

Podstawy programowania obiektowego w języku Java – dla programujących w C++

Wykład 8

Użyte przykłady częściowo zaczerpnięte z książki: T. Lis: "Java - Ćwiczenia praktyczne", Helion 2004 - wszystkie można ściągnąć z <ftp://helion.com.pl>

Ogólne zasady języka, jego kompilacji i wykonania

Podstawową zasadą języka Java jest *przenośność* – tzn. niezależność od środowiska programowego (system operacyjny) i sprzętowego (procesor, urządzenia zewnętrzne), w którym program się wykonuje oraz uniemożliwienie powstawania dialektów języka (tzn. jego implementacji niezgodnych ze specyfikacją lub wprowadzających rozszerzenia)

W celu osiągnięcia tej przenośności liczne aspekty języka są znacznie bardziej szczegółowo zdefiniowane w specyfikacji niż to było w C++:

- precyzyjnie określono postać i zakresy elementarnych typów danych, które w implementacjach innych języków były zależne sprzętowo,
- dokładnie określono zbiór sytuacji, które powodują błędy wykonania,
- uniemożliwiono korzystanie przez programistę z usług API systemu operacyjnego dostarczając w zamian zbiór standardowych klas oferujących typowe usługi z poziomu API SO.

**Podstawowe konstrukcje składniowe Javy są podobne do C++.
Dla zapoznania się z językiem java wystarczy wskazać różnice w
stosunku do C++**

Zasady konstrukcji programu, jego kompilacji i wykonania

W Javie wszystkie byty programistyczne muszą być elementami klas – nic nie może być zdefiniowane poza klasą

Program w Javie składa się ze zbioru klas.

Funkcja od której zaczyna się wykonanie programu musi być zadeklarowana jako metoda klasy publicznej z danego pliku źródłowego o deklaracji

```
public static void main ( String[] args );
```

Najprostszy program w Javie (plik Program.java)

```
public
class Program
{
    public static void main (String args[])
    {
        System.out.println( "Program uruchomiony" );
    }
}
```

```
}
```

Klasy składające się na program po skompilowaniu tworzą zbiór plików wynikowych o nazwach postaci:

```
<nazwa_klasy_publicznej>.class
```

W jednym pliku skompilowanym znajduje się tylko jedna klasa

W wielu klasach składających się na program mogą być zdefiniowane metody **main**, ale wykonanie programu rozpocznie się od metody **main** klasy wskazanej interpreterowi

Kompilator javy nie generuje kodu wynikowego, ale kod maszyny wirtualnej, który jest niezależny od procesora i systemu operacyjnego, w którym będzie wykonywał się program

Do wykonania programu w Javie potrzebny jest program maszyny wirtualnej oraz szereg skompilowanych klas standardowych tworzących razem środowisko wykonania Javy (ang. *Java Runtime Environment – JRE*)

Do kompilowania programów Javy potrzebny jest kompilator oraz dodatkowe komponenty tworzące razem zestaw uruchomieniowy Javy (ang. *Java Development Kit – JDK*)

Aby skompilować klasę należy wywołać kompilator podając nazwę pliku źródłowego z rozszerzeniem Java

```
javac <nazwa_klasy>.java
```

Kompilacja:

```
javac Program.java
```

Wykonanie

```
java Program
```

W Javie nie rozróżnia się plików nagłówkowych i źródłowych. Nie stosuje się też rozdzielania deklaracji od definicji klas – wszystkie metody definiuje się w deklaracji klasy.

Kompilator w celu sprawdzenia poprawności odwołania do składowych innej klasy musi mieć dostęp do jej pliku skompilowanego.

Wykonanie programu w maszynie wirtualnej jest wolniejsze niż w maszynie rzeczywistej – dlatego nowe środowiska wykonania wprowadzają mechanizm JIT (*just-in-time Compiling*) polegający na wykonywaniu translacji z postaci kodu maszyny wirtualnej do kodu maszynowego w momencie gdy dana metoda po raz pierwszy zostanie użyta w wykonywanym programie. W ten sposób łączy się niezależność programu skompilowanego od lokalnego środowiska z efektywnością wykonania.

Plik zawierający klasę publiczną musi mieć nazwę taką samą jak ta klasa – uwzględniając różnicę pomiędzy małymi i dużymi literami (nawet jeśli SO ich w nazwach plików nie rozróżnia).

Plik źródłowy Javy może zawierać tylko jedną klasę publiczną (o nazwie takiej jak nazwa pliku) i dowolną liczbę klas prywatnych – dostępnych tylko w pliku

Kompilacja pliku źródłowego powoduje utworzenie tylu plików wynikowych ile jest klas w pliku źródłowym.

W przypadku braku klasy, do której odwołuje się plik źródłowy w tym pliku kompilator poszukuje odpowiedniego pliku źródłowego w tej samej kartotece i jeśli znajdzie plik źródłowy to kompiluje go (uwzględniając zależności czasowe).

Programowanie obiektowe

Klasy definiuje się podobnie jak w C++

Nie ma zmiennych obiektowych a jedynie zmienne referencyjne (odpowiadające wskaźnikom w C++ - jednakże nie można na wskaźnikach wykonywać operacji)

Aby zmienna referencyjna wskazywała na konkretny obiekt należy jej przypisać referencję do obiektu - np. zwracaną przez **new**.

```
class X
{
    public int i;
}

X    x1, x2;

x1 = new X();
x2 = new X();
x2.i = 20;
x1 = x2;
x1.i = 10;

System.out.println( x1.i );    // wynik: 10
System.out.println( x2.i );    // wynik: również 10
                                //          - inaczej niż w C++;
```

Wszystkie funkcje Javy muszą być metodami klas. Funkcje stojące w C++ poza klasami muszą stać się komponentami statycznymi pewnych klas, w szczególności klas grupujących funkcje wg ich charakteru, przeznaczenia, tak jak biblioteki w C/C++.

Wszystkie zmienne globalne muszą być również polami (statycznymi) klas

Tworzenie i usuwanie klas

Ponieważ nie ma zmiennych obiektowych a jedynie zmienne referencyjne to każdy obiekt używany w programie musi być zaalokowany dynamicznie (operatorem **new** - podobnie jak w C++)

Nie ma operatora **delete** – obiekty kasowane są automatycznie, gdy tylko nie referuje do nich żaden istniejący obiekt. W tym celu system zarządzający wykonaniem programu utrzymuje licznik referencji do każdego obiektu i uznaje obiekt za gotowy do usunięcia, gdy licznik ten zostanie wyzerowany.

Fizyczna likwidacja obiektów w obszarze składu jest wykonywana przez „odśmiecacz” (ang garbage collector). Strategia odśmiecania stara się zoptymalizować ten proces ze względu na szybkość działania programu. Nie ma pewności w którym momencie moduł przystąpi do odśmiecania i kiedy faktycznie obiekt który stał się zbędny będzie usunięty.

W związku z tym nie ma też potrzeby tworzenia destruktorów dla zwalniania komponentów składowych klasy.

Odpowiednikiem destruktora jest metoda o nazwie **finalize**, która będzie wywołana automatycznie przed zniszczeniem obiektu. Jednakże, ponieważ programista nie wie w którym momencie obiekt zostanie faktycznie zniszczony to skutki zastosowania tej metody są nieprzewidywalne.

W przypadku klas używających zasobów, które powinny być zwolnione w momencie kontrolowanym przez programistę należy napisać własną metodę i wywołać ją jawnie.

Przekazywanie parametrów:

Parametry przekazywane są metodzie zawsze przez wartość - w przeciwieństwie do C++ gdzie przekazanie może być przez wartość lub zmienną (tam nazywaną referencją).

Ponieważ jednak można przekazać przez wartość referencję dla typów obiektowych i tablic to wewnątrz wywołanej metody można zmienić obiekty, do których referencje zostały przekazane.

public static void MetodaX(KlasaX obiekt)

oznacza przekazanie parametru przez referencję – a nie jak w takiej sytuacji w C++ przez wartość (gdzie obiekt był kopiowany)

Parametr **obiekt** jest referencją i może być zmieniany wewnątrz metody, ale nie ma to wpływu na referencję, która była parametrem aktualnym. Jednak, jeśli referencji użyjemy do zmiany pól referowanego obiektu, to skutek ten będzie trwały tzn. będzie istniał po zakończeniu wywołania metody.

Zasady ogólne związane z przekazywaniem parametrów:

- metoda nie może zmodyfikować parametru typu prymitywnego (liczb, znaków i wartości logicznych)
- metoda może zmieniać stan obiektu, którego referencja jest parametrem aktualnym

- metoda nie może sprawić, aby referencja będąca parametrem aktualny zaczęła wskazywać na inny obiekt

Zwracanie obiektu jako wartości funkcji

Funkcja nie może zwrócić - jak w C++ obiektu. Może jedynie utworzyć obiekt dynamicznie i zwrócić referencję do niego.

```
public
class Punkt
{
    int x, y;
    void ustawWspolrzedne(int wspX, int wspY)
    {
        x = wspX;
        y = wspY;
    }
    Punkt kopiuj()
    {
        Punkt punkt = new Punkt();
        punkt.x = x;
        punkt.y = y;
        return punkt;
    }
}
```

Programista nie musi dbać o skasowanie obiektu tymczasowego utworzonego do zwrócenia wartości funkcji - zrobi to za niego odśmieczacz gdy tylko do obiektu nie będzie dostępu.

Inicjalizacja pól obiektu:

Konstruktor domyślny – jeśli nie jest określony jawnie powoduje wyzerowanie wszystkich składowych obiektu.

Konstruktor domyślny jest tworzony przez kompilator automatycznie tylko jeśli nie określono jawnie żadnego innego konstruktora z parametrami

Przy wykonaniu konstruktora wszystkie pola obiektu nie inicjalizowane jawnie wypełniane są wartościami zerowymi (0, null, false)

W Javie nie ma elementu składniowego odpowiadającego liście inicjalizacyjnej. W C++ istniała konstrukcja listy inicjalizacyjnej, aby umożliwić nadawanie wartości polom stałym oraz wywoływać konstruktory dla pól będących obiektami. W Javie nie ma takiej potrzeby, bo polom **final** można nadawać wartości w konstruktorze a wszystkie obiekty składowe i tak muszą być tworzone przez referencje

Inicjalizacja pól obiektu wartościami niezerowymi:

- a) poprzez podanie stałych wartości inicjalizujących w deklaracji klasy

```
class X
{
```

```
        private x = 20;
    }
```

- b) w konstruktorze
- c) w tzw. bloku inicjalizacyjnym

```
class X
{
    private int x;
    private double y;

    static private double z;
    ...

    public X()
    {
    }
    ...

    // Blok inicjalizacyjny dla obiektu
    {
        x = 10;
        y = 3.6;
    }

    // Blok inicjalizacyjny dla komponentów statycznych
    static
    {
        z = 55.7;
    }
}
```

Kolejność wykonywania operacji inicjalizacyjnych:

- blok inicjalizacji statycznych – wykonywany tylko jeden raz gdy klasa jest po raz pierwszy użyta w programie,
- wykonanie przypisań określonych w deklaracjach pól,
- wykonanie bloku inicjalizacyjnego,
- wykonanie konstruktorów.

Pole this

Pole **this** istnieje podobnie jak w C++ w każdym obiekcie, ale nie jest wskaźnikiem tylko referencją do obiektu zawierającego to pole.

Słowo kluczowe **this** może być wykorzystane do wywołania innego konstruktora, ale tylko wtedy gdy wywołanie to jest pierwszą linią innego konstruktora:

```
class X
{
    public X( String x )
    { ... };
}
```

```
public X( double d)
{
    this("ala");
    ...
};
}
```

Stałe

Stałe deklaruje się poprzedzając je słowem **final** (odpowiednik **const** w C++).

Dziedziczenie

Dziedziczenie od innej klasy wskazujemy za pomocą słowa kluczowego **extends**

```
class Pochodna extends Bazowa
{
}
}
```

Dziedziczenie w Java jest jedynie publiczne - nie można modyfikować praw dostępu do komponentów klasy bazowej.

Aby odwoływać się do przesłoniętych komponentów klasy od której dziedziczymy używany prefiksu **super**.

```
class Bazowa
{
    private    double    y;

    public double    GetParam()
    {
        return    y;
    }
    public Bazowa( double y )
    {
        this.y = y;
    }
}

class Pochodna extends Bazowa
{
    private    double x;

    public double GetParam()
    {
        return    x + super.GetParam();
    }

    Pochodna( double x )
}
```

```

    {
        super( x + 10.5 ) // Wywołanie konstruktora nadklasy
        this.x = x;
    }
}

```

Jest to odpowiednik konstrukcji **Bazowa::GetParam()** z C++.

Słowa kluczowego **super** należy użyć również gdy zamierzamy w konstruktorze klasy pochodnej wywołać konstruktor klasy bazowej. W C++ w takiej sytuacji umieszcza się wywołanie konstruktora klasy bazowej w liście inicjalizującej.

Jeśli klasa pochodna nie wywołuje w konstruktorze jawnie konstruktora klasy bazowej to przed rozpoczęciem wykonania konstruktora klasy pochodnej wywoływany jest konstruktor domyślny klasy bazowej. Jeśli taki konstruktor nie jest zdefiniowany to wystąpi błąd kompilacji.

Do zmiennej referencyjnej klasy bazowej można podstawiać referencje na obiekty klas pochodnych (podobnie jak ze wskaźnikami w C++).

W Javie nie ma potrzeby deklarowania metod wirtualnych, ponieważ każda metoda jest z założenia wirtualna (inaczej niż w C++).

Wspólna nadklasa Object

Wszystkie klasy dziedziczą od wspólnej nadklasy **Object**

Również typy tablicowe dziedziczą od klasu **Object**

```

B[]      tabB;
Object[] tabO;

tabB = new P[10];
tabO = tabB;
tabO[1] = new B();
tabO[2] = new X(); // BŁĄD wykonania bo teraz
                  // tabO wskazuje na B[]

```

W klasie **Object** zdefiniowana jest metoda **toString**, **clone** i **equals**

Metoda **clone** odpowiada konstruktorowi kopiującemu w C++
Metody te wymagają przeddefiniowania w klasach pochodnych.

Zmienna referencyjna do klasy **Object** może przechowywać referencję do dowolnej klasy (odpowiednik wskaźnika (void *) w C++

```

class A
{
    ...
}

```

```

class B
{
    ...
}

Object x, y;
x = new A();
y = new B();

A a;
a = (A)x;    // OK bo x wskazuje na A
a = (A)y;    // BŁĄD wykonania - ale nie kompilacji

```

Klasy opakujące

Aby można było wykorzystywać metody działające na referencjach dla typów prostych (**int**, **double**, **float**, **char**, **boolean**) wprowadzono klasy opakujące **Integer**, **Double**, **Float**, **Character**, **Boolean**

Klasy opakujące dostarczają dodatkowo metod konwertujących łańcuchy na wartości

```

class Integer
{
    public          int      intValue();
    public          String   toString( int i );
    public static Integer  valueOf( String s );
    public static int     parseInt( String s );
}

```

Klasa Class

Obiekt klasy **Class** zawiera informacje o klasie.

Dane w obiekcie klasy **Class** wiążą obiekt binarny istniejący w trakcie wykonania programu z jego opisem w języku źródłowym.

Obiekt klasy **Class** może być uzyskany za pomocą metody **getClass()** klasy **Object**.

Ponieważ **Class** jest nadklasą każdej innej klasy metodę **getClass** można wywołać dla każdego obiektu.

```

class Bazowa
{
}

class Pochodna extends Bazowa
{
}

```

```
B b = new Pochodna();
Class cl = b.getClass();
```

Metoda `getName` klasy `Class` zwraca nazwę klasy użytą w programie:

```
System.out.println( cl.getName() ); // Wynik: Pochodna
```

Referencję do obiektu `Class` opisującego klasę można uzyskać korzystając z pseudoatrybutu statycznego `class`:

```
System.out.println( Bazowa.class.getName() );
```

Dla każdej klasy tworzony jest jeden obiekt typu `Class` - stąd można porównywać referencje:

```
if ( b.getClass() == Bazowa.class )
    System.out.println( "To klasa Bazowa" );
else
    System.out.println( "To klasa inna niż Bazowa" );
```

Klasa `Class` dostarcza metody `newInstance()`, która tworzy obiekt klasy opisywanej przez obiekt `Class`, wywołuje dla niego konstruktor domyślny i zwraca referencję do utworzonego obiektu.

```
Bazowa b = new Pochodna();
```

```
Object o = b.getClass().newInstance();
// stworzony zostanie obiekt klasy Pochodna
```

```
System.out.println( o.getClass().getName() );
// Wynik: Pochodna
```

Polimorfizm

Z założenia wszystkie metody klasy są polimorficzne - nie trzeba umieszczać słowa `virtual` przed definicją metody

```
class B
{
    void f()
    {
        System.out.println( "Z klasy bazowej" );
    }
}

class P1 extends B
{
    void f()
```

```

    {
        System.out.println( "Z klasy P1" );
    }
}

public
class Main
{
    public static void main (String args[])
    {
        B b = new P1();

        b.f(); // Wywołane zostanie f z P1
    }
}

```

Jeśli zamierzamy wymusić wczesne wiązanie metody z jej identyfikatorem to należy w klasie bazowej przed definicją metody umieścić słowo **final**. Jednakże uniemożliwia to przesłonięcie metody z klasy bazowej w klasie pochodnej.

```

class B
{
    final void f()
    {
        System.out.println( "Z klasy bazowej" );
    }
}

class P1 extends B
{
    void f() // BŁĄD - nie można przysłonić metody finalnej
    {
        ...
    }
}

```

Jeśli klasa nie definiuje wszystkich swoich metod to jest abstrakcyjna i należy poprzedzić ją słowem kluczowym **abstract**

Jeśli metoda nie jest zdefiniowana w klasie abstrakcyjnej to należy ją również poprzedzić słowem kluczowym **abstract**.

Interfejsy

Java nie obsługuje dziedziczenia wielokrotnego (tzn. klasa może bezpośrednio dziedziczyć tylko od jednej klasy).

W miejsce tego Java dostarcza mechanizmu interfejsu.

```

public interface Int1

```

```
{
    double f1( int i );
}
```

Klasa może implementować metody zadeklarowane w interfejsie. Można wskazać wiele interfejsów, które klasa implementuje.

```
class ImplInt1 implements Int1, Int2
{
    public int f1() { return 1;} // Metoda z interfejsu Int1
    public int f1() { return 1;} // Metoda z interfejsu Int2
}
```

Jeśli metoda nie implementuje pewnej metody z zadeklarowanego w jej nagłówku interfejsu to jest abstrakcyjna i musi być poprzedzona słowem **abstract**.

Interfejsy mogą dziedziczyć od innych interfejsów, również od wielu:

```
public interface Int2 extends Int0, Int1
{
    double f2( int i );
}
```

Interfejs w znaczeniu Javy jest czystą klasą abstrakcyjną która:

- nie implementuje żadnej z metod,
- posiada wszystkie metody publiczne,
- jeśli posiada zmienne to są one **public static final**.

Można tworzyć zmienne referencyjne dla interfejsów. Zmienna taka może wskazywać na obiekt klasy która implementuje interfejs.

```
Int1 oi = new ImplIn1();
```

Rzutowanie i konwersja typów

W Javie sprawdzanie dopuszczalności konwersji wykonywane jest i na etapie kompilacji i na etapie wykonania

```
class B
{
}

class P extends B
{
}

B[] tabB = new B[10];
```

```
tabB[1] = new P();
```

```
P p1 = (P)tabB[1]; // OK bo tabB[1] wskazuje na P  
P p2 = (P)tabB[2]; // BŁĄD wykonania - ale nie kompilacji
```

Tworzenie kopii obiektów

Ponieważ nie ma zmiennych obiektowych (tak jak w C++) a tylko zmienne referencyjne to kopiowanie zmiennych nie tworzy kopii Obiektu

```
class X  
{  
}
```

```
X x1, x2;
```

```
x1 = new X();  
x2 = x1 // W C++ obiekt zostałby skopiowany za pomocą  
        // operatora podstawienia - domyślnego lub  
        // zdefiniowanego w klasie X  
        // W Javie następuje tylko skopiowanie referencji -  
        // obydwie zmienne wskazują na ten sam obiekt.
```

Aby stworzyć kopię obiektu należy do tego celu napisać własną metodę.

W klasie Object od której dziedziczy każda inna klasa zdefiniowana jest metoda **clone()**

```
Object clone();
```

clone zwraca referencję do obiektu będącego kopią tego, dla którego zawołano metodę.

clone jest metodą chronioną - można ją wywoływać tylko z wnętrza kodu metod danej klasy.

Domyślna implementacja **clone()** w klasie **Object** tylko kopiuje zawartość poszczególnych pól obiektu. Jeśli obiekt zawiera referencje do swoich lokalnych składowych taki sposób kopiowania jest niewystarczający - należy utworzyć kopię składowych.

W tym celu należy w napisać własną wersję metody clone().

Aby można było wywoływać clone klasa musi implementować interfejs Cloneable (zdefiniowany w standardowych pakietach Javy). Aby można było wykorzystywać kopiowanie z zewnątrz klasy należy przedefiniować metodę jako publiczną i dodatkowo zwracającą referencję to tej klasy dla której została wywołana

```
class X implements Cloneable  
{  
    public X clone()  
    {  
        X x = super.clone();
```

```

        // wykonaj dodatkowe operacje na x
        return x;
    }
}

```

Ponieważ wynik jest referencją na **Obiekt** przed jej użyciem należy dokonać rzutowania na typ faktyczny.

```
X x1, x2;
```

```
x1 = new X();
x2 = (X)x1.clone();
```

Klasy wewnętrzne

Klasy można definiować nie tylko na poziomie pliku źródłowego ale też:

- wewnątrz klas (jako prywatne lub publiczne komponenty innych klas)
- wewnątrz metod innych klas

Klasy takie nazywane są *klasami wewnętrznymi*

```

class Zewn
{
    private i_zewn;

    public class Wewn
    {
        public Wewn( int i )
        {
            i_wewn = i_zewn + i;
        }
        public int podajSumeI()
        {
            return i_wewn + i_zewn;
        }
    }

    private Wewn wew;
    public Zewn()
    {
        i_zewn = 6;
        wew = new Wewn( 5 );
    }
}

```

Klasa wewnętrzna zdefiniowana wewnątrz innej klasy ma dostęp do zmiennych prywatnych i publicznych klasy zewnętrznej.

W przypadku, gdy obiekt klasy wewnętrznej jest alokowany wewnątrz kodu metod klasy zewnętrznej należy przy tworzeniu obiektu wskazać na obiekt klasy zewnętrznej, do której klasa wewnętrzna będzie miała dostęp.

```
Zewn zew;  
Zewn.Wewn wew;  
  
zew = new Zewn();  
wew = zew.new Wewn(7);
```

Klasa wewnętrzna metody ma dostęp do zmiennych lokalnych metody zadeklarowanych z kwalifikatorem **final**.

Tablice

Java dopuszcza deklarowanie referencji do tablic wielowymiarowych:

```
int x[][];
```

w przeciwieństwie do C++ nie można deklarować tablicy

```
int x[5][7];
```

a jedynie referencje które należy zainicjować np. tworząc tablicę za pomocą **new**

```
x = new int [5][7];  
lub  
x = new int [5][]  
for (int i=1; i<5; i++)  
    x[i] = new int [7];
```

Pakiety

Odpowiednikiem biblioteki w innych językach programowania jest pakiet

Pakiet ma budowę hierarchiczną – pliki zawierające skompilowane klasy ułożone są w strukturze katalogu - pakiet zawiera podpakiety

Klasy w różnych gałęziach pakietu mogą mieć powtarzające się nazwy.

Klasę można jednoznacznie zidentyfikować podając pełną ścieżkę do klasy np. dla klasy **Data** zawartej w podpakiecie **util** pakietu **java**

```
java.util.Data dzisiaj = new java.util.Data();
```

Dla skrócenia zapisu można użyć dyrektywy import

```
import <ścieżka do pakietu>. *;
```

lub

```
import <ścieżka do pakietu>.<nazwa klasy>;
```

Pierwsza wersja importuje wszystkie klasy ze wskazanego pakietu, druga tylko jedną wskazaną klasę.

Druga wersja może być użyta do unikania przedrostka klasy przy odwołaniach do jej komponentów statycznych.

Druga wersja pozwala również na uniknięcie niejednoznaczności, gdy dwa zaimportowane pakiety zawierają tę samą klasę. Wtedy dodatkowo zaimportowana klasa przysłania klasy we wcześniej zaimportowanych pakietach:

```
import java.util.*;           // zawiera klasę Date
import java.sql.*;           // zawiera również klasę Date
import java.util.Date;
```

```
Date data = new Date(); // będzie użyta klasa java.util.Date;
```

Wszystkie standardowe klasy Javy znajdują się w pakietach "java" i "javax"

Odnajdowanie pakietów:

Pakiety poszukiwane są w:

- standardowych pakietach javy (w lokalizacji jre/lib/rt.jar),
- lokalizacjach wskazanych przez zmienną środowiskową CLASSPATH lub przez parametr kompilatora lub interpretera o takiej samej nazwie. Kolejne ścieżki w tej zmiennej oddzielane są znakiem ;

Do ścieżek z CLASSPATH dołączane są względne ścieżki określone w dyrektywie import. Ścieżka w CLASSPATH może wskazywać nie kartotekę ale konkretny plik archiwum (plik z rozszerzeniem **jar** - *Java ARchive*). Plik ten jest zwykłym archiwum w formacie ZIP i można go przeglądać dowolnym archiwizatorem obsługującym ten format.

Przykład:
w pliku źródłowym

```
import x.y.*
```

```
MyClass cx;
```

```
javac -classpath /home/użytkownik/klasy;./home/użytkownik/archiwa;/home/my/3d.jar
```

Importowany pakiet będzie poszukiwany w lokalizacjach:

- /home/użytkownik/klasy/x/y
- ./x/y
- /home/użytkownik/archiwa/x/y
- w podkatalogu x/y/ w archiwum /home/my/3d.jar

Klasa **MyClass** będzie poszukiwana:

- w aktualnym pliku
- w aktualnym katalogu

- w importowanych pakietach.

Wyjątkiem jest sytuacja, gdy potrzebna klasa jest osobno importowana dyrektywą import.

Względy efektywności

Nie można używać zmiennych obiektowych - zawsze należy obiekt alokować. Alokowanie i automatyczne zwalnianie obiektu jest nieco bardziej czasochłonne niż jego alokacja na stosie.

Wszystkie metody niestaticzne i nieoznaczone jako **final** są wirtualne - wołanie metody wirtualnej jest nieco bardziej czasochłonne niż dla metody o wczesnym wiązaniu.

Nie ma możliwość zażądania skasowania obiektu w ustalonej chwili. Dopóki jest gdziekolwiek referencja do tego obiektu będzie on utrzymywany na składzie.

Nie można przewidzieć kiedy odśmiecacz podejmie swoją akcję. Działanie takie spowolni w trudno przewidywalnym momencie działania programu - stąd zastosowania Javy do budowy krytycznych czasowo aplikacji działających w czasie rzeczywistym jest kłopotliwe.

Niejednorodności i niekonsekwencje

Java dla niektórych klas wbudowanych (klasy opakowujące, **String**) dokonuje konwersji typów pomimo, że programista nie ma takiej możliwości, np. można napisać

```
String s = "ala" + 10;
```

Dla klasy String przeciążono zatem operator +

Problemy wynikające z dostępu do pól klasy zewnętrznej z klasy wewnętrznej znacznie komplikują język i dają możliwość ingerencji w zmienne prywatne

Dziwaczna konstrukcja:

```
obiekt_klasy_zewnetrznej.new KlasaWewnetrzna()
```

występujący przy tworzeniu obiektów klas wewnętrznych – trudno zakwalifikować do jakiego rodzaju bytów programistycznych należy operator **new** – przecież w Javie nie można definiować operatorów

Składnia klas anonimowych znacznie zmniejsza czytelność kodu