. `Point p = evt.getPoint();`

int **getClickCount**()
Dostarcza liczbę kliknięć wielo-kliknięcia, na przykład dla ***dwu--kliknięcia*** dostarcza **2**.
np. `int count = evt.getClickCount();`

```
boolean isPopupTrigger()
Dostarcza orzeczenie o wartości: "wykonano operację, która służy do
wyświetlenia wyskakującego menu".
np. boolean isPopup = evt.isPopupTrigger();

void translatePoint(int dx, int dy)
Dokonuje przemieszczenia współrzędnych punktu, w którym zaszło zda-
rzenie o (dx, dy).
np. evt.translatePoint(20, -10);
```

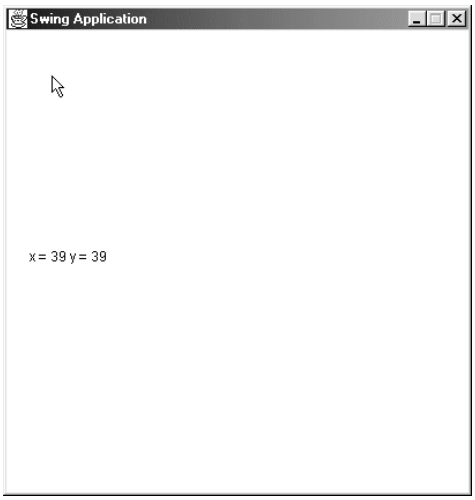
W systemach *Windows* wyświetlanie wyskakującego menu odbywa się po naciśnięciu prawego przycisku myszki. Dlatego funkcja **isPopupTrigger** dostarcza orzeczenie o takiej samej wartości jak funkcja **isMetaDown**.

Przykład A

Współrzędne kursora

Następująca pod-aplikacja, pokazana na ekranie ***Współrzędne kursora***, informuje o aktualnej pozycji kursora.

Ekran
Współrzędne kursora




```
public
class ContentPane
    extends Content {

    private int x, y, h;

    public void initPane()
    {
        h = getHeight();

        addMouseMotionListener(
            new MouseMotionAdapter() {
```

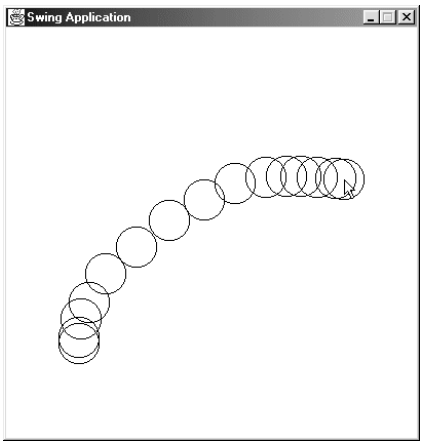
```
                public void
                mouseMoved(MouseEvent evt)
                {
                    x = evt.getX();
                    y = evt.getY();
                    repaint();
                }
            }
        );
    }

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        gDC.drawString(
            "x = " + x +
            " y = " + y,
            20, h/2
        );
    }
}
```

Przykład B

Punkty kontrolne

Następująca pod-aplikacja, pokazana na ekranie ***Punkty kontrolne***, wykreśla małe okręgi w otoczeniu punktów kontrolnych. Ponieważ funkcje **mouseMoved** i **mouseDragged** są wywoływane ze stałą **częstotliwością**, więc liczba punktów kontrolnych zależy od **szybkości** przemieszczania kursora.

Ekran
Punkty kontrolne




```
public
class ContentPane
    extends Content {
```

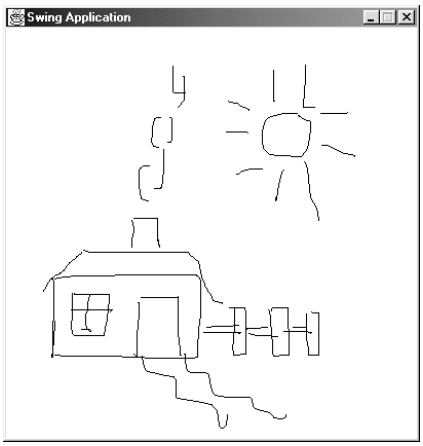
```
    private int r = 20;
    private int x, y;
    private Graphics gDC;

    public void initPane()
    {
        gDC = getGraphics();

        addMouseMotionListener(
            new MouseMotionAdapter() {

                public void
                mouseDragged(MouseEvent evt)
                {
                    int x = evt.getX(),
                        y = evt.getY();
                    gDC.drawOval(
                        x-r, y-r,
                        2*r-1, 2*r-1
                    );
                }
            }
        );
    }
}
```

Przykład C

Wykreślanie odręczne

Następująca pod-aplikacja, pokazana na ekranie ***Wykreślanie odręczne***, wykreśla linię ciągłą wzdłuż drogi przeciągania kursora.

Ekran
Wykreślanie odręczne




```
public
class ContentPane
    extends Content {
    private int xOld, yOld;
```

```
    private Graphics gDC;

    public void initPane()
    {
        gDC = getGraphics();

        Watcher watcher = new Watcher();
        addMouseListener(watcher);
        addMouseMotionListener(watcher);
    }

    class Watcher
        extends MouseAdapter
        implements MouseMotionListener {

        public void mousePressed(MouseEvent evt)
        {
            xOld = evt.getX();
            yOld = evt.getY();
        }

        public void mouseDragged(MouseEvent evt)
        {
            int x = evt.getX(),
                y = evt.getY();

            gDC.drawLine(xOld, yOld, x, y);

            xOld = x;
            yOld = y;
        }

        public void mouseMoved(MouseEvent evt)
        {
        }
    }
}
```

Klawiatura

Wykonanie operacji za pomocą klawiatury powoduje zajście zdarzenia ***key***. W jego funkcji obsługi można użyć funkcji klas **AWTEvent** (m.in. **getSource**) i **InputEvent** (m.in. **isControlDown** i **getWhen**) oraz dodatkowo funkcji klasy **KeyEvent**.

klasa KeyEvent

`char` **`getKeyChar`**`()`
Dostarcza *Unikod* naciśniętego klawisza klawiatury. Jeśli naciśnięto klawisz, któremu nie odpowiada znak *Unikodu*, to dostarcza **`KeyEvent.CHAR_UNDEFINED`** (o wartości **`65535`**).
np. `char chr = evt.getKeyChar();`

```
int getKeyCode()
Dostarcza kod wirtualny naciśniętego klawisza klawiatury.
np. int code = evt.getKeyCode();

boolean isActionKey()
Dostarcza orzeczenie o wartości: "naciśnięto klawisz sterujący:
m.in. F1..F12, Del, Enter, Home, End".
np. boolean isAction = evt.isActionKey();
```

W tabeli ***Kody wirtualne*** przytoczono oznaczenia symboliczne wybranych kodów wirtualnych klawiszy.

Wirtualny kod klawisza (taki sam dla **q** i **Q** oraz **5** i %) identyfikuje **klawisz**, a nie napis na klawiszu. W tabeli ***Kody wirtualne*** przytoczono nazwy kodów wybranych klawiszy (zdefiniowane w klasie **KeyEvent**).

Tabela Kody wirtualne

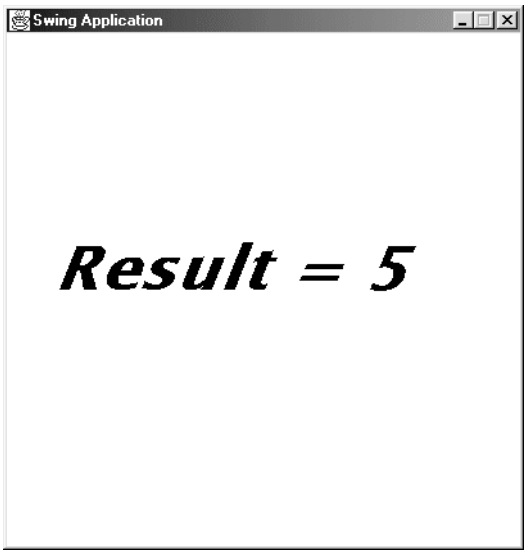
Klawisz	*Nazwa kodu*
A .. Z	VK_A .. VK_Z
F*n*	VK_F*n*
Rt	VK_RIGHT
Lt	VK_LEFT
Up	VK_UP
Dn	VK_DOWN
Enter	VK_ENTER
Space	VK_SPACE
Del	VK_DELETE
Ins	VK_INSERT
Shift	VK_SHIFT
Ctrl	VK_CONTROL
Alt	VK_ALT

Przykład A

Pomiary szybkości

Następująca pod-aplikacja, pokazana na ekranie ***Pomiary szybkości***, wyznacza **szybkość** wprowadzania znaków z klawiatury. Każdy pomiar zaczyna się w chwili naciśnięcia **spacji** i kończy w chwili naciśnięcia klawisza *Enter*.

Ekran
Pomiary szybkości




```
public
class ContentPane
    extends Content {

    private Font font =
        new Font("Lucida Sans", Font.BOLD | Font.ITALIC, 48);
    private String result = "Start typing";

    public void initPane()
    {
        addKeyListener(
            new KeyAdapter() {

                long startTime;
                int count;

                public void
                keyReleased(KeyEvent evt)
                {
                    int code = evt.getKeyCode();
                    if (startTime == 0 &&
                        code == KeyEvent.VK_SPACE) {
                        startTime = evt.getWhen();
                        count = 0;
                    } else if (startTime != 0 &&
                        code == KeyEvent.VK_ENTER) {
                        long stopTime = evt.getWhen(),
                             elapsedTime =
                                 stopTime - startTime;
                        startTime = 0;
                        double s;
                        System.out.println(
                            s = 1000.0 * count / elapsedTime
                        );
                        result = " Result = " + (int)(s + 0.5);
                        repaint();
                    } else
```

```
                        count++;
                    }
                }
            );
        }

        public void paintComponent(Graphics gDC)
        {
            super.paintComponent(gDC);

            gDC.setFont(font);
            gDC.drawString(result, 20, getHeight()/2);
        }
}
```

Przykład B

Kody klawiszy

Następująca pod-aplikacja, pokazana na ekranie ***Kody klawiszy***, podaje *Kody Wirtualne* i *Unikody* naciśniętych klawiszy. Rozpoznaje także stan klawiszy *Shift* i *Ctrl*.

Ekran
Kody klawiszy


```
public
class ContentPane
    extends Content {

    private int h, virCode;
    private Graphics gDC;
    private boolean isRepeating;

    public void initPane()
    {
        h = getHeight();
        gDC = getGraphics();
```

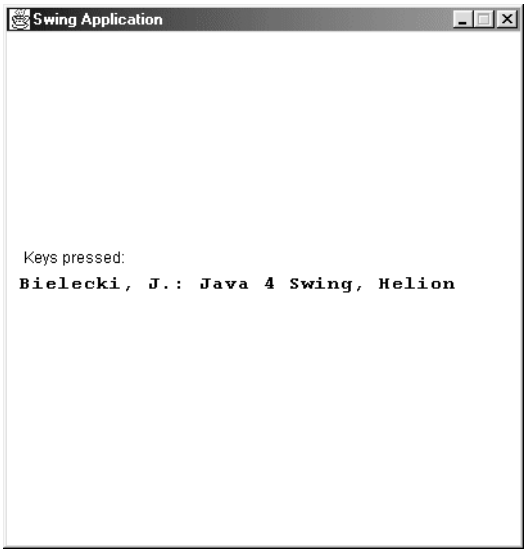
```
        Watcher watcher = new Watcher();
        addKeyListener(watcher);
    }

    class Watcher
        extends KeyAdapter {

        public void keyPressed(KeyEvent evt)
        {
            char keyChar = evt.getKeyChar();
            int  keyCode = evt.getKeyCode();
            boolean isUnicode =
                keyChar != KeyEvent.CHAR_UNDEFINED;
            if (!isUnicode)
                drawResult(
                    "Virtual code = " + keyCode
                );
            else
                virCode = keyCode;

            if (!isRepeating) {
                isRepeating = true;
                String msg = "";
                if (evt.isShiftDown())
                    msg += "Shift";
                if (evt.isControlDown())
                    msg += " Ctrl";
                if (!msg.equals(""))
                    System.out.println(msg);
            }
        }

        public void keyTyped(KeyEvent evt)
        {
            char keyChar = evt.getKeyChar();
            drawResult(
                "Virtual code = " + virCode +
                ", Key = " + keyChar
            );
        }

        public void keyReleased(KeyEvent evt)
        {
            isRepeating = false;
            repaint();
        }
    }

    public void drawResult(String string)
    {
        gDC.drawString(string, 20, h/2);
    }
}
```

Przykład C

Wykreślanie znaków

Następująca pod-aplikacja, pokazana na ekranie ***Wykreślanie znaków***, wykreśla znaki wprowadzone z klawiatury.

Ekran
Wykreślanie znaków




```
public
class ContentPane
    extends Content {

    private Font courierFont =
        new Font("Monospaced", Font.BOLD, 15);

    public void initPane()
    {
            // zdefiniowanie obsługi klawiatury
        addKeyListener(
            new KeyAdapter() {

                private int pos, count = 0;

                    // zwolnienie klawisza
                public void
                keyReleased(KeyEvent evt)
                {
                        // pobranie znaku
                    char key = evt.getKeyChar();

                        // po naciśnięciu klawisza Enter
                        // wyczyszczenie pulpitu
                    if (key == '\n') {
```

```
                        count = 0;
                        repaint();
                    }

                        // utworzenie wykreślacza
                    Graphics gDC = getGraphics();

                        // wstawienie czcionki
                    gDC.setFont(courierFont);

                        // pobranie rozmiarów pulpitu
                    int h = getHeight(),
                        w = getHeight();

                        // określenie pozycji znaku
                    if (evt.getKeyCode() != KeyEvent.VK_SHIFT)
                        count++;
                    pos = 10 * count;
                    if (pos > w - 10)
                        return;

                        // wykreślenie znaku
                    gDC.drawString("" + key, pos, h/2);

                        // zwolnienie wykreślacza
                    gDC.dispose();
                }
            }
        );
    }

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        int h = getHeight();
        gDC.drawString(" Keys pressed: ", 10, h/2-20);
    }
}
```

Głośnik

Nagrania dźwiękowe są zazwyczaj przechowywane w formatach *WAV*, *AU* i *MID*. W celu załadowania dźwięku z pliku, należy utworzyć obiekt reprezentujący ***klip dźwiękowy***, a następnie wydawać mu polecenia włączania i wyłączania dźwięku.

Java nie obsługuje jeszcze popularnego formatu **MP3**. Można oczekiwać, że stanie się tak w niedalekiej przyszłości.

klasa Applet

```
AudioClip newAudioClip(URL clipURL)
Dostarcza odniesienie do obiektu klasy implementującej interfejs
AudioClip, reprezentującego klip dźwiękowy znajdujący się w miejscu
zlokalizowanym przez clipURL.
np. AudioClip clip =
        Applet.newAudioClip(
            new File("c:/WaveClips/Gong.wav").toURL()
        );
```

interfejs AudioClip

```
void play()
Włącza odgrywanie klipu dźwiękowego.
np. clip.play();

void loop()
Włącza cykliczne odgrywanie klipu dźwiękowego.
np. clip.loop();

void stop()
Wyłącza cykliczne odgrywanie klipu dźwiękowego.
np. clip.stop();
```

klasa Toolkit

```
void beep()
Wydaje krótki sygnał dźwiękowy.
np. Toolkit.getDefaultToolkit().beep();
```

Nie należy wywoływać funkcji **beep** jeśli w danej chwili jest odgrywany klip dźwiękowy.

Następująca aplikacja jednocześnie odtwarza klipy dźwiękowe znajdujące się w plikach określonych przez argumenty (np. **Chimes.wav**, **Laser.wav**, **Explode.au**, **Africa.mid**, **Passport.mid**)

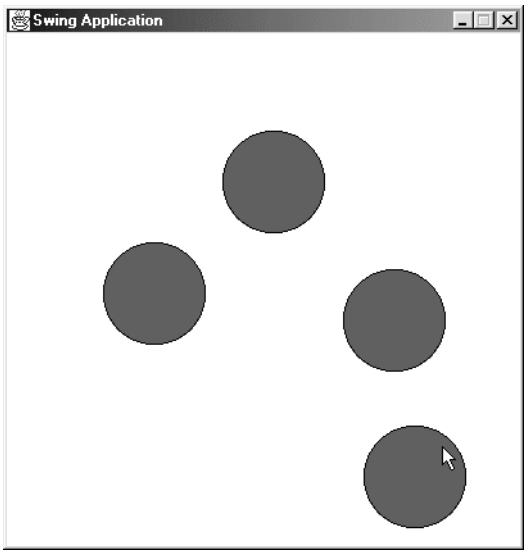
```
package jbPack;

import java.awt.*;
import java.applet.*;
import java.io.*;
import java.net.*;

public
class Program {

    public static void main(String[] args)
    {
        new Program(args);
    }
```

```
    public Program(String[] args)
    {
        int count;
        if ((count = args.length) == 0) {
            Toolkit.getDefaultToolkit().beep();
            System.exit(0);
        }

        for (int i = 0; i < count ; i++) {
            File file = new File(args[i]);
            try {
                URL url = file.toURL();
                AudioClip clip =
                    Applet.newAudioClip(url);
                clip.play();
            }
            catch (MalformedURLException e) {}
        }

            // wstrzymanie wykonania
        try {

            Thread.sleep(2000);
        }
        catch (InterruptedException e) {}
    }
}
```

Zadanie A

Umożliwić wykreślanie kół w otoczeniu punktów kliknięcia

Ekran
Wykreślanie kół

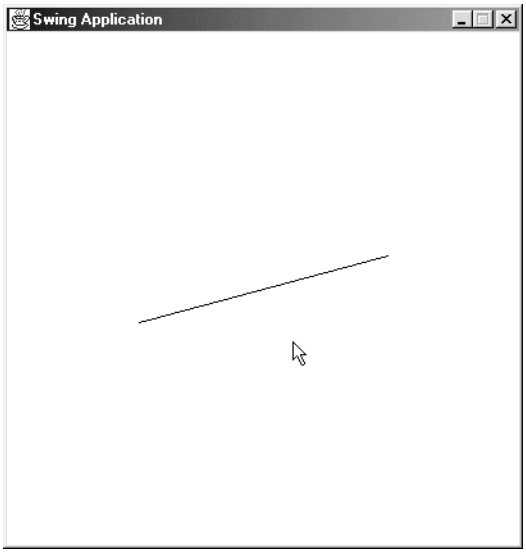


```
public
class ContentPane
    extends Content {

    private Watcher watcher;
    private Graphics gDC;

    public void initPane()
    {
        gDC = getGraphics();
        watcher = new Watcher();
        addMouseListener(watcher);
    }

    class Watcher
        extends MouseAdapter {

        private final int r = 40;
        private int d = 2*r, x = -r, y = -r;

        public void mouseReleased(MouseEvent evt)
        {
            x = evt.getX();
            y = evt.getY();
            drawCircle(gDC);
        }

        public void drawCircle(Graphics gDC)
        {
            gDC.setColor(Color.red);
            gDC.fillOval(x-r, y-r, d, d);
            gDC.setColor(Color.black);
            gDC.drawOval(x-r, y-r, d-1, d-1);
        }
    }

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        watcher.drawCircle(gDC);
    }
}
```

Zadanie B

Podczas przesuwania kursora obracać odcinek o podanej długości

Ekran
Obracanie odcinka



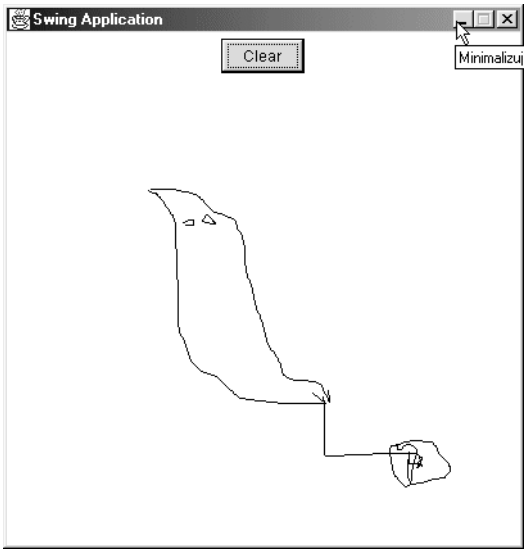

```
public
class ContentPane
    extends Content {

    private int fi = 135,
                dd = 15,
                xA, yA, xZ, yZ,
                w, h, x, y, r;

    public void initPane()
    {
        String string =
            JOptionPane.showInputDialog(
                "Enter length"
            );

        w = getWidth();
        h = getHeight();

        r = Integer.parseInt(string) / 2;

        if (r < 10 || r >= Math.min(w,h)/2)
           System.exit(0);

        Watcher watcher = new Watcher();
        addMouseListener(watcher);
        addMouseMotionListener(watcher);
    }
```

```
    class Watcher
        extends MouseAdapter
        implements MouseMotionListener {

        public void mouseMoved(MouseEvent evt)
        {
            fi += dd;
            if (fi > 360)
                fi -= 360;
            repaint();
        }

        public void mouseExited(MouseEvent evt)
        {
            System.out.println("Mouse left panel");
        }

            // niezbędna, ale nie używana
        public void mouseDragged(MouseEvent evt)
        {
        }
    }

    private double sin(int arg)
    {
        return Math.sin(Math.PI / 180 * arg);
    }

    private double cos(int arg)
    {
        return Math.cos(Math.PI / 180 * arg);
    }

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        x = (int)(r * cos(fi) + 0.5);
        y = (int)(r * sin(fi) + 0.5);
        xA = w/2 + x;
        yA = h/2 - y;
        xZ = w/2 - x;
        yZ = h/2 + y;

        gDC.drawLine(xA, yA, xZ, yZ);
    }
}
```

Zadanie C

Umożliwić wykreślanie linii

Ekran
Czyszczenie pulpitu



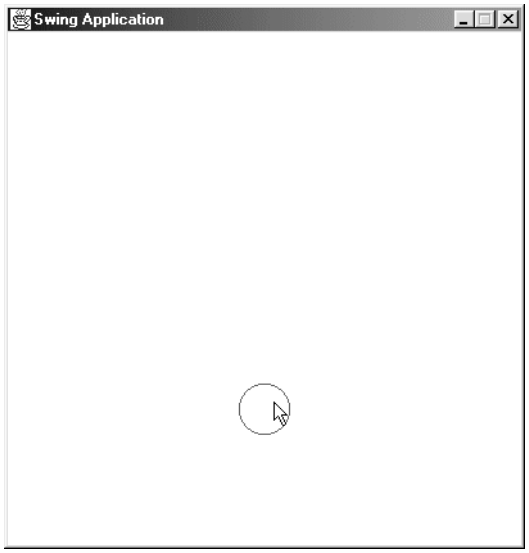

```
public
class ContentPane
    extends Content {

    private JButton clear = new JButton("Clear");
    private Graphics gDC;

    public void initPane()
    {
            // utworzenie wykreślacza
        gDC = getGraphics();

            // umieszczenie przycisku na pulpicie
        add(clear);

            // utworzenie obiektu
            // nasłuchującego zdarzenia
        Watcher watcher = new Watcher();

            // oddelegowanie do obiektu nasłuchującego,
            // zdarzeń pochodzących od przycisku
        clear.addActionListener(watcher);
```

```
            // oddelegowanie do obiektu nasłuchującego,
            // zdarzeń pochodzących od myszki
        addMouseListener(watcher);
        addMouseMotionListener(watcher);
    }

        // klasa obiektów nasłuchujących
    class Watcher
        extends MouseAdapter
        implements ActionListener,
                   MouseMotionListener {

            // obsługa kliknięcia przycisku
        public void actionPerformed(ActionEvent evt)
        {
                // wyczyszczenie pulpitu
            repaint();
        }

        private int x, y;  // współrzędne kursora myszki

            // obsługa naciśnięcia przycisku myszki
        public void mousePressed(MouseEvent evt)
        {
                // pobranie współrzędnych
            x = evt.getX();
            y = evt.getY();
        }

            // obsługa przeciągnięcia kursora myszki
        public void mouseDragged(MouseEvent evt)
        {
                // wykreślenie odcinka łączącego
                // punkt poprzedni z bieżącym
            gDC.drawLine(
                x, y,                              // od
                x = evt.getX(), y = evt.getY()  // do
            );
        }

            // funkcja, która musi wystąpić,
            // mimo iż nie będzie użyta
        public void mouseMoved(MouseEvent evt)
        {
        }
    }
}
```

Zadanie D

Zliczać kliknięcia wykonane w obrębie przemieszczającego się pierścienia

Ekran
Gra zręcznościowa




```
public
class ContentPane
    extends Content {

    private int r = 20,
                d = 2*r, x, y, w, h,
                w2, dy, count = 0;
    private Graphics gDC;
    private boolean gameStopped;

    public void initPane()
    {
        gDC = getGraphics();
        gDC.setColor(Color.red);

        w = getWidth();
        h = getHeight();

        x = w2 = w/2;
        y = h - r;
        dy = getRandom(10) + 1;

        Watcher watcher = new Watcher();
        addMouseListener(watcher);
        addMouseMotionListener(watcher);
```

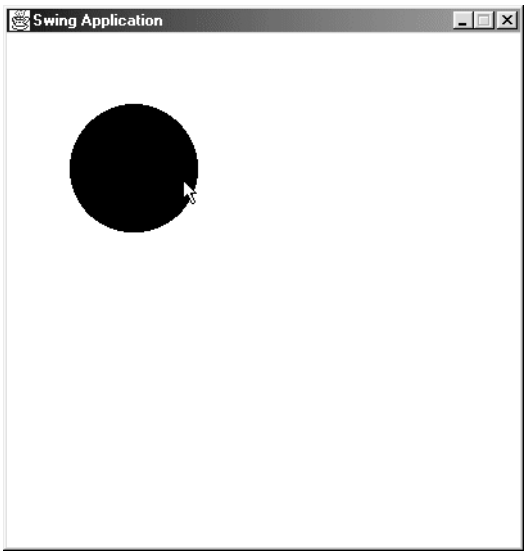
```
        class Runner
            extends Thread {

            public Runner()
            {
                start();
            }

            public void run()
            {
                try {
                    Thread.sleep(10000);
                } catch (InterruptedException e) {}
                gameStopped = true;
            }
        }

        new Runner();
    }

    class Watcher
        extends MouseAdapter
        implements MouseMotionListener {

        public void mouseMoved(MouseEvent evt)
        {
            if (gameStopped)
                return;

                // wyczyszczenie pulpitu
            gDC.clearRect(0, 0, w, h);

            gDC.drawOval(w2-r, y-r, d-1, d-1);
            y -= dy;
            if (y < 0)
                toBottom();
        }

        public void mousePressed(MouseEvent evt)
        {
            int xx = evt.getX(),
                yy = evt.getY();

            if (gameStopped)
                System.exit(0);

            if (sqr(xx-x) + sqr(yy-y) < sqr(r-3))
                count++;
            else {
                Toolkit.getDefaultToolkit().beep();
                count -= 2;
            }

            toBottom();
```

```
            System.out.println(count);
        }

        public void mouseDragged(MouseEvent evt)
        {
        }
    }

    public int sqr(int par)
    {
        return par * par;
    }

    public int getRandom(int range)
    {
        double rand = Math.random();
        return (int)(rand * range);
    }

    public void toBottom()
    {
        y = h - r;
        dy = getRandom(10) + 1;
        gDC.clearRect(0, 0, w, h);
        gDC.drawOval(w2-r, y-r, d-1, d-1);
    }
}
```

Zadanie E

Umożliwić przeciąganie koła po pulpicie

Ekran
Przeciąganie koła



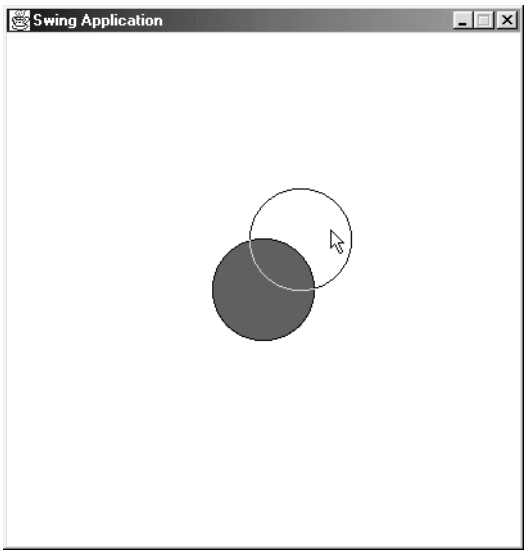
```
public
class ContentPane
    extends Content {

    private int r = 50;
    private int xCor, yCor;
    private int xOld, yOld;
    private Watcher watcher = new Watcher();
    private boolean isInside;

    public void initPane()
    {
        int w = getWidth(),
            h = getHeight();
        xCor = (w - 2 * r) / 2;
        yCor = (h - 2 * r) / 2;

        addMouseListener(watcher);
        addMouseMotionListener(watcher);
    }

    class Watcher
        extends MouseAdapter
        implements MouseMotionListener {

        public void mousePressed(MouseEvent evt)
        {
            int x = evt.getX(),
                y = evt.getY();
            if (isInside(x, y)) {
                isInside = true;
                xOld = x;
                yOld = y;
            }
        }

        public void mouseDragged(MouseEvent evt)
        {
            if (!isInside)
                return;
            int x = evt.getX(),
                y = evt.getY(),
                dx = x - xOld,
                dy = y - yOld;
            xCor += dx;
            yCor += dy;
            xOld = x;
            yOld = y;
            repaint();
        }

        public void mouseMoved(MouseEvent evt)
        {
        }
```

```
        public void mouseReleased(MouseEvent evt)
        {
            isInside = false;
        }
    }

    public boolean isInside(int x, int y)
    {
        int dx = xCor+r - x,
            dy = yCor+r - y;
        return dx * dx + dy * dy < r * r;
    }

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        gDC.fillOval(xCor, yCor, 2*r, 2*r);
    }
}
```

Zadanie F

Umożliwić przeciąganie obręczy koła

Ekran
Przeciąganie obręczy




```
public
class ContentPane
    extends Content {

    private Watcher watcher;
    private int w, h;
    private Graphics gDC;
```

```
public void initPane()
{
    RepaintManager.currentManager(this).
        setDoubleBufferingEnabled(false);

    w = getWidth();
    h = getHeight();

    watcher = new Watcher(w, h);
    addMouseListener(watcher);
    addMouseMotionListener(watcher);

    gDC = getGraphics();
}

class Watcher
    extends MouseAdapter
    implements MouseMotionListener {

    private final int r = 40;
    private int d = 2*r, x, y, xBase, yBase,
                xOld, yOld, xDrag, yDrag;
    private boolean isInside, isDragged;

    public Watcher(int w, int h)
    {
        x = xOld = w/2;
        y = yOld = h/2;
    }

    public void mousePressed(MouseEvent evt)
    {
        int x = evt.getX(),
            y = evt.getY();
        isInside = isInside(x, y);
        if (isInside) {
            setDragged(true);
            drawCircle(
               gDC,
               xBase = xOld = this.x,
               yBase = yOld = this.y
            );
            xDrag = x;
            yDrag = y;
        }
    }

    public void mouseDragged(MouseEvent evt)
    {
        if (isDragged) {
            drawCircle(gDC, xOld, yOld);
            int dx = evt.getX() - xDrag,
                dy = evt.getY() - yDrag;
            drawCircle(gDC, xOld += dx, yOld += dy);
            xDrag += dx;
```

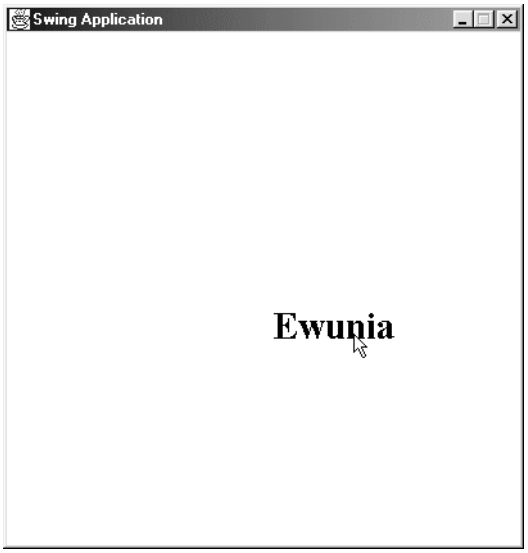
```
            yDrag += dy;
            this.x += dx;
            this.y += dy;
        }
    }

    public void mouseReleased(MouseEvent evt)
    {
        if (isDragged) {
            setDragged(false);
            clearBaseCircle();
        }
        drawCircle(gDC, x, y);
    }

    public void mouseMoved(MouseEvent evt)
    {
    }

    public boolean isInside(int x, int y)
    {
        int dx = this.x - x,
            dy = this.y - y;
        return dx*dx + dy*dy < r*r;
    }

    public void setDragged(boolean dragged)
    {
        if (isDragged = dragged)
            gDC.setXORMode(Color.white);
        else
            gDC.setPaintMode();
    }

    public void clearBaseCircle()
    {
        gDC.clearRect(xBase-r, yBase-r, d, d);
    }

    public void drawCircle(Graphics gDC, int x, int y)
    {
        if (!isDragged) {
            gDC.setColor(Color.red);
            gDC.fillOval(x-r, y-r, d, d);
        }

        gDC.setColor(Color.black);
        gDC.drawOval(x-r, y-r, d-1, d-1);
    }

    public void drawCircle(Graphics gDC)
    {
        drawCircle(gDC, x, y);
    }
}
```

```
    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        watcher.drawCircle(gDC);
    }
}
```

Zadanie G

Umożliwić przeciąganie napisu

Ekran
Przeciąganie napisu




```
public
class ContentPane
    extends Content {

    private String dragMe;
    private int width, height,
                w, h, a, d, x = 50, y = 50;
    private boolean stringPressed = false;
    private int xOld, yOld;
    private Graphics gDC, mDC;
    private Image buffer;
    private Font font =
        new Font("Serif", Font.BOLD, 30);

    public void initPane()
    {
        dragMe = JOptionPane.showInputDialog(
                    "Enter name"
                );
```

```
        if (dragMe == null || dragMe.equals(""))
            System.exit(0);

        gDC = getGraphics();

        FontMetrics mtx =
            gDC.getFontMetrics(font);

        a = mtx.getAscent();
        d = mtx.getDescent();
        h = a + d;
        w = mtx.stringWidth(dragMe);

        buffer = createImage(
                  width  = getWidth(),
                  height = getHeight()
              );

        mDC = buffer.getGraphics();
        mDC.setFont(font);

        Watcher watcher = new Watcher();

        addMouseListener(watcher);
        addMouseMotionListener(watcher);
    }

    class Watcher
        extends MouseAdapter
        implements MouseMotionListener {

        public void mousePressed(MouseEvent evt)
        {
            int xx = evt.getX(),
                yy = evt.getY();
            if (xx > x && yy > y &&
               xx < x+w && yy < y+h) {
                stringPressed  = true;
                xOld = xx;
                yOld = yy;
            } else
                beep();
        }

        public void mouseDragged(MouseEvent evt)
        {
            int xx = evt.getX(),
                yy = evt.getY();
            if (stringPressed) {
                x += xx - xOld;
                y += yy - yOld;
                mDC.clearRect(0, 0, width, height);
                mDC.drawString(dragMe, x, y+a);
                gDC.drawImage(buffer, 0, 0, null);
                xOld = xx;
```

```
                yOld = yy;
            }
        }

        public void mouseReleased(MouseEvent evt)
        {
            stringPressed = false;
        }

        public void mouseMoved(MouseEvent evt)
        {
        }
    }

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        gDC.setFont(font);
        gDC.drawString(dragMe, x, y+a);
    }
}
```

Odtwarzanie pulpitu

Odtwarzanie pulpitu odbywa się w ***funkcji odtwarzającej*** (**paintComponent**).

Funkcja odtwarzająca jest wywoływana przez *System* tuż po wyświetleniu pulpitu oraz wkrótce po tym, jak *System* rozpozna, że nastąpiło zniszczenie fragmentu pulpitu. Tuż przed wywołaniem funkcji odtwarzającej czyści się uszkodzoną część pulpitu ***kolorem tła*** oraz tworzy *Wykreślacz Systemowy* zarządzający pulpitem.

Wykonanie funkcji odtwarzającej odbywa się w *Wątku Zdarzeniowym*. Odtwarzanie umieszczonych na pulpicie komponentów klas wywodzących się od **JComponent** (w tym **JLabel** i **JButton**) odbywa się **automatycznie**, ale tylko wówczas, gdy ciało funkcji odtwarzającej zaczyna się od wywołania **super.paintComponent**(*gDC*).

```
public void paintComponent(Graphics gDC)
{
    super.paintComponent(gDC);

    // pozostałe instrukcje
}
```

Własny wykreślacz

Jeśli wyłączono buforowanie, to wykreślanie odbywa się wprost na pulpicie. Jeśli je włączono, to odbywa się w ***buforze systemowym***, a po zakończeniu wykonywania funkcji odtwarzającej kopiuje się bufor na pulpit. Dlatego, jeśli do sporządzenia wykresu na pulpicie użyje się własnego wykreślacza, to po zakończeniu wykonywania funkcji odtwarzającej, wykres ten zostanie **zastąpiony** wykresem z bufora.

W następującej pod-aplikacji kliknięcie na pulpicie powoduje wykreślenie koła w otoczeniu punktu kliknięcia. Na skutek wywołania funkcji **repaint** i spowodowanego tym wywołania funkcji **paintComponent**, koło to niemal natychmiast **znika**.

```
public
class ContentPane
    extends Content {

    public void initPane()
    {
        addMouseListener(
```

```
            new MouseAdapter() {
                public void mouseReleased(MouseEvent evt)
                {
                    Graphics pDC = getGraphics();

                    pDC.fillOval(
                        evt.getX()-50, evt.getY()-50,
                        100, 100
                    );

                    repaint();
                }
            }
        );
    }
}
```

Wykreślanie w funkcji odtwarzającej

Najprostszym sposobem uniknięcia komplikacji związanych z własnym wykreślaczem jest zastosowanie wykreślania tylko w funkcji odtwarzającej.

Następująca pod-aplikacja rejestruje informacje o tym co należy wykreślić, pozostawiając wykreślanie funkcji odtwarzającej.

```
public
class ContentPane
    extends Content {

    private int x = -10000, y = -10000;

    public void initPane()
    {
        addMouseListener(
            new MouseAdapter() {
                public void mouseReleased(MouseEvent evt)
                {
                    x = evt.getX();
                    y = evt.getY();

                    repaint();
                }
            }
        );
    }

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        gDC.fillOval(x-50, y-50, 100, 100);
    }
}
```

Odtwarzanie z bazy danych

Jeśli na pulpicie należy wykreślić wiele obiektów, to tradycyjną metodą postępowania jest zapisywanie informacji o obiektach w ***bazie danych***, a następnie odtwarzanie ich w funkcji odtwarzającej.

Następująca pod-aplikacja, wykreślająca koła w otoczeniu punktu kliknięcia, ilustruje typową metodykę posługiwania się własną bazą danych.

```
public
class ContentPane
    extends Content {

    private Vector dataBase = new Vector();

    public void initPane()
    {
        addMouseListener(
            new MouseAdapter() {
                public void mouseReleased(MouseEvent evt)
                {
                    Point point =
                        new Point(
                            evt.getX(),
                            evt.getY()
                        );

                    dataBase.add(point);

                    repaint();
                }
            }
        );
    }

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        int count = dataBase.size();
        for (int i = 0; i < count ; i++) {
            Point point = (Point)dataBase.get(i);
            int x = point.x,
                y = point.y;
            gDC.fillOval(x-50, y-50, 100, 100);
        }
    }
}
```

Zwiększenie wrażliwości

Nadużywanie wywołań funkcji **repaint** i związane z tym wielokrotne wywoływanie funkcji odtwarzającej, może spowodować zmniejszenie ***wrażliwości*** aplikacji na zdarzenia. Aby zwiększyć wrażliwość, można zastosować własny wykreślacz, a funkcję odtwarzającą wykorzystywać tylko w przypadku zniszczenia pulpitu.

```
public
class ContentPane
    extends Content {

    private Vector dataBase = new Vector();
    private Graphics pDC;

    public void initPane()
    {
        pDC = getGraphics();

        addMouseListener(
            new MouseAdapter() {
                public void mouseReleased(MouseEvent evt)
                {
                    final int x, y;

                    Point point =
                        new Point(
                            x = evt.getX(),
                            y = evt.getY()
                        );

                    SwingUtilities.invokeLater(
                        new Runnable() {
                            public void run()
                            {
                                pDC.fillOval(
                                    x-50, y-50, 100, 100
                                );
                            }
                        }
                    );

                    dataBase.add(point);
                }
            }
        );
    }

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        int count = dataBase.size();
        for (int i = 0; i < count ; i++) {
            Point point = (Point)dataBase.get(i);
```

```
            int x = point.x,
                y = point.y;
            gDC.fillOval(x-50, y-50, 100, 100);
        }
    }
}
```

Własne buforowanie

Posługiwania się bazą danych można uniknąć, jeśli się zastosuje ***własne buforowanie***. W takim przypadku wszystkie wykreślone obiekty graficzne przechowuje się w buforze i wykreśla w funkcji odtwarzającej.

```
public
class ContentPane
    extends Content {

    private Image buffer;
    private Graphics mDC, pDC;

    public void initPane()
    {
        RepaintManager.currentManager(this).
            setDoubleBufferingEnabled(false);

        pDC = getGraphics();

        buffer = createImage(getWidth(), getHeight());
        mDC = buffer.getGraphics();

        addMouseListener(
            new MouseAdapter() {
                public void mouseReleased(MouseEvent evt)
                {
                    int x = evt.getX(),
                        y = evt.getY();
                    mDC.fillOval(x-50, y-50, 100, 100);
                    pDC.fillOval(x-50, y-50, 100, 100);
                }
            }
        );
    }

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        if (buffer != null)
            gDC.drawImage(buffer, 0, 0, null);
    }
}
```

Własna klasa

Dwukrotne wykonanie takiej samej operacji graficznej, **w-buforze** i **na-ekranie**, czyni program mniej przejrzystszym i dłuższym. Do tego może się stać źródłem błędów wynikających z nie-identycznego powtórzenia operacji. W celu uniknięcia tych zagrożeń, można się posłużyć ***własną klasą wykreślacza***, definiując w niej wszystkie wymagane funkcje.

```
public
class ContentPane
    extends Content {

    private BufferedImage buffer;
    private Grapher pDC;

    public void initPane()
    {
        RepaintManager.currentManager(this).
            setDoubleBufferingEnabled(false);

        setResizable(true);

        pDC = new Grapher(this);

        addMouseListener(
            new MouseAdapter() {
                public void mouseReleased(MouseEvent evt)
                {
                    int x = evt.getX(),
                        y = evt.getY();
                    pDC.fillOval(x-50, y-50, 100, 100);
                }
            }
        );
    }

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        pDC.drawBuffer(gDC);
    }

    class Grapher {

        private Container container;
        private Graphics pDC;
        private BufferedImage buffer;
        private Graphics mDC;

        public Grapher(Container container)
        {
            this.container = container;
            pDC = container.getGraphics();
```

```
            Toolkit kit = Toolkit.getDefaultToolkit();
            Dimension screen = kit.getScreenSize();
            buffer = (BufferedImage)container.createImage(
                        screen.width, screen.height
                    );
            mDC = buffer.getGraphics();
        }

        public void dispose()
        {
            mDC.dispose();
        }

        public void drawBuffer(Graphics gDC)
        {
            if (buffer != null)
                gDC.drawImage(buffer, 0, 0, null);
        }

        // ==========

            // taką definicję należy przygotować
            // dla każdej użytej funkcji klasy Graphics

        public void fillOval(int x, int y, int w, int h)
        {
            pDC.fillOval(x, y, w, h);
            mDC.fillOval(x, y, w, h);
        }
    }
}
```

Współbieżność

Program jest ***współbieżny***, jeśli jego wykonanie wiąże się z więcej niż jednym ***przepływem sterowania***.

Każdy przepływ sterowania jest realizowany przez odrębny ***wątek***. Początkowo wykonuje się tylko wątek ***główny*** i wątki ***systemowe***. Niezależnie od liczby procesorów, program wielowątkowy zachowuje się tak, jakby każdy przepływ sterowania był realizowany przez odrębny procesor.

Jeśli program współbieżny jest wykonywany w *Systemie* jedno-procesorowym, to wielo-procesorowość jest ***emulowana***. Emulowanie wielo-procesorowości polega na przerywaniu wykonania wątku, w celu przydzielenia procesora innemu wątkowi.

Zasady ***przełączania wątków*** nie są objęte specyfikacją języka.

Priorytety

Z każdym wątkiem jest związany jego ***priorytet***. Priorytet jest liczbą z przedziału [**1**, **10**].

Logika programu nie powinna zależeć od priorytetów wątków, a przydzielenie priorytetów powinno nastąpić dopiero po uruchomieniu programu. Tylko przy takim sposobie postępowania można się zbliżyć do nieosiągalnego w praktyce celu, jakim jest napisanie programu w pełni przenośnego.

Tworzenie wątków

W celu utworzenia wątku należy utworzyć ***obiekt wątku*** klasy **Thread** albo jej klasy pochodnej, a następnie wydać mu polecenie **start**. Spowoduje to utworzenie niezależnego przepływu sterowania przez instrukcje bezparametrowej funkcji **run** interfejsu **Runnable**.

klasa Thread

```
Thread()
Thread(Runnable obj)
Konstruuje obiekt klasy Thread, który jest obiektem wątku realizu-
jącego przepływ sterowania przez instrukcje funkcji run zdefiniowa-
nej w klasie obiektu obj, a jeśli go nie użyto, to w klasie obiektu
wątku.
np. Thread thread = new Thread(this);

void start()
Tworzy wątek realizujący przepływ sterowania przez instrukcje funk-
cji run określonej podczas konstruowania obiektu wątku.
np. thread.start();
```

Użycie klasy Thread

Fabrykuje się obiekt wątku klasy **Thread**, dostarczając jego konstruktorowi odniesienie do obiektu klasy implementującej interfejs **Runnable**, a następnie obiektowi wątku wydaje polecenie **start**.

```
package jbPack;

public
class Program
    implements Runnable {

    public static void main(String[] args)
    {
        new Program();
    }

    public Program()
    {
        // ...
        new Thread(this).start();
        // ...
    }

    public void run()
    {
        // ...
    }
}
```

Użycie klasy pochodnej od Thread

Fabrykuje się obiekt wątku klasy pochodnej od **Thread** (implementującej interfejs **Runnable**), a polecenie **start** wydaje **w konstruktorze** tej klasy.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    public Program()
    {
        // ...
        new Runner();
        // ...
    }
}

class Runner
    extends Thread {

    public Runner()
    {
        start();
    }

    public void run()
    {
        // ...
    }
}
```

Użycie klasy anonimowej

Fabrykuje się obiekt klasy anonimowej pochodnej od **Thread** (implementującej interfejs **Runnable**), a następnie wydaje się mu polecenie **start**.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        // ...
        new Thread() {

            public void run()
            {
                // ...
            }
        }.start();
```

```
        // ...
    }
}
```

Następująca aplikacja tworzy wątki realizujące przepływ sterowania przez **tę samą** funkcję **run**. Zastosowano w nim zalecany sposób tworzenia wątków, wykorzystując własną klasę pochodną od **Thread**.

Wyniki

```
*$#**#*$*#**$#**#*$*#**#$**#*$
*#**#$**#*$*#**#$**#*$*#**#$**
#*$*#**#$**#*$*#**#$**#*$*#**#
$*#$##$#$##$#$##$#$##$#$##$#$#
#$#$##$#$##$#$$$$$$$$$$$$$$$$$
```

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Runner('*', 10);
        new Runner('#', 20);
        new Runner('$', 30);
    }

    private static int count;

    public static synchronized void print(char chr)
    {
        System.out.print(chr);
        if (++count % 30 == 0) {
            System.out.println();
            count = 0;
        }
    }
}

class Runner
    extends Thread {

    private char chr;
    private int sleep;

    public Runner(char chr, int sleep)
    {
        this.chr = chr;
        this.sleep = sleep;
        start();
    }

    public void run()
    {
        for (int i = 0; i < 50 ; i++) {
            Program.print(chr);
            try {
                Thread.sleep(sleep);
            } catch (InterruptedException e) {}
        }
    }
}
```

Wykluczanie dostępu

Program współbieżny musi zapewnić ***wykluczanie*** jednoczesnego dostępu do wspólnego zasobu dowolnej pary wątków korzystających z tego zasobu. Pojęcie zasób dotyczy przy tym nie tylko zmiennej i pliku, ale również dowolnych urządzeń, takich jak **monitor**, **klawiatura** i **głośnik**. Zaniedbanie tego wymagania może spowodować, że stan zasobu stanie się **nieokreślony**.

W celu przekonania się o konieczności wykluczania dostępu do wspólnego zasobu, można rozpatrzyć sytuację, w której rolę wątków pełnią dwaj **kasjerzy** wykonujący operacje na tej samym koncie oszczędnościowym.

Jeśli na wspólnym koncie dwóch osób jest na przykład ***$100***, a każda z nich wpłaca w osobnym okienku ***$20***, to może zaistnieć następująca sytuacja

*Pierwszy kasjer sprawdza konto i odnotowuje jego stan (**$100**), ale zostaje oderwany do telefonu.*

*Drugi kasjer sprawdza konto, odnotowuje jego stan (**$100**), dodaje **$20** i aktualizuje konto (do **$120**).*

*Pierwszy kasjer kończy rozmowę, dodaje **$20** do odnotowanej sumy i aktualizuje bazę (do **$120**).*

W następstwie nie-synchronizowanego dostępu do bazy kont, następuje zwiększenie stanu konta nie o **$40**, ale o **$20**.

Wykluczanie musi dotyczyć także operacji, które wydają się ***atomowe***, jak na przykład **counter++**. Wynika to stąd, że program współbieżny powinien wykonywać się poprawnie nie tylko w komputerze, którego procesor wykonuje operację **++** za pomocą **nieprzerywalnej** instrukcji "*dodaj do pamięci*", ale i takim, który tę operację wykonuje za pomocą **przerywalnej** sekwencji: "*pobierz*", "*dodaj*", "*zapamiętaj*".

Sekcje krytyczne

Wykluczenie współbieżnego wykonania wycinka programu odbywa się za pomocą instrukcji

```
synchronized (lock) {
    block
}
```

w której ***lock*** jest odnośnikiem do obiektu ***synchronizatora***, a ***block*** jest sekwencją instrukcji stanowiącą ***sekcję krytyczną***.

Synchronizatorem może być obiekt **dowolnej** klasy. Zaleca się, aby odnośnik identyfikujący synchronizator był zmienną ustaloną klasy **Object**.

Jeśli pewien wątek wykona instrukcję synchronizującą identyfikująca pewien synchronizator, to każdy inny wątek, którego sterowanie napotka instrukcję synchronizującą identyfikująca **ten sam** synchronizator, zostanie ***zatrzymany*** do zakończenia wykonywania sekcji krytycznej przez pierwszy wątek.

Następującą aplikację napisano w taki sposób, że jeśli utworzy ona **2** wątki, z których jeden będzie realizować przepływ sterowania przez funkcję **fun1**, a drugi przez funkcję **fun2**, to **nigdy** nie dojdzie do sytuacji, kiedy podczas wykonywania operacji **count++** w jednym wątku, mogłoby dojść do wykonania operacji **count++** w drugim.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        // ...
    }

    private int count;
    private final Object countLock = new Object();

    public void fun1()
    {
        // ...

        synchronized(countLock) {
            // ...
            count++;
            // ...
        }

        // ...
    }

    public void fun2()
    {
        // ...

        synchronized(countLock) {
            // ...
            count++;
            // ...
        }

        // ...
    }
}
```

Funkcje synchronizowane

Funkcją ***synchronizowaną*** jest funkcja zadeklarowana ze specyfikatorem **synchronized**.

Jeśli definicja funkcji synchronizowanej ma postać

```
Type synchronized fun(par, par, ... , par)
{
    block
}
```

to jest niejawnie zastępowana definicją

```
Type fun(par, par, ... , par)
{
    synchronized (lock) {
        block
    }
}
```

w której ***lock*** jest dla **sposobu** klasy ***Name*** nazwą unikalnego odnośnika ***Name*.class**, a dla metody jest nazwą odnośnika **this**. A zatem ciało funkcji synchronizowanej jest niejawnie zawarte w sekcji krytycznej.

Jeśli polecenie wywołania synchronizowanej **metody** zostanie współbieżnie wydane dwóm **różnym obiektom**, to ponieważ synchronizatory identyfikowane przez **this** będą różne, więc sekwencja instrukcji ***blok*** nie będzie sekcją krytyczną. Taka sytuacja **nie wystąpi**, jeśli współbieżnie wywoła się synchronizowany **sposób**.

Następującą aplikację napisano w taki sposób, że jeśli utworzy ona **2** wątki, z których jeden będzie realizował przepływ sterowania przez funkcję **fun1**, a drugi przez funkcję **fun2**, to **nigdy** nie dojdzie do sytuacji, kiedy polecenie **System.out.println** mogłoby być wykonane **współbieżnie** przez oba te wątki.

Usunięcie z programu specyfikatora **synchronized** uczyniłoby program **błędnym**, ponieważ umożliwiłoby **jednoczesne** wykonanie polecenia **System.out.println** na **wspólnym zasobie** wątków jakim jest ekran **monitora**.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        // ...
    }

    // ...
```

```
    public void fun1(String par)
    {
        println(par);
    }

    public void fun2(String par)
    {
        println(par);
    }

        // synchronizowany sposób
    public static synchronized void println(String par)
    {
        System.out.println(par);
    }
}
```

Niszczenie wątków

W celu ***zniszczenia*** wątku należy doprowadzić do **zakończenia wykonywania** jego funkcji **run**.

Jeśli wykonywanie wątku może być **wstrzymane** albo gdy wątek może być **uśpiony**, to do wyprowadzenia go z tego stanu należy użyć polecenia **interrupt** wydanego obiektowi wątku.

klasa Thread

```
void interrupt()
W obiekcie wątku, któremu wydano to polecenie, ustawia flagę interrupted na true.
np. thread.interrupt();
```

Jeśli polecenie dotyczy wątku, który jest **wstrzymany** albo **uśpiony**, to po uaktywnieniu wysyła się z miejsca uaktywnienia wyjątek klasy **InterruptedException**.

Tuż przed wysłaniem wyjątku flaga ***interrupted*** jest ustawiana na **false**.

```
boolean isInterrupted()
Dostarcza aktualną wartość flagi interrupted, ale nie zmienia jej.
np. boolean isInterrupted = thread.isInterrupted();

boolean interrupted()
Dostarcza aktualną wartość flagi interrupted, po czym ustawia ją na false.
np. boolean isInterrupted = Thread.interrupted();
```

Następująca aplikacja tworzy wątek, który zamierza zasnąć na **3** s, ale wątek główny przerywa ten sen po **1** s, co doprowadza do zakończenia wykonywania funkcji **run** i zniszczenia wątku.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        Thread sleeper =
            new Thread() {

                public void run()
                {
                    long time = System.currentTimeMillis();
                    try {
                        Thread.sleep(3000);
                    }
                    catch (InterruptedException e) {
                        System.out.println(
                            "Sleeping time = " +
                            (System.currentTimeMillis() - time) +
                            "ms"
                        );
                    }
                }
            };

        sleeper.start();

        try {
            Thread.sleep(1000);
        }
        catch (InterruptedException e) {}

        sleeper.interrupt();
    }
}
```

Wyniki

```
Sleeping time = 1050ms
  // albo zbliżony
```

dla dociekliwych

W przytoczonej aplikacji zapewne się nie zdarzy, że ustawienie flagi ***interrupted*** nastąpi **przed** zaśnięciem wątku. Nie można jednak tego wykluczyć. Dlatego przedstawiono ulepszoną wersję programu.

```
package jbPack;

public
class Program {

    private static int intendedSleep = 3000,
                       allowedSleep  = 1000;
    private static Object sleepLock = new Object();

    public static void main(String[] args)
```

```
    {
        Thread sleeper =
            new Thread() {

                public void run()
                {
                    long time = System.currentTimeMillis();
                    int sleepTime = 0;
                    Loop:
                    while (sleepTime < intendedSleep) {
                        try {
                            synchronized(sleepLock) {
                                if (isInterrupted())
                                    break Loop;
                                Thread.sleep(100);
                                sleepTime += 100;
                            }
                        }
                        catch (InterruptedException e) {
                            break Loop;
                        }
                    }

                    System.out.println(
                        "Sleeping time = " +
                        (System.currentTimeMillis() - time) +
                        "ms"
                    );
                }
            };

        sleeper.start();

        try {
            Thread.sleep(allowedSleep);
        }
        catch (InterruptedException e) {}

        synchronized(sleepLock) {
            sleeper.interrupt();
        }
    }
}
```

Zarządzanie wątkami

Zarządzanie wątkami odbywa się za pomocą funkcji klasy **Thread**. Umożliwiają one identyfikację obiektu wątku, uzależnienie dalszego jego wykonania od stanu innych wątków, uśpienie wątku, ustawienie i rozpoznanie priorytetu oraz przerwanie stanu uśpienia i wstrzymania wątków.

klasa Thread

```
Thread currentThread()
Dostarcza odniesienie do obiektu wątku, z którego wywołano tę funk-
cję.
np. Thread thisThread = Thread.currentThread();

void yield()
Zezwala na przydzielenie procesora innym wątkom. Jeśli wątek, z któ-
rego wydano to polecenie znajduje się w sekcji krytycznej, to
nastąpi zwolnienie sekcji.
np. Thread.yield();

void join()
    throws InterruptedException
Wstrzymuje wykonywanie wywołującego ją wątku do chwili zakończenia
wątku opartego na obiekcie wątku któremu wydano to polecenie. Jeśli
wątek, z którego wydano polecenie znajduje się w sekcji krytycznej,
to nie nastąpi zwolnienie sekcji.
np. thread.join();

boolean isAlive()
Dostarcza orzeczenie o wartości: "istnieje wątek oparty na obiekcie
wątku, któremu wydano to polecenie".
np. boolean isAlive = thread.isAlive();

void sleep(long time)
    throws InterruptedException
Usypia bieżący wątek na time milisekund. Uśpienie można anulować,
wydając obiektowi wątku polecenie interrupt. Spowoduje to wysłanie
z miejsca uśpienia wątku, wyjątku klasy InterruptedException. Jeśli
wątek, z którego wydano polecenie znajduje się w sekcji krytycznej,
to nie nastąpi zwolnienie sekcji.
np. try {
       Thread.sleep(200);
    }
    catch (InterruptedException e) {}

void setPriority(int priority)
Nadaje wątkowi, którego obiektowi wątku wydano to polecenie, prio-
rytet priority o wartości z przedziału [1, 10]. Im ten priorytet ma
większą wartość, tym wątek jest "ważniejszy".
np. thread.setPriority(8);

int getPriority()
Dostarcza priorytet wątku opartego na obiekcie wątku, któremu wyda-
no to polecenie.
np. int priority = thread.getPriority();
```

Następująca aplikacja ilustruje użycie funkcji **yield**. Zastosowanie tej funkcji jest konieczne w *Systemach*, które umożliwiają wątkom **zawłaszczenie** procesora.

```
package jbPack;

public
class Program
    implements Runnable {

    private static char[] chr   = { '*', '#', '$' };
    private static int[]  sleep = { 10, 20, 30 };
    private static int i;

    public static void main(String[] args)
    {
        new Program();
    }

    public Program()
    {
        for (i = 0; i < 3 ; i++) {
            new Thread(this).start();
            synchronized(this) {
                try {
                    wait();
                }
                catch (InterruptedException e) {}
            }
        }
    }

    public void run()
    {
        int id = Program.i;

        synchronized(this) {
            notify();
        }

        char chr  = Program.chr[id];
        int sleep = Program.sleep[id];

        for (int i = 0; i < 50 ; i++) {
            print(Program.chr[id]);

            Thread.yield();

            try {
                Thread.sleep(sleep);
            } catch (InterruptedException e) {}
        }
    }

    private static int count;

    public static synchronized void print(char chr)
    {
        System.out.print(chr);
```

Wyniki

```
*#$*#**$#**$#**#*$**#*$#**#*$*
#**$#**#$*#**$#**#*$*#**$#**#*
$*#**$#**#*$*#**$#**#*$*#**$#*
*#*$#$##$#$##$#$##$#$##$#$##$#
$##$#$##$#$##$$$$$$$$$$$$$$$$
```

```
        if (++count % 30 == 0) {
            System.out.println();
            count = 0;
        }
    }
}
```

Koordynowanie wątków

Koordynowanie ma na celu ***wstrzymanie*** przepływu sterowania wątku do chwili, gdy zostanie on ***uwolniony*** przez inny wątek. Wstrzymywanie wątków odbywa się za pomocą funkcji **wait**, a ich uwalnianie za pomocą funkcji **notify** i **notifyAll**.

klasa Object

```
void wait()
    throws InterruptedException
Wstrzymuje wykonanie bieżącego wątku do chwili, gdy zostanie on
uwolniony przez polecenie notify albo notifyAll albo gdy jego obiektowi
wątku wyda się polecenie interrupt. Bezpośrednio po wstrzymaniu
wątku następuje zwolnienie sekcji krytycznej.
np. lock.wait();
```

Wywołanie funkcji **wait** musi wystąpić w **sekcji krytycznej**. Jeśli występuje w sekcji synchronizatora ***lock***, to musi mieć postać

```
lock.wait()
```

Przykład

*Funkcja **wait** z synchronizatorem identyfikowanym przez **readyLock***

```
synchronized (readyLock) {
    // ...
    try {
        while (!gameReady)
            readyLock.wait();
    }
    catch (InterruptedException e) {
        return;
    }
    // ...
}
```

klasa Object

```
void notify()
Uwalnia i powoduje zatrzymanie przypadkowego wątku wstrzymanego na
synchronizatorze, któremu wydano to polecenie. Po zakończeniu wyko-
nywania sekcji krytycznej związanej z tym synchronizatorem, umożli-
wia to przydzielenie jej jednemu z zatrzymanych na nim wątków.
np. lock.notify();
```

Wywołanie funkcji **notify** musi wystąpić w **sekcji krytycznej**. Jeśli występuje w sekcji synchronizatora ***lock***, to ma postać

```
lock.notify()
```

Przykład

*Funkcja **notify** z synchronizatorem identyfikowanym przez **readyLock***

```
synchronized (readyLock) {
    // ...
    readyLock.notify();
    gameReady = true;
    // ...
}
```

klasa Object

```
void notifyAll()
Uwalnia i powoduje zatrzymanie wszystkich wątków wstrzymanych na
synchronizatorze, któremu wydano to polecenie. Po zakończeniu wyko-
nywania sekcji krytycznej związanej z tym synchronizatorem, umożli-
wia to przydzielenie jej jednemu z zatrzymanych na nim wątków.
```

Wywołanie funkcji **notifyAll** musi wystąpić w **sekcji krytycznej**. Jeśli występuje w sekcji synchronizatora ***lock***, to ma postać

```
lock.notifyAll()
```

Przykład

*Funkcja **notifyAll** z synchronizatorem identyfikowanym przez **gameReady***

```
synchronized (gameReady) {
    // ...
    gameReady.notifyAll();
    // ...
}
```

dla dociekliwych

Jeśli nazwą ***lock*** jest **this**, to kwalifikator **this** występujący przed nazwami funkcji **wait**, **notify** i **notifyAll**, można **pominąć**.

```
synchronized (this) {
    while (!gameReady)
       try {
           wait();          // this.wait();
       }
       catch(InterruptedExceptio e) {}
    notify();               // this.notify();
}
```

Następująca aplikacja ilustruje komunikację **3** wątków, z których jeden jest **producentem**, a **2** pozostałe są **konsumentami**. Producent produkuje liczby całkowite i umieszcza je w **buforze**. Konsumenci ***na wyścigi*** rywalizują o liczby i informują o dokonanej konsumpcji. Dla większej czytelności użyto **2** zmiennych: **bufferIsFull** i **bufferIsEmpty**. Oczywiście wystarczyłaby jedna.

Wyniki

```
P:   1
C1: 1
P:   2
C1: 2
...
P:   999
C2: 999
P:   1000
C2: 1000
Consumer1 finished
     with first Item: 1
Consumer2 finished
     with first Item: 39
```

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    private final int Count = 1000,
                      Empty = -1,
                      Last  = 0;
    private int count = 0;
    private int[] firstItem = new int [2];

    private int theItem = Empty;
    private Object buffer = new Object();
    private boolean bufferIsEmpty = true,
                    bufferIsFull;
    private Producer producer;
    private Consumer consumer1, consumer2;

    public Program()
    {
        consumer1 = new Consumer(1);
        producer  = new Producer();
        consumer2 = new Consumer(2);
    }

    private Object printLock = new Object();
```

```
public void println(String line)
{
    synchronized (printLock) {
        System.out.println(line);
    }
}

    // producent
class Producer
    extends Thread {

    public Producer()
    {
        start();
    }

    public void run()
    {
        for (int i = 1; i <= Count+2 ; i++) {
            synchronized (buffer) {
                while (!bufferIsEmpty) {
                    try {
                        buffer.wait();
                    }
                    catch (InterruptedException e) {
                    }
                }

                    // produkcja
                if (i < Count+1) {
                    int item = i;
                    theItem = item;
                    println(
                        "P:   " + item
                    );
                } else
                    theItem = Last;

                bufferIsFull  = true;
                bufferIsEmpty = false;
                buffer.notify();
            }
        }
    }
}

    // konsumenci
class Consumer
    extends Thread {

    private int id;

    public Consumer(int id)
    {
        this.id = id;
```

```
            start();
        }

        public void run()
        {
            Loop:
            while (true) {
                synchronized (buffer) {
                    while (!bufferIsFull) {
                        try {
                            buffer.wait();
                        }
                        catch (InterruptedException e) {
                        }
                    }

                        // konsumpcja
                    int item = theItem;

                    if (firstItem[id-1] == 0)
                        firstItem[id-1] = item;

                    theItem = Empty;

                    bufferIsEmpty = true;
                    bufferIsFull  = false;
                    buffer.notifyAll();

                    if (item == Last)
                        break;
                    System.out.println(
                        "C" + id + ": " + item
                    );
                    count++;
                }
            }

            println(
                "Consumer" + id + " finished\n" +
                "     with first Item: " + firstItem[id-1]
            );
        }
    }
}
```

Występowanie impasu

Podczas wykonywania źle zaprojektowanego programu współbieżnego może dojść do ***impasu***, to jest do sytuacji, gdy nie istnieje już ani jeden wątek **uaktywniony** albo być może istnieją wątki uaktywnione, ale nie ma istotnego postępu wykonania programu.

Gdyby w klasie konsumenta z poprzedniego programu zmieniono polecenie **notifyAll** na **notify**, to do impasu mogłoby dojść m.in. w ramach następującego scenariusza

1. Producent produkuje liczbę i zostaje **wstrzymany** do czasu uwolnienia go przez konsumenta.
2. Konsument konsumuje liczbę i wydaje polecenie **notify**, które uwalnia drugiego **konsumenta** (a nie **producenta**).
3. Każdy z konsumentów stwierdza, że nie wyprodukowano kolejnej liczby i zostaje **wstrzymany**.

Nie ma uniwersalnej metody unikania impasu. Jedną z najprostszych jest stosowanie defensywnego schematu wykluczania i koordynowania oraz przydzielanie zasobów wątkom zawsze w tej samej kolejności.

Wykluczanie i koordynowanie

Technikę defensywnego wykluczania i koordynowania najlepiej oddaje rozwiązanie problemu ***Producent - Konsument***. W następującym schemacie należy zwrócić uwagę na użycie instrukcji **while**, a nie **if**. Niekiedy można użyć instrukcji **if**, ale na ogół prowadzi to do impasu.

deklaracje

```
private Object buffer = new Object();
private boolean bufferIsFull;
```

wątek producenta

```
while (true) {
    // ...
    synchronized (buffer) {
        // ...
        while (bufferIsFull) {
            try {
                    // wstrzymanie
                buffer.wait();
            }
            catch (InterruptedException e) {
                // ...
            }
            // ...
        }
        // ...
    }

    // ... produkcja

    synchronized (buffer) {
        bufferIsFull = true;
```

```
            // uwolnienie
        buffer.notify();
    }
}
```

wątek konsumenta

```
while (true) {
    // ...
    synchronized (buffer) {
        // ...
        while (!bufferIsFull) {
            try {
                    // wstrzymanie
                buffer.wait();
            }
            catch (InterruptedException e) {
                // ...
            }
            // ...
        }
        // ...
    }

    // ... konsumpcja

    synchronized (buffer) {
        bufferIsFull = false;

            // uwolnienie
        buffer.notify();
    }
}
```

Przydzielanie zasobów

Klasyczną ilustracją właściwego przydzielania zasobów jest rozwiązanie problemu ***posiłku filozofów***:

> ***Grupa filozofów zasiada do spagetti. Na środku stołu znajdują się łyżki i widelce. Każdy filozof sięga po łyżkę i widelec, je trochę spagetti, po czym odkłada oba sztućce i myśli. Jeśli wziął łyżkę, ale zabrakło dla niego widelca, to zwraca łyżkę i czeka.***

Uniknięcie impasu wynika stąd, że **każdy** najpierw sięga po łyżkę, a jeśli nie może wziąć widelca, to ją zwraca. Gdyby filozofowie postępowali inaczej, to mogłoby dojść do sytuacji, kiedy pobrano by **wszystkie** łyżki i widelce, ale nikt nie dostałby **obu** sztućców.

```
package jbPack;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    private final int EatersCount = 4,
                      ToolsCount  = 3,
                      ExitCount   = 10000;

    private int allCount = 0;

    class Eater {
        int id, count;
        public Eater(int id, int count)
        {
            this.id = id;
            this.count = count;
        }
    }

    private Eater[]  eaters  =
        new Eater [EatersCount+1];
    private Thread[] threads =
        new Thread [EatersCount+1];

    private Object toolsLock = new Object(),
                   countLock = new Object(),
                   outLock   = new Object();

    private boolean started, toolsBack;

    public Program()
    {
        for (int i = 1; i <= EatersCount ; i++) {
            eaters[i] = new Eater(i, 0);
            threads[i] = new Runner(eaters[i]);
        }

        while (true) {
            sleep(500);
            synchronized (countLock) {
                if (allCount < ExitCount) {
                    report();
                    try {
                        countLock.wait();
                    }
                    catch (InterruptedException e) {}
                } else
                    break;
            }
        }

        System.out.println("Done!");
    }
```

Wyniki

```
Eater 2 starts
Eater 3 starts
Eater 4 starts
Eater 1 starts
Report:
     5    5    5    5
Report:
     10   11   10   10
Report:
     16   16   16   15
Report:
     21   21   21   21
Report:
     26   27   26   26
```

```
public static void sleep(int timeMillis)
{
    try {
        Thread.sleep(timeMillis);
    }
    catch (InterruptedException e) {}
}

public void report()
{
    synchronized (outLock) {
        System.out.println("Report: ");
        for (int i = 1; i <= EatersCount ; i++)
            System.out.print("\t" + eaters[i].count);
        System.out.println();
    }
}

int noOfSpoons = ToolsCount;

public synchronized boolean getSpoon()
{
    if (noOfSpoons == 0)
        return false;
    return noOfSpoons-- > 0;
}

public int noOfForks = ToolsCount;

public synchronized boolean getFork()
{
    if (noOfForks == 0)
        return false;
    return noOfForks-- > 0;
}

public void putBack(
           boolean hasSpoon, boolean hasFork
       )
{
    synchronized (toolsLock) {
        if (hasSpoon)
            noOfSpoons++;
        if (hasFork)
            noOfForks++;
        toolsLock.notifyAll();
        toolsBack = true;
    }
}

class Runner
    extends Thread {

    private Eater eater;

    public Runner(Eater eater)
    {
        this.eater = eater;
```

```
            start();
        }

        public void run()
        {
            int id = eater.id;

            synchronized (outLock) {
                System.out.println(
                    "Eater " + id + " starts"
                );
            }

            while (true) {
                boolean hasSpoon, hasFork;

                while (true) {
                    synchronized(toolsLock) {
                            // pobiera
                        hasSpoon = getSpoon();
                        hasFork  = getFork();
                    }

                        // jeśli pobrał oba
                    if (hasSpoon && hasFork) {
                            // używa
                        synchronized (countLock) {
                            ++allCount;
                            ++eater.count;
                            countLock.notify();
                        }

                            // odkłada
                        putBack(hasSpoon, hasFork);

                            // odpoczywa przez 100 ms
                        Program.sleep(100);
                    } else {
                        synchronized (toolsLock) {
                            putBack(hasSpoon, hasFork);
                            toolsBack = false;
                            while (!toolsBack) {
                                try {
                                    toolsLock.wait();
                                }
                                catch (InterruptedException e) {}
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Animacje

Animacja polega na wykreślaniu obiektów w kolejnych ***fazach ruchu***. Jeśli szybkość wykreślania obrazów jest dostatecznie duża, to powstaje wrażenie ***ruchu ciągłego***. Wykorzystano to w **kinematografii**, gdzie liczba **24** kadrów/s jest wystarczająca do tego, aby „oszukać" nasz niedoskonały zmysł wzroku.

Aby wykreślanie kadrów mogło się odbywać „automatycznie", aplikację należy wyposażyć w ***wątek animacyjny***. Wątek taki może realizować animowanie ***wykreśleń*** albo animowanie ***kadrów***.

Animowanie wykreśleń

Następująca pod-aplikacja wykreśla okrąg odbijający się od pionowych krawędzi pulpitu.

```
public
class ContentPane
    extends Content {

    private int r = 30, d = 2*r,
                x, y, w, h;
    private Graphics gDC;

    public void initPane()
    {
        gDC = getGraphics();

        w = getWidth();
        h = getHeight();
        x = w/2;
        y = h/2;

            // odpalenie wątku
        new Thread(
            new Runnable() {
                public void run()
                {
                    int dx = 1;
                    while (true) {
                            // wyczyszczenie pulpitu
                        gDC.clearRect(0, 0, w, h);

                            // wykreślenie okręgu
                        gDC.drawOval(x-r, y-r, d-1, d-1);
```

```
                        // uśpienie na 5 ms
                    sleep(5);

                        // zmiana pozycji
                    x += dx;

                        // odbicie od krawędzi
                    if (x > w-r || x < r)
                            // zmiana kierunku ruchu
                        dx = -dx;
                }
            }
        }
    ).start();
}

public void sleep(int timeMillis)
{
    try {
        Thread.sleep(timeMillis);
    }
    catch (InterruptedException e) {}
}
}
```

Animowanie kadrów

Animowanie kadrów polega na rytmicznym wykreślaniu uprzednio przygotowanych obrazów. W skrajnych przypadkach obrazy mogą się znajdować w **wielu** plikach albo w **jednym** pliku.

Wiele obrazów

Następująca pod-aplikacja, pokazana na ekranie ***Animowanie kadrów,*** animuje sekwencję obrazów zapisanych w formacie *GIF*.

Ekran
Animowanie kadrów

```
public
class ContentPane
    extends Content {

    private String prefix = "Duke";
    private int count = 3;

    private Image[] image = new Image [count];
    private int x, y, ww, hh;
    private Graphics gDC;

    public void initPane()
    {
            // utworzenie wykreślacza
        gDC = getGraphics();

        for (int i = 0; i < count ; i++) {

                // pobranie obrazu
            ImageIcon icon =
                new ImageIcon(prefix + (i+1) + ".gif");

            if (i == 0) {
                    // rozmiary obrazu
                ww  = icon.getIconWidth();
                hh  = icon.getIconHeight();

                    // czy obraz istnieje
                if (ww == -1 || hh == -1) {
                    System.out.println(
                        "Image does not exist"
                    );
                    System.exit(0);
                }

                    // rozmiary docelowe
                int w = getWidth(),
                    h = getHeight();

                    // lewy-górny narożnik
                x = (w - ww) / 2;
                y = (h - hh) / 2;
            }

                // utworzenie odnośnika do obrazu
            image[i] = icon.getImage();
        }

        new Thread() {
            public void run()
            {
                while (true) {
                    for (int i = 0; i < count ; i++) {
```

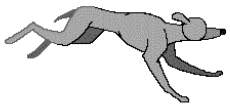
```
                        // wykreślenie
                    gDC.drawImage(
                        image[i], x, y, ww, hh, null
                    );

                        // opóźnienie
                    ContentPane.this.sleep(200);
                }
            }
        }
    }.start();
}

public void sleep(int timeMillis)
{
    try {
        Thread.sleep(timeMillis);
    } catch (InterruptedException e) {}
}
}
```

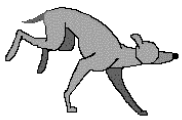
Obrazy zgrupowane

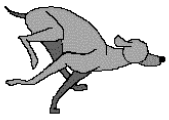
Następująca pod-aplikacja, pokazana na ekranie ***Obrazy zgrupowane,*** animuje sekwencję obrazów zapisanych w pliku w formacie *GIF*, przytoczonym na ekranie ***Obraz wejściowy***.

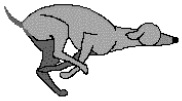
Ekran
Obrazy zgrupowane

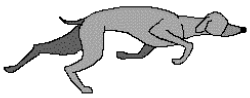
Ekran
Obraz wejściowy

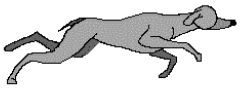


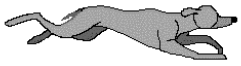












```
public
class ContentPane
    extends Content {

    private String fileName = "Dogs.gif";
    private int count = 7;

    private Image image;
    private int x, y, ww, hh;
    private Graphics gDC;
    private Object start = new Object();
    private boolean isReady;

    public void initPane()
    {
            // utworzenie wykreślacza
        gDC = getGraphics();

            // pobranie obrazu
        ImageIcon icon = new ImageIcon(fileName);

            // rozmiary obrazu
        ww = icon.getIconWidth();
        hh = icon.getIconHeight() / count;

            // czy obraz istnieje
        if (ww == -1 || hh == -1) {
            System.out.println(
                "Image does not exist"
            );
            System.exit(0);
        }

        image = icon.getImage();

            // rozmiary docelowe
        int w = getWidth(),
            h = getHeight();

            // lewy-górny narożnik
        x = (w - ww) / 2;
        y = (h - hh) / 2;

        new Thread() {
            public void run()
            {
                    // wstrzymanie do chwili
                    // wyświetlenia pulpitu
                synchronized (start) {
                    try {
                        while (!isReady)
                            start.wait();
                    }
                   catch (InterruptedException e) {}
                }
```

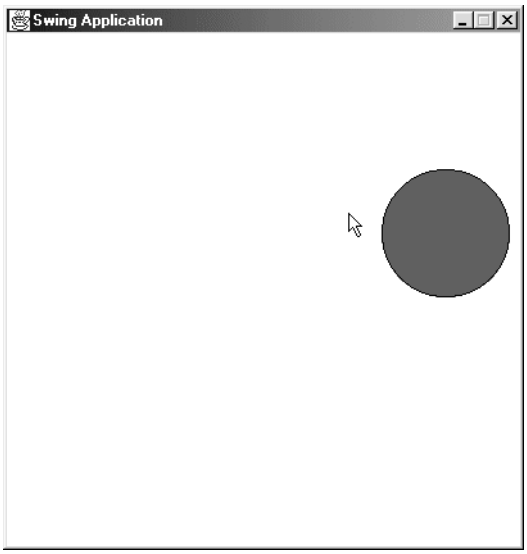
```
                while (true) {

                    for (int i = 0; i < count ; i++) {

                            // wykreślenie
                        gDC.drawImage(
                            image,
                            x, y, x+ww, y+hh,
                            0, i * hh, ww, (i+1) * hh,
                            null
                        );
                            // opóźnienie
                        try {
                            Thread.sleep(100);
                        } catch (InterruptedException e) {}
                    }
                }
            }
        }.start();
    }

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

            // odpalenie animacji
        synchronized (start) {
            isReady = true;
            start.notifyAll();
        }
    }
}
```

Animacje nie-buforowane

Następująca pod-aplikacja, pokazana na ekranie ***Animacja z migotaniem***, animuje koło o zadanym promieniu, odbijające się od krawędzi pulpitu. Ponieważ przed każdym wykreśleniem koła odbywa się czyszczenie pulpitu, więc występuje niemiłe dla oczu ***migotanie*** ekranu.

Bardziej spostrzegawczy zgodzą się, że przytoczona aplikacja narusza wymóg synchronizowania dostępu do wspólnego zasobu różnych wątków. Może się bowiem zdarzyć, że wątek animacyjny przystąpi współbieżnie z wątkiem wykonującym funkcję **paintComponent** do korzystania z ich wspólnego zasobu jakim jest pulpit. Może to spowodować dodatkowe "zamigotanie" na początku wykonywania aplikacji. Właściwe rozwiązanie tego problemu podano w następnym rozdziale.

Ekran
Animacja z migotaniem




```
public
class ContentPane
    extends Content {

    private int r = 50, d = 2 * r,
                dx = 1, dy = -2,
                w, h;
    private Graphics gDC;

    public void initPane()
    {
        w = getWidth();
        h = getHeight();

        gDC = getGraphics();

        new Thread() {
            public void run()
            {
                int x = (w-d)/2,
                    y = (h-d)/2;

                while (true) {
                        // wyczyszczenie pulpitu
                    gDC.clearRect(0, 0, w, h);

                        // wykreślenie koła
                    gDC.setColor(Color.red);
                    gDC.fillOval(x, y, d, d);
                    gDC.setColor(Color.black);
                    gDC.drawOval(x, y, d-1, d-1);

                        // uśpienie na 10 ms
                    ContentPane.this.sleep(10);
```

```
                        // przemieszczenie w poziomie
                    x += dx;
                    if (x > w-d || x < 0)
                        dx = -dx;

                        // przemieszczenie w pionie
                    y += dy;
                    if (y > h-d || y < 0)
                        dy = -dy;
                }
            }
        }.start();
    }

    public void sleep(int timeMillis)
    {
        try {
            Thread.sleep(timeMillis);
        } catch (InterruptedException e) {}
    }
}
```

Animacje buforowane

Następująca pod-aplikacja, pokazana ilustruje sposób wyeliminowania migotania.

Gdyby nie użyto synchronizacji, to wątek wykreślający mógłby wystartować **wcześniej** niż wątek wykonujący funkcję **paintComponent**. Mogłoby to spowodować wyczyszczenie pulpitu już po wykreśleniu koła i w konsekwencji chwilowe „zamigotanie" ekranu.

```
public
class ContentPane
    extends Content {

    private int r = 50, d = 2 * r,
                dx = 1, dy = -2,
                w, h;
    private Graphics gDC;
    private Object paintLock = new Object();
    private boolean painted;

    public void initPane()
    {
        w = getWidth();
        h = getHeight();

        gDC = getGraphics();

        new Thread() {
            public void run()
```

```
        {
            synchronized (paintLock) {
                try {
                    while (!painted)
                        paintLock.wait();
                }
                catch (InterruptedException e) {}
            }

            int x = (w-d) / 2,
                y = (h-d) / 2;

                // utworzenie bufora
            Image img = createImage(w, h);

                // utworzenie wykreślacza bufora
            Graphics mDC = img.getGraphics();

            while (true) {

                    // wyczyszczenie bufora
                mDC.clearRect(0, 0, w, h);

                    // wykreślenie koła
                mDC.setColor(Color.red);
                mDC.fillOval(x, y, d, d);
                mDC.setColor(Color.black);
                mDC.drawOval(x, y, d-1, d-1);

                    // wykreślenie obrazu z bufora
                gDC.drawImage(img, 0, 0, null);

                    // uśpienie na 10 ms
                try {
                    Thread.sleep(10);
                } catch (InterruptedException e) {}

                    // przemieszczenie w poziomie
                x += dx;
                if (x > w-d || x < 0)
                    dx = -dx;

                    // przemieszczenie w pionie
                y += dy;
                if (y > h-d || y < 0)
                    dy = -dy;
            }
        }
    }.start();
}

public void sleep(int timeMillis)
{
    try {
        Thread.sleep(timeMillis);
```

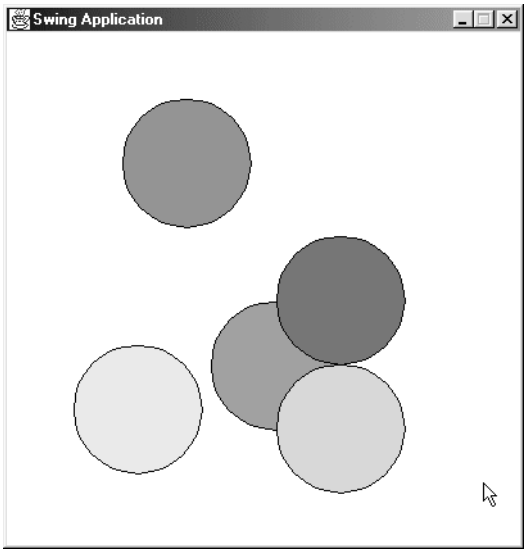
```
            } catch (InterruptedException e) {}
        }

        public void paintComponent(Graphics gDC)
        {
            super.paintComponent(gDC);

            synchronized (paintLock) {
                painted = true;
                paintLock.notify();
            }
        }
}
```

Animacje wielowątkowe

Animacje można realizować w wielu **współbieżnie** wykonywanych wątkach. W takim wypadku poszczególne wątki realizują zazwyczaj ***pętle animacyjne***, a dodatkowemu wątkowi powierza się wykreślanie.

Następująca pod-aplikacja, pokazana na ekranie ***Animacja wielowątkowa***, wyświetla animowane koła, odbijające się od krawędzi pulpitu.

Ekran
Animacja wielowątkowa




```
public
class ContentPane
    extends Content {

    private int Count = 5,
                r = 50, d = 2*r;
    private Image buffer;
    private Graphics gDC, mDC;
```

```
private int w, h;
private Point[] points = new Point[Count];
private Color[] colors = new Color[Count];
private Object pointLock = new Object();
private Random rand = new Random();

public void initPane()
{
    w = getWidth();
    h = getHeight();

    gDC = getGraphics();

    buffer = createImage(w, h);
    mDC = buffer.getGraphics();

    for (int i = 0; i < Count ; i++) {
        colors[i] = getColor();
        new Runner(
            points[i] = new Point(w/2, h/2),
            getDir(), getDir()
        );
    }

    new Painter();  // nie wcześniej!
}

public Color getColor()
{
    Color color;
    int max3, min3, r, g, b;
    do {
        int rnd = rand.nextInt();
        color = new Color(rnd);
        r = color.getRed();
        g = color.getGreen();
        b = color.getBlue();
        max3 = Math.max(Math.max(r, g), b);
        min3 = Math.min(Math.min(r, g), b);
    } while (max3 - min3 < 20);
    return color;
}

public int getDir()
{
    int dir;
    while ((dir = rand.nextInt() % 5) == 0);
    return dir;
}

class Runner
    extends Thread {

    private Point p;
    private int dx, dy;
```

```
        public Runner(Point p, int dx, int dy)
        {
            this.p = p;
            this.dx = dx;
            this.dy = dy;

            start();
        }

        public void run()
        {
            int x = p.x,
                y = p.y;

            while (true) {
                synchronized (pointLock) {
                    p.setLocation(
                        x += dx, y += dy
                    );

                    if (x < r || x > w-1-r)
                        dx = -dx;

                    if (y < r || y > h-1-r)
                        dy = -dy;
                }

                ContentPane.this.sleep(10);
                yield();
            }
        }
    }

    class Painter
        extends Thread {

        public Painter()
        {
            start();
        }

        public void run()
        {
            while (true) {
                synchronized (pointLock) {
                    mDC.clearRect(0, 0, w, h);
                    for (int i = 0; i < Count ; i++) {
                        mDC.setColor(colors[i]);
                        mDC.fillOval(
                            points[i].x - r,
                            points[i].y - r,
                            d, d
                        );

                        mDC.setColor(Color.black);
```

```
                        mDC.drawOval(
                            points[i].x - r -1,
                            points[i].y - r -1,
                            d, d
                        );
                    }

                    gDC.drawImage(buffer, 0, 0, null);

                    ContentPane.this.sleep(10);
                }

                yield();
            }
        }
    }

    public void sleep(int timeMillis)
    {
        try {
            Thread.sleep(timeMillis);
        }
        catch (InterruptedException e) {}
    }
}
```

Data i czas

Czas można mierzyć w skali bezwzględnej albo odliczać w milisekundach od początku ***epoki*** (**1 stycznia 1970**).

Do określenia daty i czasu służą funkcje klasy **Calendar**, wykorzystujące oznaczenia symboliczne wymienione w tabeli ***Opisy daty i czasu***.

Tabela Opisy daty i czasu

Symbol	***Opis***	
HOUR_OF_DAY	liczba godzin	(od północy)
MINUTE	liczba minut	(od godziny)
SECOND	liczba sekund	(od minuty)
YEAR	rok	(np. **2000**)
DAY_OF_YEAR	numer dnia roku	(1 stycznia: **0**)
MONTH	numer miesiąca roku	(styczeń: **0**)
DAY_OF_WEEK	numer dnia tygodnia	(niedziela: **0**)

klasa Calendar

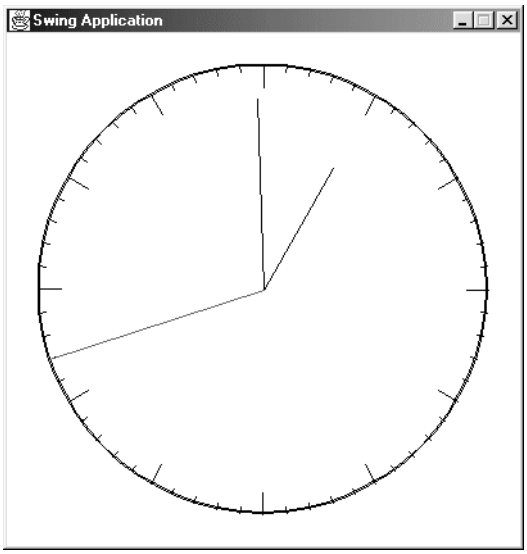
```
Calendar getInstance()
Dostarcza odnośnik do obiektu reprezentującego datę i czas w kalendarzu gregoriańskim, w domyślnej strefie czasowej.
np. Calendar time = Calendar.getInstance();

int get(int info)
Dostarcza informacji o dacie i czasie określonej za pomocą info.
W tabeli Opisy daty i czasu wyszczególniono najważniejsze symbole reprezentujące info.
np. int hours = time.get(Calendar.HOUR_OF_DAY);
```

klasa System

```
long currentTimeMillis()
Dostarcza liczbę milisekund od początku epoki.
np. long millis = System.currentTimeMillis();
```

Następująca pod-aplikacja, pokazana na ekranie ***Zegar analogowy***, ilustruje użycie funkcji do określania daty i czasu.

Ekran
Zegar analogowy




```
public
class ContentPane
    extends Content {

    private double Pi = Math.PI;
    private int x, y;

    public void initPane()
    {
        new Thread(
            new Runnable() {
                public void run()
                {
                    long millis = System.currentTimeMillis();
                    while (true) {
                        long nowMillis =
                            System.currentTimeMillis();
                        long s = (nowMillis - millis) / 1000;

                        System.out.println(
                            "Running " + s + " second" +
                            (s != 1 ? "s" : "")
                        );

                        ContentPane.this.sleep(1000);

                        repaint();
                    }
                }
            }
        ).start();
    }
```

```
void drawLine(Graphics gDC, int xTo, int yTo)
{
    gDC.drawLine(x, y, xTo, yTo);
}

public void sleep(int timeMillis)
{
    try {
        Thread.sleep(timeMillis);
    }
    catch (InterruptedException e) {}
}

public void paintComponent(Graphics gDC)
{
    super.paintComponent(gDC);

    int w = getWidth(),
        h = getHeight();

    int r = 7 * Math.min(w, h) / 16,
        d = 2*r-1;
    x = w / 2;
    y = h / 2;

        // tarcza zegara
    gDC.setColor(Color.black);
    gDC.drawOval(x-r, y-r, d-1, d-1);
    gDC.drawOval(x-r-1, y-r-1, d+2-1, d+2-1);

        // oznaczenia godzin
    for (int i = 0; i < 60 ; i++) {
        double alpha = i * 2*Pi / 60;
        int rr = i % 5 == 0 ? r/10 : r/30;
        gDC.drawLine(
            (int)(x + (r-rr) * Math.cos(alpha)),
            (int)(y - (r-rr) * Math.sin(alpha)),
            (int)(x + r * Math.cos(alpha)),
            (int)(y - r * Math.sin(alpha))
        );
    }

        // pobranie czasu
    Calendar time = Calendar.getInstance();
    int hh = time.get(Calendar.HOUR_OF_DAY),
        mm = time.get(Calendar.MINUTE),
        ss = time.get(Calendar.SECOND),
        secs = ss + (mm + hh * 60) * 60;

        // wskazówka godzinowa
    double alpha = 2*Pi/4 - secs / 3600.0 * 2*Pi / 12;
    drawLine(
        gDC,
        (int)(x + r * Math.cos(alpha) * 5/8),
        (int)(y - r * Math.sin(alpha) * 5/8)
   );
```

```
            // wskazówka minutowa
        alpha = 2*Pi/4 - secs % 3600 / 60.0 * 2*Pi / 60;
        drawLine(
            gDC,
            (int)(x + (r-3*r/20) * Math.cos(alpha)),
            (int)(y - (r-3*r/20) * Math.sin(alpha))
        );

            // wskazówka sekundowa
        gDC.setColor(Color.red);
        alpha = 2*Pi/4 - secs % 60 * 2*Pi / 60;
        drawLine(
            gDC,
            (int)(x + r * Math.cos(alpha)),
            (int)(y - r * Math.sin(alpha))
        );

    }
}
```

Oblicze aplikacji

Na oblicze aplikacji składają się komponenty jej okien. Sposób rozmieszczenia komponentów określają ***zarządcy rozkładu***, a sposób reagowania na zdarzenia opisują funkcje zdefiniowane w klasach obiektów nasłuchujących.

Zestaw predefiniowanych **komponentów** (np. **JButton**) i **zarządców** (np. **BorderLayout**) jest dość obszerny. Jednak w celu poznania zasad posługiwania się nimi wystarczy się ograniczyć do rozpatrzenia zaledwie kilku z nich.

Pulpit

Pulpitem jest ten obszar komponentu, na którym można rozmieszczać **komponenty** i wykreślać **obiekty graficzne**.

Lewy-górny narożnik pulpitu ma współrzędne (**0**, **0**). Odcięte pulpitu zwiększają się **w prawo**, a rzędne **w dół**.

Odsłonięcie, uprzednio zasłoniętej, części pulpitu powoduje wywołanie funkcji **paintComponent**, a w przypadku komponentu klasy **JFrame** i **JApplet**, funkcji **paint**.

Zmiana rozmiarów pulpitu może być w pełni rozpoznana tylko przez obsłużenie zdarzenia ***component***.

klasa Component

```
int getWidth()
Dostarcza szerokość komponentu (np. pulpitu).
np. int w = getWidth();

int getHeight()
Dostarcza wysokość komponentu (np. pulpitu).
np. int h = getHeight();

Dimension getSize()
Dostarcza rozmiary komponentu (np. pulpitu).
np. Dimension d = getSize();
```

```
    int w = d.width,
        h = d.height;
```

Następująca pod-aplikacja wykreśla przekątne pulpitu. Dzięki pobraniu rozmiarów w funkcji **paintComponent**, a nie w funkcji **initPane**, pod-aplikacja jest "odporna" na zmianę rozmiarów okna pulpitu (w wypadku gdyby mogło to nastąpić).

```
public
class ContentPane
    extends Content {

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        int w = getWidth(),
            h = getHeight();

        gDC.drawLine(0, 0, w-1, h-1);
        gDC.drawLine(w-1, 0, 0, h-1);
    }
}
```

Komponenty

Komponentem jest element oblicza graficznego reprezentowany przez obiekt klasy pochodnej od klasy **Component**. Komponentowi można ustawić ***położenie***, ***rozmiary***, ***czcionkę***, ***kolory*** oraz ***widoczność***.

W tabeli ***Hierarchia*** pokazano umiejscowienie klasy **Component** w hierarchii klas.

Tabela Hierarchia

```
Object
    Component
        Container
            JComponent
                JLabel
                JButton
                JTextField
                ...
```

Dalej wymieniono tylko funkcje do określania właściwości komponentów. W klasach **Component** i **JComponent** zdefiniowano także funkcje do **pobierania** właściwości (m.in. **getWidth**, **getHeight**, **isVisible**, **isEnabled**, **getPreferredSize**, itp.).

klasa Component

```
void setLocation(int x, int y)
Umieszcza lewy-górny narożnik komponentu w punkcie (x, y).
np. textArea.setLocation(100, 100);

void setSize(int w, int h)
Nadaje komponentowi rozmiary w x h.
np. textArea.setSize(50, 20);

void setBounds(int x, int y, int w, int h)
Umieszcza lewy-górny narożnik komponentu w punkcie (x, y) i nadaje
mu rozmiary w x h.
np. textField.setBounds(100, 100, 50, 20);

void setForeground(Color color)
Ustawia kolor komponentu na color.
np. label.setForeground(Color.blue);

void setBackground(Color color)
Ustawia kolor tła komponentu na color.
np. label.setBackground(Color.red);

void setFont(Font font)
Ustawia czcionkę komponentu na font.
np. label.setFont(new Font("Serif", Font.PLAIN, 50));

void setVisible(boolean visible)
Na podstawie visible określa, czy komponent ma być widoczny.
np. button.setVisible(false);

void setEnabled(boolean enabled)
Jeśli enabled ma wartość true, czyni komponent wrażliwym na akcje
zewnętrzne.
np. button.setEnabled(true);
```

klasa JComponent

```
void setPreferredSize(Dimension dim)
Określa za pomocą dim, optymalne rozmiary komponentu. Jeśli dim ma
wartość null, to stają się nimi rozmiary domyślne (np. dla przyci-
sku 45 x 27)
np. control.setPreferredSize(
        new Dimension(50, 50)
    );
```

Etykiety

Etykietą jest komponent opisany przez klasę **JLabel**. Oblicze prostej etykiety składa się tylko z tekstu.

klasa JLabel

JLabel(String text)
JLabel(String text, int align)
Konstruuje obiekt klasy **JLabel** reprezentujący etykietę z opisem ***text***, wyrównanym w jej obszarze zgodnie z ***align***: **LEFT**, **CENTER**, **RIGHT**, domyślnie **LEFT**.
np. JLabel counter = new JLabel("0", JLabel.RIGHT);

void **setText**(String text)
Zmienia opis etykiety na ***text***.
np. counter.setText("" + i);

String **getText**()
Dostarcza opis etykiety.
np. String caption = counter.getText();

Następująca pod-aplikacja, pokazana na ekranie ***Definiowanie etykiet***, ilustruje wyposażanie oblicza w etykiety.

Ekran
Definiowanie etykiet


```
public
class ContentPane
    extends Content {

    public void initPane()
    {
        add(new JLabel("Hello"));
        add(new JLabel("World"));
    }
}
```

Przyciski

Przyciskiem jest komponent opisany przez klasę **JButton**. Oblicze prostego przycisku składa się tylko z tekstu.

klasa JButton

```
JButton(String text)
Konstruuje obiekt klasy JButton reprezentujący przycisk z opisem
text.
np. JButton start = new JButton("Start");

void setText(String text)
Zmienia opis przycisku na text.
np. start.setText("Done!");

String getText()
Dostarcza opis przycisku.
np. String text = start.getText();

void setActionCommand(String command)
Związuje z przyciskiem łańcuch command. Jego domyślną wartością
jest opis przycisku.
np. start.setActionCommand("1");
```

Następująca pod-aplikacja, pokazana na ekranie ***Definiowanie przycisków***, ilustruje wyposażanie oblicza w przyciski.

Ekran
Definiowanie przycisków


```
public
class ContentPane
    extends Content {
```

```
    public void initPane()
    {
        add(new JButton("Start"));
        add(new JButton("Stop"));
    }
}
```

Klatki

Klatką jest komponent opisany przez klasę **JTextField**. Użycie klatki umożliwia przygotowanie danych wykorzystywanych przez program.

klasa JTextField

```
JTextField(String text)
JTextField(int columns)
Konstruuje obiekt klasy JTextField reprezentujący klatkę, która
może pomieścić napis text albo columns znaków o średniej szeroko-
ści.
np. JTextField field = new JTextField(20);

void setText(String text)
Zmienia zawartość klatki na text.
np. field.setText("Data.txt");

String getText()
Dostarcza zawartość klatki.
np. String string = field.getText();
```

Następująca pod-aplikacja, pokazana na ekranie ***Definiowanie klatek***, ilustruje wyposażanie oblicza w klatki.

Ekran
Definiowanie klatek

```
public
class ContentPane
    extends Content {

    public void initPane()
    {
        add(new JTextField(10));
        add(new JTextField("Abracadabra"));
    }
}
```

Rozkłady

Rozmieszczanie komponentów w obrębie pojemnika, na przykład **okna**, **panelu**, itp. może się odbywać ***ręcznie*** albo z udziałem ***zarządcy rozkładu***. W pierwszym przypadku należy samemu określić położenie i rozmiary każdego komponentu. W drugim określenie położenia i rozmiarów jest **zbyteczne**, gdyż określi je zarządca.

Komponenty wyświetla się z rozmiarami obranymi przez zarządcę. Są to zazwyczaj ***rozmiary optymalne*** (określone za pomocą funkcji **setPreferredSize** albo domyślne) albo rozmiary określone za pomocą funkcji **setSize** albo **setBounds**.

klasa Container

```
void setLayout(LayoutManager mgr)
Narzuca pojemnikowi zarządcę rozkładu określonego przez mgr. Jeśli
mgr ma wartość null, to rozmieszczanie komponentów należy wykonać
ręcznie.
np. pane.setLayout(new FlowLayout());

Component add(Component control)
Component add(Component control, Object where)
Wstawia do pojemnika komponent control, umożliwiając za pomocą
where, określenie miejsca wstawienia. Dostarcza pierwszy argument.
np. pane.add(
        new JLabel("Author"), "North"
    );
```

Rozkład ręczny

Aby **zrezygnować** z usług zarządców rozkładu, należy metodę **setLayout** wywołać z argumentem **null**.

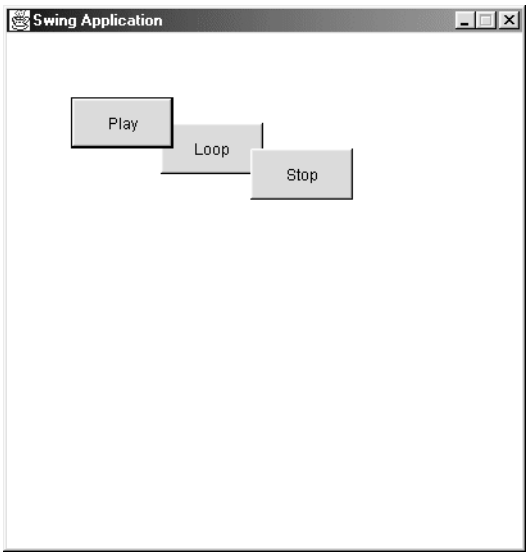
Położenie i rozmiar każdego komponentu należy wówczas określić za pomocą funkcji **setLocation** i **setSize** albo za pomocą funkcji **setBounds**.

klasa Component

```
void setLocation(int x, int y)
Umieszcza lewy-górny narożnik komponentu w punkcie (x, y).
np. control.setLocation(100, 200);

void setSize(int w, int h)
Nadaje komponentowi rozmiary w x h.
np. control.setSize(50, 50);

void setBounds(int x, int y, int w, int h)
Umieszcza lewy-górny narożnik komponentu w punkcie (x, y) i nadaje
mu rozmiary w x h.
np. control.setBounds(100, 200, 50, 50);
```

Następująca pod-aplikacja, pokazana na ekranie ***Rozkład ręczny***, rozmieszcza komponenty bez użycia zarządcy rozkładu.

Ekran
Rozkład ręczny




```
public
class ContentPane
    extends Content {

    private JButton play = new JButton("Play"),
                    loop = new JButton("Loop"),
                    stop = new JButton("Stop");

    public void initPane()
    {
        setLayout(null);

        play.setLocation(50, 50);
        play.setSize(80, 40);
        add(play);
```

```
        loop.setLocation(120, 70);
        loop.setSize(80, 40);
        add(loop);

        stop.setBounds(190, 90, 80, 40);
        add(stop);
    }
}
```

Rozkład ciągły

Aby narzucić pojemnikowi rozkład **ciągły**, należy metodę **setLayout** wywołać z argumentem

```
new FlowLayout()
```

W takim wypadku komponenty będą rozmieszczane ***rzędami***. W pierwszym rzędzie zarządca umieści maksymalną liczbę nie zachodzących na siebie komponentów, a pozostałe umieści analogicznie, w kolejnych rzędach. Komponenty każdego rzędu zostaną rozmieszczone **równomiernie**.

Komponenty przyjmują rozmiary **optymalne** (domyślne albo określone za pomocą funkcji **setPreferredSize**). Wydanie komponentowi polecenia **setSize** albo **setBounds** nie ma wpływu na rozmiary komponentu.

Następująca pod-aplikacja, pokazana na ekranie ***Rozkład ciągły***, rozmieszcza swoje komponenty przy pomocy zarządcy rozkładu ciągłego.

Ekran
Rozkład ciągły


```
public
class ContentPane
    extends Content {

    private JButton play = new JButton("Play"),
                    loop = new JButton("Loop"),
                    stop = new JButton("Stop");
```

```
    public void initPane()
    {
        setLayout(
            new FlowLayout()
        );

        add(play);
        add(loop);
        add(stop);
    }
}
```

Rozkład brzegowy

Aby narzucić pojemnikowi rozkład **brzegowy**, należy metodę **setLayout** wywołać z argumentem

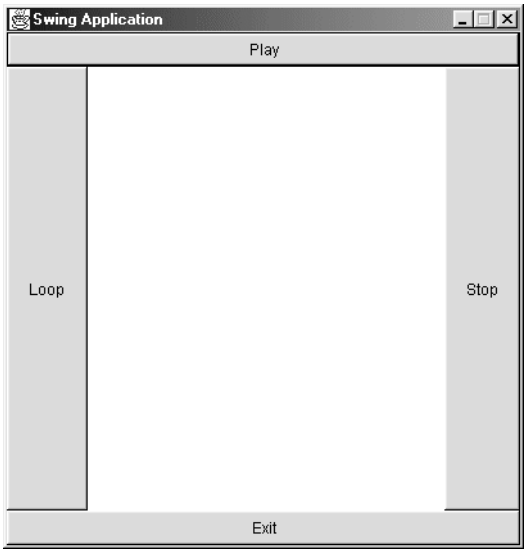
```
new BorderLayout()
```

W takim wypadku komponenty będą rozmieszczane w **5** obszarach: ***zachodnim*** (**West**), ***wschodnim*** (**East**), ***północnym*** (**North**), ***południowym*** (**South**) i *środkowym* (**Center**). Jeśli w takim zestawie nie użyje się pewnego obszaru, to może zostać **skonsumowany** przez pozostałe. Widocznym tego objawem są zazwyczaj nadmierne rozmiary komponentów.

Komponent umieszczony w obszarze zachodnim oraz wschodnim przyjmuje **szerokość** optymalną, a umieszczony w obszarze północnym oraz południowym przyjmuje **wysokość** optymalną. Wydanie komponentowi polecenia **setSize** albo **setBounds** nie ma wpływu na jego rozmiary.

Następująca pod-aplikacja, pokazana na ekranie ***Rozkład brzegowy***, rozmieszcza swoje komponenty przy pomocy zarządcy rozkładu brzegowego.

Ekran
Rozkład brzegowy



```
public
class ContentPane
    extends Content {

    private JButton play = new JButton("Play"),
                    loop = new JButton("Loop"),
                    stop = new JButton("Stop"),
                    exit = new JButton("Exit");

    public void initPane()
    {
        setLayout(
            new BorderLayout()
        );

        add(play, BorderLayout.NORTH);
        add(loop, BorderLayout.WEST);
        add(stop, BorderLayout.EAST);
        add(exit, BorderLayout.SOUTH);
    }
}
```

Rozkład siatkowy

Aby narzucić pojemnikowi rozkład **siatkowy**, należy metodę **setLayout** wywołać z argumentem

```
new GridLayout(rows, cols)
```

albo

```
new GridLayout(rows, cols, hGap, vGap)
```

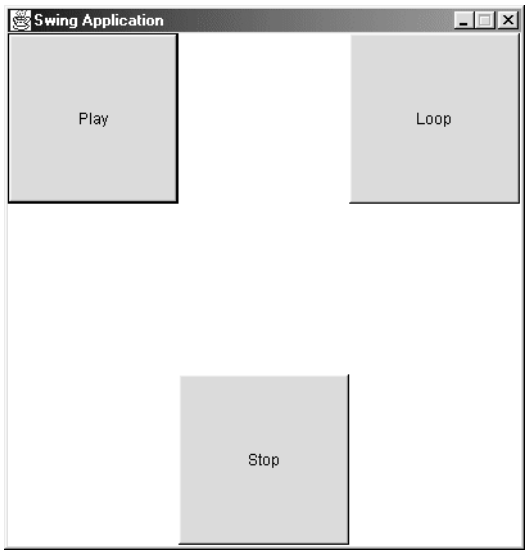
W takim wypadku komponenty będą rozmieszczane **wierszami**, w prostokątnym układzie ***rows*** *x* ***cols*** komórek o identycznych rozmiarach, z odstępami ***hGap*** (w poziomie) i ***vGap*** (w pionie).

Wypełnianie komórek siatki odbywa się **wierszami**. Jeśli jako liczbę **wierszy** poda się **0**, to ich liczba wyniknie z liczby komponentów i podanej liczby kolumn. Jeśli jako liczbę **kolumn** poda się **0**, to ich liczba wyniknie z liczby komponentów i podanej liczby wierszy.

Komponent wypełnia całą komórkę siatki. Rozmiary i rozmiary optymalne komponentu nie są brane pod uwagę.

Następująca pod-aplikacja, pokazana na ekranie ***Rozkład siatkowy***, rozmieszcza komponenty przy pomocy zarządcy rozkładu siatkowego.

Ekran
Rozkład siatkowy



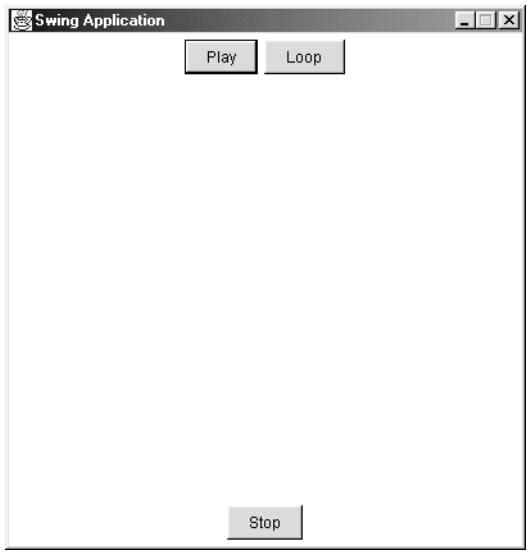

```
public
class ContentPane
    extends Content {

    private JButton play = new JButton("Play"),
                    loop = new JButton("Loop"),
                    stop = new JButton("Stop");

    public void initPane()
    {
        setLayout(
            new GridLayout(0, 3)
        );

        add(play);
        add(new JLabel(""));
        add(loop);

        for (int i = 0; i < 4 ; i++)
            add(new JLabel(""));

        add(stop);
    }
}
```

Grupowanie komponentów

Komponenty mogą być grupowane w ***panele*** rozmieszczane przez zarządców. Umożliwia to rozmieszczenie w tym samym **obszarze** rozkładu brzegowego albo w tej samej **komórce** rozkładu siatkowego, więcej niż jednego komponentu.

Następująca pod-aplikacja, pokazana na ekranie ***Grupowanie komponentów***, grupuje komponenty w panelach.

Ekran
Grupowanie komponentów




```
public
class ContentPane
    extends Content {

    private JButton play = new JButton("Play"),
                    loop = new JButton("Loop"),
                    stop = new JButton("Stop");

    public void initPane()
    {
        setLayout(
            new BorderLayout()
        );

        JPanel upPanel = new JPanel();
        upPanel.setBackground(Color.white);

        upPanel.add(play);
        upPanel.add(loop);

        add(upPanel, BorderLayout.NORTH);

        JPanel dnPanel = new JPanel();
        dnPanel.setBackground(Color.white);

        dnPanel.add(stop);

        add(dnPanel, BorderLayout.SOUTH);
    }
}
```

Wymiarowanie komponentów

Niekiedy **optymalne** i **domyślne** rozmiary komponentów okazują się zbyt małe. W takim przypadku można zdefiniować klasę pochodną od klasy komponentu predefiniowanego i umieścić w niej metodę **getPreferredSize** definiującą własne rozmiary komponentu albo wydać mu polecenie **setPreferredSize** z argumentem określającym rozmiary optymalne.

Zaleca się używanie funkcji **setPreferredSize**.

klasa Container

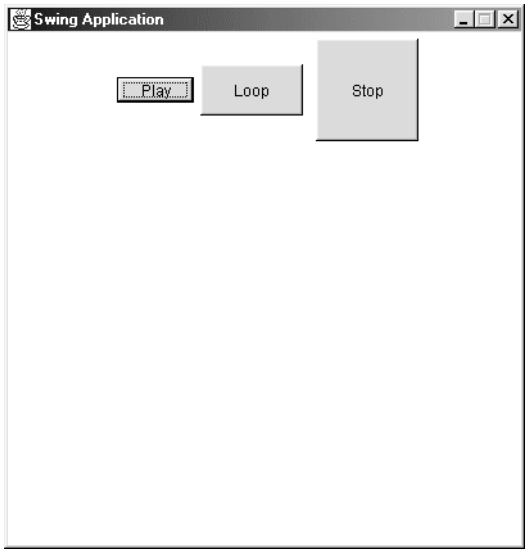
```
Dimension getPreferredSize()
Metodę należy zawrzeć w klasie komponentu i zdefiniować tak, aby
dostarczała jego oczekiwane rozmiary.
np. public Dimension getPreferredSize()
    {
        return new Dimension(w, h);
    }
```

klasa JComponent

```
void setPreferredSize(Dimension optimal)
Definiuje za pomocą optymalne rozmiary komponentu.
np. button.setPreferredSize(new Dimension(40, 40));
```

Następująca pod-aplikacja, pokazana na ekranie ***Wymiarowanie komponentów***, tworzy własną klasę komponentów.

Ekran
Wymiarowanie komponentów



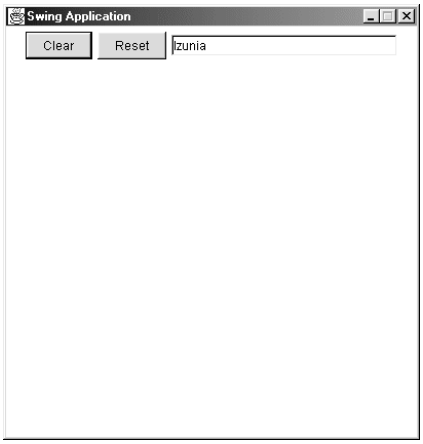
```
public
class ContentPane
    extends Content {

    private JButton play = new JButton("Play"),
                    loop = new JButton("Loop"),
                    stop = new JButton("Stop");

    public void initPane()
    {
        setLayout(
            new FlowLayout()
        );

        play.setPreferredSize(new Dimension(60, 20));
        loop.setPreferredSize(new Dimension(80, 40));
        stop.setPreferredSize(new Dimension(80, 80));

        Panel panel = new Panel();

        panel.add(play);
        panel.add(loop);
        add(panel);

        add(stop);
    }
}
```

Rozpoznawanie zdarzeń

Jeśli wywołanie metody obsługi może być spowodowane zajściem więcej niż jednego zdarzenia, to istotne staje się jego ***rozpoznanie***. Można tego dokonać za pomocą metody **getSource** oraz za pomocą operatora **instanceof**.

Następująca pod-aplikacja, pokazana na ekranie ***Rozpoznawanie zdarzeń***, umożliwia przeniesienie napisu z klatki na konsolę. Do wyczyszczenia klatki służy przycisk ***Clear***, a do jej zapełnienia określonym napisem, przycisk ***Reset***.

Ekran
Rozpoznawanie zdarzeń



```
public
class ContentPane
    extends Content {

    private String initial = "Hello";
    private JButton clear = new JButton("Clear"),
                    reset = new JButton("Reset");
    private JTextField input = new JTextField(20);

    public void initPane()
    {
        setLayout(new FlowLayout());

        add(clear);
        add(reset);
        add(input);

        Watcher watcher = new Watcher();

        clear.addActionListener(watcher);
        reset.addActionListener(watcher);
        input.addActionListener(watcher);
    }

    class Watcher
        implements ActionListener {

        public void actionPerformed(ActionEvent evt)
        {
            Object source = evt.getSource();

            if (source instanceof JTextField)
                System.out.println(input.getText());
            else if (source == clear)
                input.setText("");
            else
                input.setText(initial);
        }
    }
}
```

Menu

Wyposażenie aplikacji w menu wymaga utworzenia obiektu ***paska menu***, obiektów poszczególnych ***menu*** (np. ***File*** i ***Help***) oraz obiektów ***poleceń menu*** (np. ***Exit***). Po dodaniu poleceń do menu i menu do paska, należy ustanowić pasek menu.

klasa JMenuBar

`JMenuBar``()`
Konstruuje obiekt klasy **`JMenuBar`** reprezentujący pasek menu.
np. `JMenuBar menuBar = new JMenuBar();`

klasa JMenuItem

JMenuItem(String name)
Konstruuje obiekt klasy **JMenuItem** reprezentujący polecenie menu o nazwie ***name***.
np. JMenuItem item = new JMenuItem("Exit");

klasa JMenu

JMenu(String name)
Konstruuje obiekt klasy **JMenu** reprezentujący menu o nazwie ***name***.
np. JMenu menu = new JMenu("File");

Component **add**(JMenuItem item)
Dodaje do menu polecenie ***item***. Dostarcza odniesienie do ***item***.
np. menu.add(item);

void **addSeparator**()
Dodaje do menu separator.
np. menu.addSeparator();

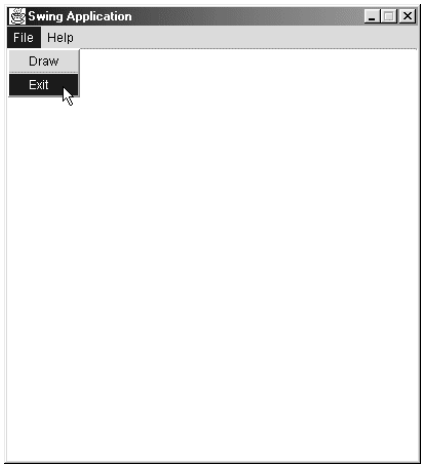
klasa JRootPane

void **setJMenuBar**(JMenuBar menuBar)
Wyposaża pojemnik główny w pasek menu ***menuBar***.
np. frame.setJMenuBar(menuBar);

Następująca pod-aplikacja, pokazana na ekranie ***Polecenia menu***, tworzy pasek menu składający się z menu ***File*** i ***Help***. Polecenie ***File / Draw*** służy do wykreślenia napisu pobranego z dialogu wejściowego, a polecenie ***Help / About*** do podania nazwiska autora aplikacji.

Pogrubiono komentarze opatrujące te instrukcje, które wiążą się z uwidocznieniem polecenia ***File / Exit***.

Ekran
Polecenia menu



```
public
class ContentPane
    extends Content {

    public ContentPane()
    {
            // zdefiniowanie paska menu
        JMenuBar menuBar = new JMenuBar();

            // zdefiniowanie menu File
        JMenu file = new JMenu("File");

            // zdefiniowanie polecenia File / Draw
        JMenuItem draw = new JMenuItem("Draw");

            // zdefiniowanie polecenia File / Exit
        JMenuItem exit = new JMenuItem("Exit");

            // dodanie polecenia Exit do menu File
        file.add(draw);

            // dodanie separatora do menu File
        file.addSeparator();

            // dodanie polecenia Draw do menu File
        file.add(exit);

            // zdefiniowanie menu Help
        JMenu help = new JMenu("Help");

            // zdefiniowanie polecenia Help / About
        JMenuItem about = new JMenuItem("About");

            // dodanie polecenia About do menu Help
        help.add(about);

            // dodanie menu File do paska menu
        menuBar.add(file);

            // dodanie menu Help do paska menu
        menuBar.add(help);

            //  ustanowienie paska menu
        setJMenuBar(menuBar);

            // zdefiniowanie obsługi polecenia Exit
        exit.addActionListener(
            new ActionListener() {
                public void
                actionPerformed(ActionEvent evt)
                {
                    System.exit(0);
                }
            }
        );
```

```
        draw.addActionListener(
            new ActionListener() {
                public void
                actionPerformed(ActionEvent evt)
                {
                    string = JOptionPane.showInputDialog(
                                "Enter arguments"
                             );
                    if (string == null)  // Cancel
                        string = "";
                    repaint();
                }
            }
        );

        about.addActionListener(
            new ActionListener() {
                public void
                actionPerformed(ActionEvent evt)
                {
                    JOptionPane.showMessageDialog(
                        null, "Author: Jan Bielecki"
                    );
                }
            }
        );

        setBackground(Color.white);
    }

    private int h;
    private String string = "";

    public void initPane()
    {
        h = getHeight();
    }

    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        gDC.drawString(string, 20, h/2);
    }
}
```

Wyskakujące menu

W celu wyświetlenia wyskakującego menu należy wykonać ***akcję wyzwalającą***. Sposób wykonania akcji zależy od *Systemu*, ale zawsze można ją rozpoznać za pomocą funkcji **isPopupTrigger**.

W systemach *Windows* akcją wyzwalającą jest zwolnienie **prawego** przycisku myszki. W *Systemach* z myszką jedno-przyciskową jest nią naciśnięcie ustalonego klawisza, połączone z operacją wykonaną za pomocą myszki.

W celu **przenośnego** obsłużenia akcji wyzwalającej, nie należy się posługiwać funkcją **isMetaDown**, a wywołanie funkcji **isPopupTrigger** umieścić w **mousePressed**, **mouseReleased** oraz w **mouseClicked**.

klasa JPopupMenu

```
JPopupMenu()
Konstruuje obiekt reprezentujący wyskakujące menu.
np. JPopupMenu popup = new JPopupMenu();

void show(Component invoker, int x, int y)
Wyświetla wyskakujące menu w prostokącie z lewym-górnym narożnikiem
w (x, y) względem komponentu invoker.
np. popup.show(source, 20, 30);
```

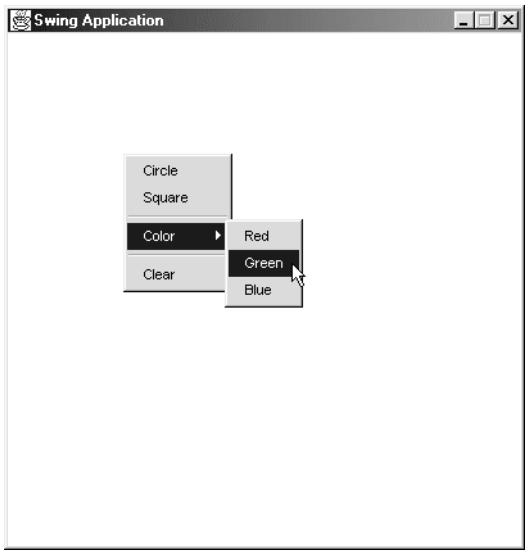
klasa MouseEvent

```
boolean isPopupTrigger()
Dostarcza orzeczenie o wartości: "wykonano akcję wyzwalającą".
np. boolean isPopup = evt.isPopupTrigger();
```

Następująca pod-aplikacja, ilustruje zasadę utworzenia wielo-poziomowego, ***wyskakującego menu***. Po wykonaniu *p*-kliknięcia jest wyświetlane menu, pokazane na ekranie ***Polecenia menu***. Menu składa się z poleceń ***Circle***, ***Square*** i ***Clear***. Wydanie polecenia ***Circle*** i ***Square*** powoduje wykreślenie okręgu albo kwadratu, a wydanie polecenia ***Clear*** powoduje wyczyszczenie pulpitu.

Ponieważ wykreślenie figury odbywa się za pomocą własnego wykreślacza, a nie jest tworzona baza figur odtwarzanych w funkcji **paintComponent**, więc występuje **kolizja** między wykreślanymi figurami, a odtworzeniem fragmentu pulpitu zajętego przez menu. Jej usunięcie może się okazać wartościowym ćwiczeniem praktycznym.

Ekran
Polecenia menu




```
public
class ContentPane
    extends Content
    implements ActionListener {

    private int x, y;
    private Color color = Color.black;

    public void initPane()
    {
        final JPopupMenu popup =
            new JPopupMenu();

        JMenuItem mi;

        mi = new JMenuItem("Circle");
        mi.addActionListener(this);
        popup.add(mi);

        mi = new JMenuItem("Square");
        mi.addActionListener(this);
        popup.add(mi);

        popup.addSeparator();

        JMenu color = new JMenu("Color");
        String colors[] = { "Red", "Green", "Blue" };
        for (int i = 0; i < 3 ; i++) {
            mi = new JMenuItem(colors[i]);
            mi.setActionCommand("" + i);
            mi.addActionListener(this);
            color.add(mi);
        }
```

```
        popup.add(color);

        popup.addSeparator();

        mi = new JMenuItem("Clear");
        mi.addActionListener(this);
        popup.add(mi);

        addMouseListener(
            new MouseAdapter() {
                public void
                mouseReleased(MouseEvent evt)
                {
                    if (evt.isPopupTrigger()) {
                        x = evt.getX();
                        y = evt.getY();
                        popup.show(
                            evt.getComponent(),
                            x, y
                        );
                    }
                }
            }
        );

        add(popup);  // dodaj do pulpitu
    }

    public void
    actionPerformed(ActionEvent evt)
    {
        String cmd = evt.getActionCommand();
        Graphics gDC = getGraphics();
        gDC.setColor(color);
        if (cmd =="Circle")
            gDC.drawOval(x-25, y-25, 50, 50);
        else if (cmd == "Square")
            gDC.drawRect(x-25, y-25, 50, 50);
        else if (cmd == "Clear")
            repaint();
        else {
            Color rgb[] =
                {
                    Color.red,
                    Color.green,
                    Color.blue
                };
            color = rgb[new Integer(cmd).intValue()];
        }
    }
}
```

Odrębne okna

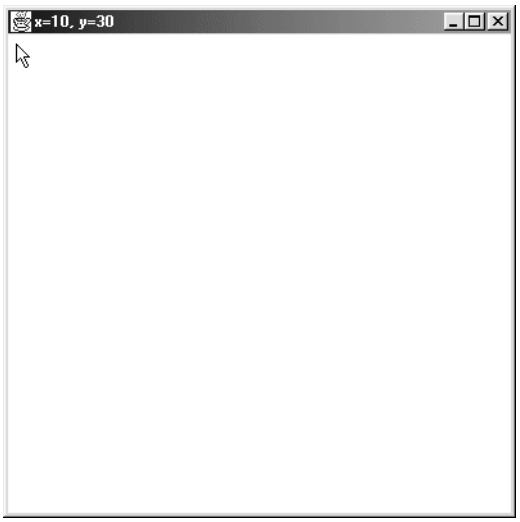
W celu utworzenia odrębnego okna aplikacji należy utworzyć obiekt klasy pochodnej od **JFrame**, a następnie wydać mu polecenie **setSize** albo **pack** oraz polecenie **show** albo **setVisible** (z argumentem **true**). W celu określenia miejsca okna na ekranie, można dodatkowo użyć polecenia **setLocation**.

klasa JFrame

```
JFrame(String caption)
Konstruuje obiekt klasy JFrame reprezentujący okno opatrzone paskiem tytułowym z napisem caption.
np. JFrame frame = new JFrame("Framed window");

void setSize(int w, int h)
Nadaje oknu rozmiary w x h.
np. frame.setSize(400, 200);

void setLocation(int x, int y)
Umieszcza lewy-górny narożnik okna w punkcie (x, y) ekranu.
np. frame.setLocation(50, 50);

void setTitle(String caption)
Zmienia napis na pasku tytułowym okna na caption.
np. frame.setTitle("Frame");

void pack()
Nadaje oknu minimalne rozmiary umożliwiające pomieszczenie jego wszystkich komponentów.
np. frame.pack();

void show()
Wyświetla okno.
np. frame.show();
```

Następująca pod-aplikacja, pokazana na ekranie ***Współrzędne okna***, tworzy odrębne okno i na jego pasku tytułowym pokazuje współrzędne kursora myszki. Są one liczone względem lewego-górnego narożnika **okna**, a nie względem narożnika jego **pulpitu**.

Ekran
Współrzędne okna




```
public
class ContentPane
    extends Content {

    private JFrame frame;

    public void initPane()
    {
       frame = new JFrame("Framelet");

       frame.setLocation(100, 100);
       frame.setSize(400, 400);

       frame.show();

       final Graphics gDC = frame.getGraphics();

       frame.addMouseMotionListener(
           new MouseMotionAdapter() {
               public void mouseMoved(MouseEvent evt)
               {
                   int x = evt.getX(),
                       y = evt.getY();
                   gDC.setColor(Color.white);
                   gDC.fillRect(
                       0, 0,
                       frame.getWidth(), frame.getHeight()
                   );

                   frame.setTitle("x=" + x + ", y=" + y);
               }
           }
       );
    }
}
```

Uwzględnianie wcięć

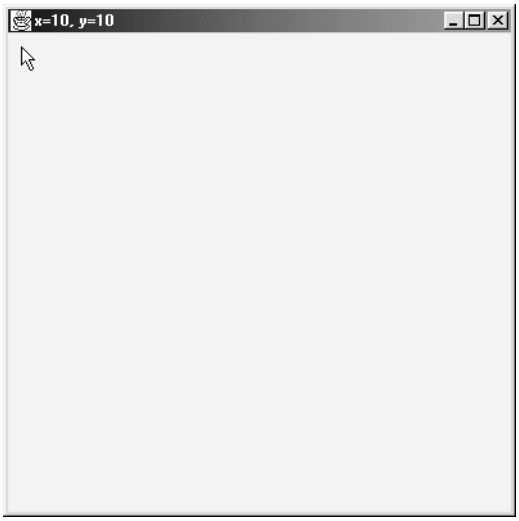
Okno ma domyślnie narzucony rozkład **brzegowy**. Ponieważ jest wyposażone w obrzeża i pasek tytułowy, a niekiedy i w menu, więc rozmiary jego pulpitu wynikają z uwzględnienia ***wcięć***: ***lewego*** (*left*), ***górnego*** (*top*), ***prawego*** (*right*) i ***dolnego*** (*bottom*). W szczególności współrzędnymi lewego-górnego narożnika pulpitu okna nie jest(**0**, **0**) ale (***left***, ***top***).

klasa Container

```
Insets getInsets()
Dostarcza odniesienie do obiektu określającego wcięcia okna.
np. Insets ins = frame.getInsets();
    int left   = ins.left,
        top    = ins.top,
        right  = ins.right,
        bottom = ins. bottom;
```

Następująca pod-aplikacja, pokazana na ekranie ***Współrzędne pulpitu***, tworzy odrębne okno i na jego pasku tytułowym pokazuje współrzędne kursora myszki. Są one liczone względem lewego-górnego narożnika **pulpitu** okna.

Ekran
Współrzędne pulpitu




```
public
class ContentPane
    extends Content {

    private JFrame frame;

    public void initPane()
    {
       frame = new JFrame("Framelet");
```

```
        frame.setLocation(100, 100);
        frame.setSize(400, 400);

        frame.show();

        final Graphics gDC = frame.getGraphics();
        Insets ins = frame.getInsets();
        final int lt = ins.left,
                  tp = ins.top,
                  rt = ins.right,
                  bt = ins.bottom;

        frame.addMouseMotionListener(
            new MouseMotionAdapter() {
                public void mouseMoved(MouseEvent evt)
                {
                    int x = evt.getX(),
                        y = evt.getY();
                    gDC.setColor(Color.yellow);
                    gDC.fillRect(
                        lt, tp,
                        frame.getWidth()  - (lt+rt),
                        frame.getHeight() - (tp+bt)
                    );

                    frame.setTitle(
                        "x=" + (x-lt) +
                        ", y=" + (y-tp)
                    );
                }
            }
        );
    }
}
```

Wstawianie pulpitu

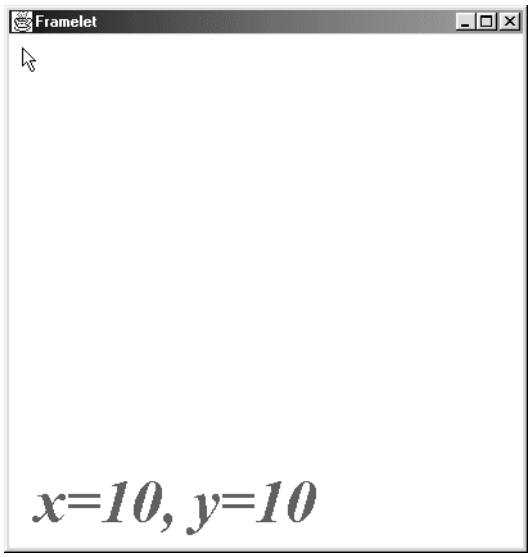
Posługiwanie się wcięciami jest na ogół dość uciążliwe. Znacznie wygodniejsze jest zdefiniowanie pulpitu jako **panelu** i wstawienie go do okna. Po tym zabiegu układ współrzędnych pulpitu zostaje zamocowany w jego lewym-górnym narożniku.

klasa JFrame

```
void setContentPane(Container panel)
Wstawia pulpit do okna.
np. frame.setContentPane(new MyPanel());
```

Następująca pod-aplikacja, pokazana na ekranie ***Wstawianie pulpitu***, tworzy odrębne okno i pokazuje w nim współrzędne kursora myszki. Są one liczone względem lewego-górnego narożnika **pulpitu** okna.

Ekran
Wstawianie pulpitu




```
public
class ContentPane
    extends Content {

    private JFrame frame;
    private Font font =
       new Font("Serif", Font.BOLD | Font.ITALIC, 48);

    public void initPane()
    {
       frame = new JFrame("Framelet");

       frame.setLocation(100, 100);

       final JPanel panel = new JPanel();
       panel.setPreferredSize(
           new Dimension(400, 400)
       );

       frame.setContentPane(panel);

       frame.pack();
       frame.show();

       final Graphics gDC = panel.getGraphics();
       gDC.setFont(font);

       panel.addMouseMotionListener(
           new MouseMotionAdapter() {
               public void mouseMoved(MouseEvent evt)
               {
                   int x = evt.getX(),
                       y = evt.getY();
```

```
                    gDC.setColor(Color.white);
                    gDC.fillRect(
                        0, 0,
                        panel.getWidth(),
                        panel.getHeight()
                    );
                    gDC.setColor(Color.red);
                    gDC.drawString(
                        "x=" + x + ", y=" + y,
                        20, panel.getHeight()-20
                    );
                }
            }
        );
    }
}
```

Zamykanie okna

Kliknięcie ikony zamknięcia okna powoduje zniknięcie okna z ekranu, ale jeśli jest to okno aplikacji, to nie powoduje to zakończenia wykonywania samej aplikacji. Dlatego w celu zakończenia wykonywania aplikacji należy **obsłużyć** zamknięcie okna.

Następująca aplikacja wyprowadza na konsolę **gwiazdki** i kończy się w chwili zamknięcia jej okna. Gdyby nazwę **windowClosing** zmieniono na taką, która **nie jest** funkcją obsługi (np. na **windowClosing2**), to na skutek **braku obsługi** zamknięcia okna aplikacja wyprowadzałaby gwiazdki "w nieskończoność"; nawet po zamknięciu jej okna (sic!).

```
package jbPack;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public
class Program {

    public static void main(String[] args)
    {
        new Program();
    }

    public Program()
    {
        JFrame frame = new JFrame("Framelet");

        frame.setLocation(100, 100);
        frame.setSize(300, 300);

        frame.show();
```

```
        frame.addWindowListener(
            new WindowAdapter() {
                public void
                windowClosing(WindowEvent evt)
                {
                    System.exit(0);
                }
            }
        );

        while (true) {
            System.out.print("*");
            try {
                Thread.sleep(500);
            }
            catch (InterruptedException e) {}
        }
    }
}
```

Wykreślanie czcionek

O zestawie dostępnych czcionek decyduje *System*. W spolonizowanych systemach *Windows* i tandemie

Java 2 Platform / Java 2 SDK 1.3 / Kawa 5.0

polskie litery, o kodach w tabeli ***Polskie litery***, są dostępne praktycznie bez ograniczeń (brakuje ich tylko wśród czcionek *Serif-PLAIN*).

Tabela Polskie litery

ą	ć	ę	ł	ń	ó	ś	ź	ż
0105	0107	0119	0142	0144	00f3	015b	017a	017c
Ą	Ć	Ę	Ł	Ń	Ó	Ś	Ź	Ż
0104	0106	0118	0141	0143	00d3	015a	0179	017b

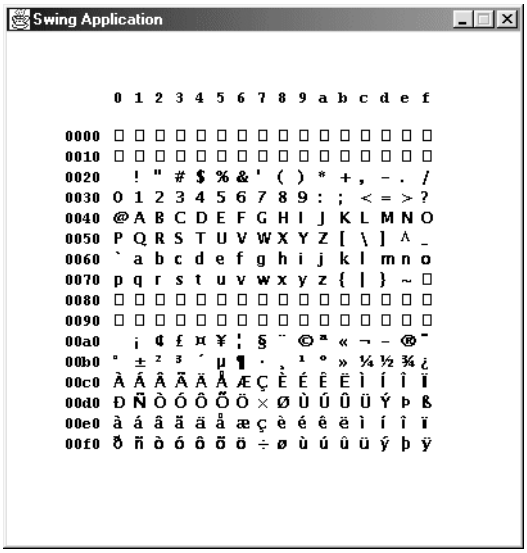
We wszystkich systemach są dostępne czcionki ***Lucida***. Z małymi ograniczeniami udostępniają one m.in. litery **polskie**, **rosyjskie**, **arabskie** i **hebrajskie**.

Zaleca się używanie czcionek ***Lucida Sans*** i stylu **BOLD**. Wśród innych krojów i stylów może brakować czcionek narodowych.

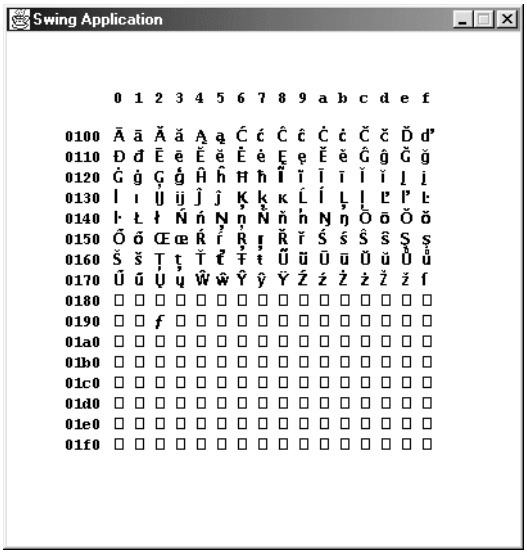
Następująca pod-aplikacja, z wynikami na ekranach ***Strona 0000*** do ***Strona 0600***, itd. ujawnia pierwszych **7** stron znaków *Lucida Sans - BOLD* dostarczanych z *JDK 1.3*. Wśród nich znajdują się litery narodowe, w tym polskie. Przemieszczanie między stronami odbywa się za pomocą **kliknięć** (do przodu) i *p*-**kliknięć** (wstecz).

Niepuste są ponadto strony: **1e00**, **2000-2200**, **25c0**, **f800**, **fb00**, **fc00** i **fe00** (ostatnia z nich zawiera wiele znaków arabskich). Zachęca się do wykonania eksperymentów z innymi czcionkami (np. **Serif**).

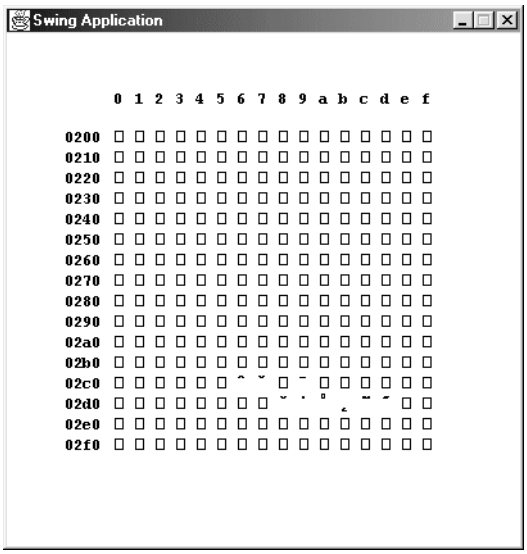
Ekran
Strona 0000



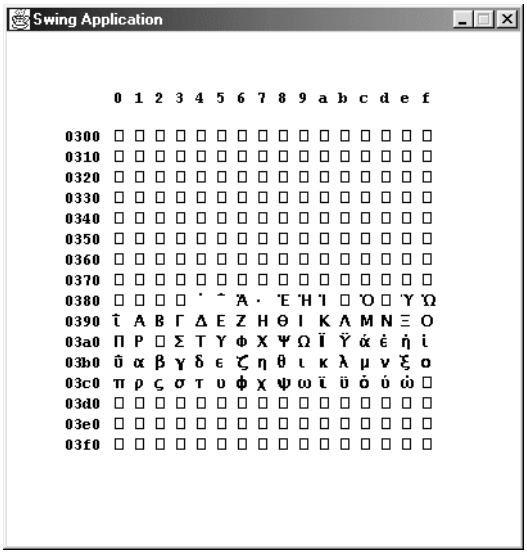
Ekran
Strona 0100



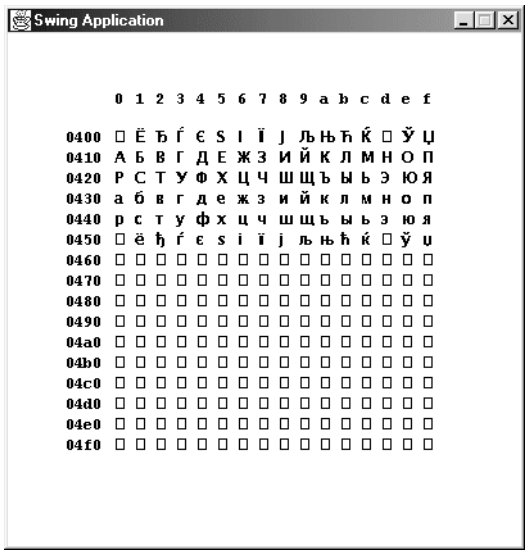
Ekran
Strona 0200



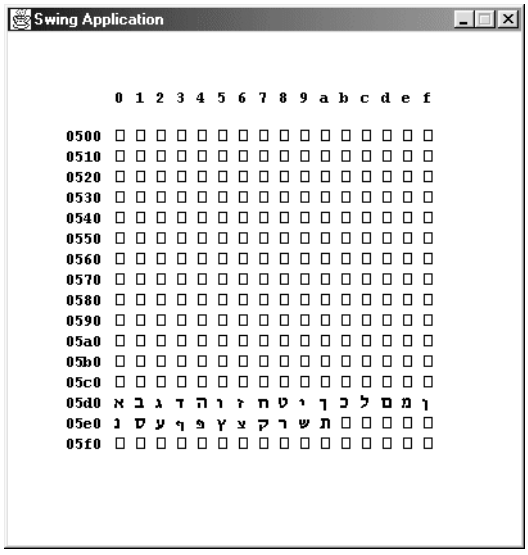
Ekran
Strona 0300



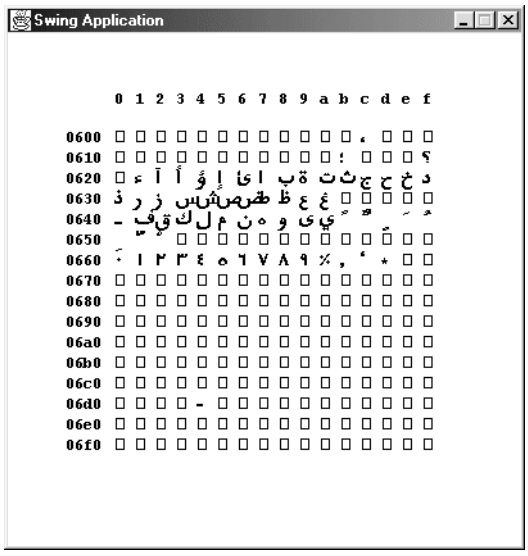
Ekran
Strona 0400



Ekran
Strona 0500



Ekran
Strona 0600




```
public
class ContentPane
    extends Content {

    private String fontFace = "Lucida Sans";

    private String lines[] = new String[16],
                   header  = "0123456789abcdef";
    private int base;
    private Font mono = new Font("Monospaced", Font.BOLD, 12),
                 font = new Font(fontFace, Font.BOLD, 12);
    private FontMetrics fmFont, fmMono;

    public void initPane()
    {
        Graphics gDC = getGraphics();
        fmMono = gDC.getFontMetrics(mono);
        base = 0;

        addMouseListener(
            new MouseAdapter() {
                public void
                mouseReleased(MouseEvent evt)
                {
                    if (evt.isMetaDown())
                        base -= 256;
                    else
                        base += 256;
                    base = base & 0xffff;

                    repaint();
                }
            }
        );
    }
```

```
    public void paintComponent(Graphics gDC)
    {
        super.paintComponent(gDC);

        gDC.translate(25, 25);

        char chr;
        for (int n = 0, i = 0; i < 16 ; i++) {
            String line = "";
            for (int j = 0; j < 16 ; j++) {
                chr = (char)(base + n++);
                line += chr;
            }
            lines[i] = line;
        }

        gDC.setFont(mono);
        int bW = fmMono.stringWidth("0000");
        for (int j = 0; j < 16 ; j++) {
            chr = header.charAt(j);
            gDC.drawString("" + chr, bW+30 + j*16, 30);
        }

        for (int i = 0; i < 16 ; i++) {
            gDC.drawString(getBase(i), 20, 60+16*i);
            gDC.setFont(font);
            for (int j = 0; j < 16 ; j++) {
                chr = lines[i].charAt(j);
                gDC.drawString(
                    "" + chr, bW+30 + j*16, 60+16*i
                );
            }
            gDC.setFont(mono);
        }
    }

    String getBase(int n)
    {
        int val = base + 16 * n;
        String hex = Integer.toHexString(val);
        switch (hex.length()) {
            case 1:
               hex = "0" + hex;
            case 2:
               hex = "0" + hex;
            case 3:
               hex = "0" + hex;
        }
        return hex + " ";
    }
}
```

Projektowanie apletów

Apletem jest program składający się z zestawu definicji klas, wśród których występuje **publiczna** klasa ***apletowa*** wywodząca się od klasy **JApplet**.

```
package jbPack;

import javax.swing.*;

public
class Program
    extends JApplet {

    // ...
}
```

Ponieważ odpalenie apletu jest realizowane przez **przeglądarkę**, a złośliwie napisane aplety mogłyby zagrozić integralności ***Systemu***, więc bez udzielenia przeglądarce specjalnych uprawnień, aplety nie mogą uzyskiwać dostępu do urządzeń systemowych, m.in. dysku (z jedynym wyjątkiem głośnika), a każde okno utworzone przez aplet jest opatrywane ostrzeżeniem ***Warning: Applet Window***. Dzięki temu unika się możliwości ujawnienia prywatnych informacji.

Następujący aplet, pokazany na ekranie ***Okno apletowe***, służy tylko do wyświetlenia okna.

Ekran
Okno apletowe

```
package jbPack;

import javax.swing.*;
import java.awt.*;

public
class Program
     extends JApplet {

     public void init()
     {
          JFrame frame = new JFrame("Swing");
          frame.setLocation(150, 150);
          frame.setSize(400, 400);
          frame.getContentPane().
                setBackground(Color.white);
          frame.show();

     }
}
```

Przeglądarka

W odróżnieniu od aplikacji, która jest zazwyczaj programem **samodzielnym**, wykonanie apletu wymaga użycia ***przeglądarki***. Przeglądarka niejawnie tworzy obiekt klasy apletowej, a następnie wydaje mu polecenia wyrażone przez metody **init**, **start** i **paint**. Ponadto ***obiektowi apletowemu*** mogą być wydawane polecenia **stop** i **destroy**.

Metoda **init** oraz metoda **destroy** jest wywoływana **jednokrotnie**. Metody **start**, **stop** i **paint** mogą być wywoływane **wielokrotnie**.

1. Metoda **init** jest wywoływana tuż przed pierwszym wyświetleniem apletu.
2. Metoda **start** jest wywoływana tuż po wywołaniu metody **init** oraz przed każdym ponownym wyświetleniem apletu.
3. Metoda **paint** jest wywoływana tuż po wykonaniu metody **start**, a także w każdej sytuacji, gdy zaistnieje potrzeba odtworzenia pulpitu apletu (na przykład po zasłonięciu i odsłonięciu go przez pewne okno).
4. Metoda **stop** jest wywoływana w chwili zniknięcia apletu ze strony *WWW*.
5. Metoda **destroy** jest wywoływana tuż przed zamknięciem przeglądarki.

Jeśli aplet inicjuje odtwarzanie pliku **dźwiękowego**, to wyłączenie głośnika powinno się odbyć w metodzie **stop**, a jego ponowne włączenie w metodzie **start**.

Następująca aplikacja, pokazana na ekranie ***Pozdrowienie z apletu***, wykreśla na swoim **pulpicie** tradycyjny ciąg znaków.

Ekran
Pozdrowienie z apletu


```
package jbPack;

import javax.swing.*;
import java.awt.*;

public
class Program
    extends JApplet {

    private Font font;

    public void init()
    {
        font = new Font(
                    "Serif",
                    Font.BOLD | Font.ITALIC,
                    24
                );
        getContentPane().
            setBackground(Color.white);
    }

    public void paint(Graphics gDC)
    {
        gDC.setFont(font);

        gDC.drawString(
            "Hello, I am JanB.", 10, 100
        );
    }
}
```

Opisy apletów

Aby przeglądarka mogła wyświetlić aplet, należy jej przekazać ***opis apletu***. Opis umieszcza się w pliku z rozszerzeniem **.html** i nadaje mu najczęściej postać

```
<applet  code=fullName.class
         codebase=codeURL
         width=width
         height=height>
</applet>
```

w której ***fullName*** jest pełną nazwą klasy opisującej aplet (np. **jbPack.Program**), ***codeURL*** jest **lokalizatorem**, a ***width*** i ***height*** są rozmiarami ramki przydzielonej apletowi na stronie *WWW*.

Lokalizatory

Lokalizator ***codeURL*** można pobrać do programu za pomocą metody **getCodeBase**. Jeśli w opisie apletu nie ma parametru **codebase**, to wywołanie metody **getCodeBase** ma taki sam skutek jak wywołanie metody **getDocumentBase**.

klasa JApplet

```
URL getDocumentBase()
Dostarcza lokalizator pliku zawierającego opis apletu.
np. URL codeBase = getDocumentBase();  // file:/d:/Java/Index.html

URL getCodeBase()
Dostarcza lokalizator użyty w jawnym albo w domniemanym parametrze
codebase apletu.
np. URL codeBase = getCodeBase();      // file:/d:/Java/
```

Kompilacja

Kompilacja klasy ***Name*** programu odbywa się w taki sposób, że jeśli należy ona do pakietu ***pack***, a katalogiem wyjściowym jest ***output***, to skompilowany *B*-kod klasy umieszcza się w pliku o poglądowej nazwie

***output*/*pack*/*Name*.class**

W szczególności, jeśli następujący plik **Program.java** skompiluje się do katalogu wyjściowego **C:\jbOutputs**, to powstanie plik

```
C:\jbOutputs\jbPack\jbHello\Program.class
```

zawierający *B*-kod apletu **Program**.

plik Program.java

```
package jbPack.jbHello;

import javax.swing.*;
import java.awt.*;

public
class Program
    extends JApplet {

    public void init()
    {
        getContentPane().
            setBackground(Color.white);
    }

    public void paint(Graphics gDC)
    {
        gDC.drawString("Hello", 20, 50);
    }
}
```

Jeśli przeglądarce dostarczy się lokalizator pliku *HTML* na przykład

```
file:/c:/jbWeb/Index.html
```

albo

```
http://bolek.ii.pw.edu.pl/~jbl/jbWeb/Index.html
```

w którym występuje opis

```
<applet  code=jbPack.jbHello.Program.class
         codebase=codeURL
         width=200
         height=100>
</applet>
```

to spowoduje to wykonanie apletu **Program** należącego do pakietu **jbPack.jbHello**.

Poszukiwanie pliku **Program.class** apletu odbędzie się kolejno

1. W katalogach parametru **classpath**.
2. W katalogu określonym przez (jawny albo domniemany) parametr **codebase**.

Poszukiwanie zakończy się w chwili znalezienia pliku.

W szczególności, jeśli plik **Program.class** umieści się w katalogu

```
c:\jbPacks\jbPack\jbHello
```

a przeglądarce ustawi się parametr środowiska

```
classpath=c:\jbPacks
```

to plik zostanie znaleziony w **c:\jbPacks\jbPack\jbHello**, a lokalizator zawarty w parametrze **codebase** nie będzie wzięty pod uwagę (sic!).

Jeśli w **classpath** nie poda się nazwy katalogu **c:\jbPacks**, to można użyć parametru

```
codebase=file:/c:/jbPacks
```

dla dociekliwych

Opis apletu, zawarty w pliku *HTML*, ma w ogólnym przypadku postać

```
<applet code=fullName.class
        archive=jarList
        codebase=codeURL
        width=width
        height=height
        name=appletName
        align=align
        vspace=vSpace
        hspace=hSpace
>
<param name=parName value=parValue>
...
</applet>
```

Fraza **code** podaje kwalifikowaną nazwę klasy apletowej. Frazy **archive** i **codebase** określają położenie katalogu, w którym znajduje się plik zawierający *B*-kod tej klasy, na przykład

```
archive=jbClasses.jar
codebase=file:/c:/jbPacks/
```

albo

```
archive="jars/Mary.jar,jars/John.jar"
codebase=http://bolek.ii.edu.pl/~jbl/jbPacks
```

a frazy **width** i **height** określają rozmiary ramki na stronie *WWW*.

Fraza **name** określa pełną nazwę identyfikującą aplet, fraza **align** określa sposób rozmieszczenia apletu na stronie *WWW* (m.in. **left**, **right**, **top**, **bottom**), a frazy **vspace** i **hspace** określają (wyrażony w pikselach) pionowy i poziomy odstęp przed i po aplecie.

Za pomocą fraz **param** można określić wartości parametrów przekazywanych do apletu.

Jeśli użyto frazy **archive**, to plik z *B*-kodem apletu będzie w pierwszej kolejności poszukiwany w podanych archiwach (lokalizowanych względem katalogu zawierającego plik *HTML*), a dopiero potem w katalogach określonych za pomocą parametru **classpath** i parametru **codebase**.

W wypadku użycia frazy **archive**, ta sama kolejność poszukiwania pliku co dla klasy **Program** dotyczy wszystkich innych klas, które są ładowane podczas wykonywania apletu.

W szczególności, gdyby opis apletu **Program** należącego do pakietu **jbTools.jbAudio**

```
<applet  code=jbTools.jbAudio.Program.class
         archive="jars/Mary.jar,jars/John.jar"
         codebase=file:/d:/jbPacks
         width=200
         height=100>
</applet>
```

znajdował się w pliku

```
d:\jbHTML\Index.html
```

to plik z *B*-kodem apletu **Program** byłby poszukiwany kolejno w archiwach

```
d:\jbHTML\jars\Mary.jar
d:\jbHTML\jars\John.jar
```

oraz w katalogu

```
c:\jbPacks\jbTools\jbAudio
```

W przypadku pominięcia frazy **codebase**, plik z *B*-kodem apletu **Program** byłby poszukiwany najpierw w podanych archiwach, a dopiero po tym w pliku

```
d:\jbHTML\jbTools\jbAudio
```

Następująca aplikacja ilustruje zastosowanie frazy **name** do komunikowania się apletów o nazwach **One** i **Two**. Naciśnięcie przycisku myszki w obrębie pulpitu jednego z nich powoduje wykreślenie **okręgu** na pulpicie **drugiego**.

Przeglądarka tworzy **2** aplety. Opisy apletów w dokumencie *HTML* są **różne**, ale *B*-kod obu apletów jest **wspólny**.

opis apletu

```
<applet  code=jbPack.Program.class
         width=200 height=200 name=One>
</applet>

<applet  code=jbPack.Program.class
         width=200 height=200 name=Two>
</applet>
```

program apletu

```
package jbPack;

import javax.swing.*;
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
```

```
public
class Program
    extends JApplet {

    protected String name;
    protected Thread myThread = null;

    public void init()
    {
        name = getParameter("name");

        Watcher watcher = new Watcher();

        addMouseListener(watcher);

        getContentPane().
            setBackground(Color.white);
    }

    Applet otherApplet()
    {
        String other = null;

        if (name.equals("One"))
            other = "Two";

        if (name.equals("Two"))
            other = "One";

        AppletContext context =
            getAppletContext();

        return context.getApplet(other);
    }

    class Watcher
        extends MouseAdapter {

        public void mousePressed(MouseEvent evt)
        {
            int x = evt.getX(),
                y = evt.getY();

            Applet other = otherApplet();

            Graphics gDC = other.getGraphics();

            gDC.drawOval(x-20, y-20, 40-1, 40-1);
        }
        public void mouseReleased(MouseEvent evt)
        {
            otherApplet().repaint();
        }
    }
}
```