
PROGRAMOWANIE W JAVIE

Andrzej Marciniak

Uniwersytet Zielonogórski
Instytut Sterowania i Systemów Informatycznych

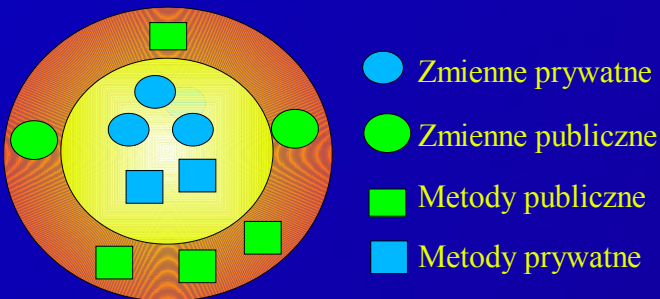
Programowanie obiektowe w Javie:

- ❑ Koncepcje programowania obiektowego
- ❑ Klasy
- ❑ Obiekty
- ❑ Pola danych i metody
- ❑ Klasy zagnieżdżone
- ❑ Usuwanie obiektów

Koncepcje programowania obiektowego

Obiekt jest programowym zestawem powiązanych zmiennych i metod . Zmienne przechowują informacje dotyczące stanu modelowanych obiektów lub procesów świata rzeczywistego, a metody definiują ich zachowanie.

Powszechnie stosowana reprezentacja wizualna obiektu programistycznego



Główne mechanizmy programowania obiektowego

Języki zorientowane obiektowo zawierają następujące mechanizmy wymuszające stosowanie obiektów:

- enkapsulacja (ang. *encapsulation*),
- dziedziczenie (ang. *inheritance*),
- polimorfizm (ang. *polymorphism*).

Mechanizmy te funkcjonują w Javie bardzo podobnie jak w C++.

Enkapsulacja

Enkapsulacja polega na łączeniu danych i instrukcji, które wykonują na nich działania, przez umieszczanie ich we wspólnych obiektach.

Środkiem do osiągnięcia enkapsulacji w Javie są klasy (ang. *classes*).

Klasa stanowi model abstrakcyjny pewnej grupy obiektów wyróżniających się tą samą strukturą i zachowaniem i jest modułem posiadającym: nazwę i atrybuty w postaci pól danych i metod. Zatem obiekt (ang. *object*) to pojedynczy egzemplarz klasy.

Obiekty nazywa się nieraz egzemplarzami klas (ang. *instances of classes*). Dane należące do klasy i przechowujące informacje o stanie każdego obiektu określa się zmiennymi egzemplarzowymi (ang. *instance variables*). Wykonywane zadania i sposób dostępu do danej klasy określają jej metody.

Enkapsulacja i klasy

Celem enkapsulacji jest zmniejszenie stopnia złożoności programu poprzez możliwość ukrycia szczegółów dotyczących funkcjonowania klasy przez zadeklarowanie ich jako prywatne. Natomiast elementy tworzące interfejs klasy deklaruje się jako publiczne.

Definicja klasy jest jedynym sposobem zdefiniowania nowego typu danych w Javie.

Posługując się pojęciami klasy, programista może w wygodny i elegancki sposób definiować różnorodne typy danych wykorzystując:

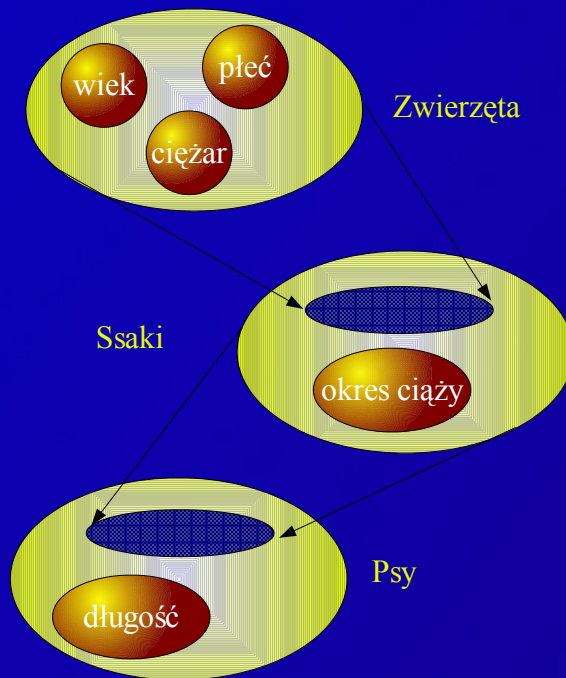
- strukturę hierarchiczną deklaracji klas (kompozycja),
- prefiksowanie klas tzw. "dziedziczenie", umożliwiające tworzenie hierarchii typu: ogólny - bardziej szczegółowy.

Dziedziczenie

Większość ludzi czuje potrzebę uporządkowania obiektów świata rzeczywistego poprzez tworzenie złożonych taksonomii (klasyfikacji, kategoryzacji). W przypadku gdy pewne klasy tworzą abstrakcyjne, wspólne modele opisu (np. klasa Zwierzęta), wówczas na ich podstawie można utworzyć podklasy (np. klasa Ssaki) będące w relacji zawierania się w klasach nadrzędnych. Opis ssaków zawierać może dodatkowe informacje dotyczące ich charakterystycznych cech.

Ponieważ ssaki są dokładniej opisanymi zwierzętami to dziedziczą wszystkie ich atrybuty. Klasę zwierząt nazwiemy nadklasą (ang. *superclass*), natomiast klasę ssaków podklasą (ang. *subclass*).

Dziedziczenie - przykład



Polimorfizm

Aby wykonać dwa różne zadania w większości języków programowania trzeba utworzyć dwie funkcje o różnych nazwach. Polimorfizm to możliwość tworzenia wielu metod o takiej samej nazwie, zgodnie z zasadą: *jeden przedmiot, wiele kształtów*.

Polimorfizm pozwala tworzyć wiele implementacji tej samej metody, co w informatyce nazywa się jej przeładowywaniem (ang. *overloading*). Wybór implementacji metody jest zależny od przekazywanych jej parametrów.

Polimorfizm czasu przebiegu

Przeładowywanie metod umożliwia ujednolicenie działań wykonywanych na danych różnych typów i jest najbardziej przydatne podczas tworzenia małych klas, gdyż ma charakter statyczny (trzeba przewidzieć wszystkie typy danych na których trzeba będzie wykonywać działania).

Czasami potrzebne jest rozwiązanie przewidujące możliwość zmiany implementacji danej metody w podklasie; wówczas jej wersja zdefiniowana w nadklasie może w ogóle nie wykonywać żadnego działania. Tego typu polimorfizm nazywa się polimorfizmem czasu przebiegu (ang. *runtime polymorphism*).

Interfejsy

Interfejs (ang. *interface*) jest "urządzeniem" które pozwala różnym obiektom współpracować ze sobą (analogicznym do protokołu). Słowo kluczowe *interface* definiuje kolekcję definicji metod oraz wartości stałych, które mogą być później wykorzystane przez inne klasy, przy użyciu słowa *implements*.

Interfejs klasy definiuje jej protokół zachowań (dostępu do klasy), który może być zaimplementowany przez dowolną klasę w dowolnym miejscu w hierarchii klas. Interfejsy są użyteczne do

- wychwytywania podobieństw między nie powiązаныmi klasami
- deklarowania metod, które mają być zaimplementowane w jednej lub większej liczbie klas
- odsłaniania interfejsu programistycznego obiektu bez ujawniania jego klasy

Tworzenie klas i obiektów

Definicja klasy przyjmuje następującą formę (modyfikatory zostaną omówione później):

```
[modyfikatory] class NazwaKlasy [ extends NazwaNadklasy]  
[implements NazwyInterfejsów] {
```

Ciało klasy:

*Tutaj znajdują się definicje pól danych , metod
i klas wewnętrznych klasy
}*

Elementy deklaracji pomiędzy nawiasami [i] są opcjonalne.

Tworzenie klas i obiektów - przykład

```
public class Punkt {  
    public int x = 0;  
    public int y = 0;  
    //Konstruktor!  
    public Punkt(int x, int y) {  
        this.x = x;this.y = y;  
    }  
    public ustaw(int x, int y){  
        this.x = x;this.y = y;  
    }  
}  
//...  
Punkt centrum = new Punkt(23, 94);
```

Deklaracja zmiennej *centrum* nie powoduje utworzenia obiektu - jest on tworzony dopiero za pomocą operatora `new` i konstruktora. Do tego czasu *centrum* jest pustą referencją (ang. *null reference*).

Pola danych i metody

```
public class Punkt {  
    public int x = 0;  
    public int y = 0; ... }
```

Pola danych są atrybutami klasy, pełniącymi rolę podobną do zmiennych lub stałych. Są one deklarowane na tych samych zasadach, co zmienne lokalne, wg. schematu

modyfikatoryPola TypPola NazwaPola;

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

Metody są modułami programowymi przypominającymi funkcje z języka C++. Każda funkcja w Javie jest związana z definicją klasy (spełnia rolę jej metody) i deklarowana jest wg. schematu:

```
modyfikatory TypRezultatu NazwaMetody(ListaParametrówFormalnych) {  
    //treść metody  
}
```

Uwagi dotyczące metod

- Jeśli celem wykonania nie jest uzyskanie rezultatu przekazywanego przez instrukcję `return`, to w deklaracji typu metody występuje słowo `void`.
- Jeśli metoda nie ma żadnych argumentów lista jest pusta. Odmienne niż w C czy C++ parametr metody nie może być typu `void`.
- Nie ma możliwości definiowania wartości domyślnych parametrów.
- Implementacje wszystkich metod klasy muszą znajdować się w tym samym miejscu co jej deklaracja. Wynika z tego że klasa w całości musi być zdefiniowana w jednym pliku.
- Jak to wcześniej zostało stwierdzone, metody mogą być przeciążane.

Modyfikatory klas, metod i pól

W Javie modyfikatory możemy podzielić na dwa rodzaje:

- modyfikatory dostępu wpływające na reguły widoczności i umożliwiające kontrolę dostępu do pól danych i metod klasy z innych klas,
- modyfikatory właściwości modyfikowanego elementu.

Modyfikatory dostępu

Modyfikatory dostępu:

- public - wszystkie klasy mają dostęp do pól danych i metod public,
- private - dostęp do metod i pól danych posiadają jedynie inne metody tej samej klasy,
- protected - metoda lub pole danych protected lub może być używana jedynie przez metody swojej klasy oraz metody wszystkich jej klas pochodnych,
- package - jest to modyfikator domyślny, wszystkie metody i pola danych bez modyfikatora dostępu traktowane są jako typu package. Metody (lub pola danych) typu package mogą być używane przez inne klasy danego pakietu.

Poziomy dostępu

| Modyfikator | klasa | podklasa | pakiet | wszędzie |
|-------------|-------|----------|--------|----------|
| private | X | | | |
| protected | X | X | X | |
| public | X | X | X | X |
| package | X | | X | |

Modyfikatory właściwości

- pól danych
 - static,
 - final,
 - transient,
 - volatile.
- klas
 - public,
 - final,
- abstract,
- metod
 - abstract,
 - static,
 - final,
 - synchronized,
 - native;

Statyczne pola danych i metody

Deklaracja ze słowem `static` oznacza, że pole danych lub metoda dotyczy klasy, a nie obiektu, tzn. dla wszystkich obiektów danej klasy pole statyczne ma tę samą wartość.

Przykład. Mamy klasę opisującą rachunek bankowy. Dla wszystkich rachunków jednego typu oprocentowanie wkładów jest jednakowe, więc oprocentowanie rachunku zadeklarowane zostało jako pole statyczne aby w razie zmiany oprocentowania nie zmieniać jego wartości dla wszystkich obiektów tej klasy.

```
class KontoOsobiste{  
    static byte oprocentowanie = 10;  
    private Osoba Wlasciciel;  
    // tu deklaracje innych pól danych klasy  
    static void ZmienOprocentowanie(byte nowyProcent) {  
        oprocentowanie = nowyProcent; }  
}
```

Odwołania do metod i pól statycznych

Metody statyczne podobnie jak statyczne pola danych są przypisane do klasy, a nie konkretnego obiektu i służą do operacji tylko na polach statycznych.

```
public static void main(String[] args)
{
    //Obiekt typu KontoOsobiste jeszcze nie istnieje
    KontoOsobiste.Oprocentowanie = 20;

    KontoOsobiste rachunek = new KontoOsobiste();// utworzenie obiektu

    // odwołanie do pola statycznego poprzez obiekt powoduje zmianę
    rachunek.ZmienOprocentowanie(40); //oprocentowania wszystkich obiektów
    // odwołanie do metody statycznej poprzez nazwę klasy
    KontoOsobiste.ZmienOprocentowanie(30);
}
```

Statyczne pola danych mogą być inicjalizowane.

Odwołania do obiektów

Deklaracja zmiennej typu obiektowego przyjmuje postać podobną do deklaracji zmiennej typu wewnętrznego, z tym, że w wyrażeniu nadającym wartość początkową zmiennej może zostać utworzony nowy obiekt, np.

```
KontoOsobiste konto=new KontoOsobiste();
```

lub

```
KontoOsobiste konto; konto=new KontoOsobiste();
```

Odwołania do obiektów przypominają wskaźniki z C/C++, z tą różnicą że nie można na nich wykonywać operacji arytmetycznych. Bezpieczeństwo Javy wynika właśnie z braku możliwości utworzenia odwołania, które by wskazywało na dowolny fragment pamięci.

Odwołania do obiektów danej klasy są kompatybilne z odwołaniami do obiektów wszystkich jej podklas. Ponadto, wszystkie typy wewnętrzne mają swoje odpowiedniki obiektowe (np.: typ `int` ma odpowiadający mu typ obiektowy `Integer`) i jako obiekty mają wiele konstruktorów i metod.

Operator new

Operator new tworzy pojedynczy egzemplarz danej klasy i zwraca wartość odwołania do niego.

```
Punkt p=new Punkt();  
Punkt p2=p;
```

Zapisanie wartości zmiennej p w p2 nie powoduje zarezerwowania dodatkowej pamięci lub przekopiowania jakiegokolwiek części obiektu wskazywanego przez p. Istnieje tylko jeden obiekt i dwa odwołania.

Dostęp do składowych

Dostęp do atrybutów obiektu, reprezentowanych przez zmienną obiektową, realizujemy za pomocą wyrażeń z operatorem dostępu (kropki) postaci:

```
Punkt p1=new Punkt();  
p1.x=10;      //dostęp do pola danych  
p1.wyswietl(); //dostęp do metody  
int zmienna_x=new Punkt().x;
```

Zauważ, że w ostatniej linii tworzony jest obiekt, którego wartość pola jest natychmiast pobierana, po czym sam obiekt nie jest dalej wskazywany przez żadną referencję.

Inicjator obiektów

Java pozwala na użycie nie tylko inicjatorów klas, ale także inicjatorów obiektów. Jeśli w klasie zdefiniowanych jest więcej inicjatorów obiektów, to wykonywane są one w kolejności wystąpienia, bezpośrednio po wywołaniu konstruktora nadklasy.

```
class MojaKlasa {  
    boolean sprawdzIniObj = false; // pole danych klasy:  
    MojaKlasa(){                    // konstruktor klasy MojaKlasa:  
        System.out.print("sprawdzIniObj = "+sprawdzIniObj);  
    }  
    // inicjator obiektów klasy:  
    {  
        sprawdzIniObj = true;  
    }  
}
```

Gdy będziemy tworzyć obiekt klasy `MojaKlasa`, to wykonanie konstruktora spowoduje wyświetlenie na ekranie napisu: `sprawdzIniObj = true`.

Konstruktory

Konstruktor zostaje wywołany podczas tworzenia nowego obiektu klasy. Każda klasa może posiadać wiele konstruktorów, różniących się listą argumentów. Ponieważ każda klasa w Javie dziedziczy z klasy *Object*, posiada też konstruktor bezparametrowy odziedziczony z tej klasy.

```
class MojaKlasa {  
    MojaKlasa(){ //pierwszy konstruktor  
        System.out.print("Konstruktor");  
    }  
    MojaKlasa(int a){...} //drugi konstruktor  
}  
...  
MojaKlasa zmienna=new MojaKlasa();
```

Zauważ odmienne niż w języku C++ odwołanie do konstruktora bezparametrowego!

Konstruktor kopiujący

W Javie nie używa się konstruktora kopiującego niejawnie i nie jest on tak często wykorzystywany jak w C++.

```
class MojaKlasa {  
    String dane;  
    MojaKlasa(){//...jakieś ciało  
    }  
    MojaKlasa(MojaKlasa x){  
        dane=new String(x.dane);  
    }  
    ...  
    MojaKlasa a=new MojaKlasa();  
    MojaKlasa zmienna=new MojaKlasa(a); //wywołanie kopiującego
```

Inicjowanie pól statycznych

Do inicjalizacji zmiennych statycznych wykorzystuje się zamiast konstruktorów tzw. inicjatory zmiennych statycznych. Inicjacja następuje wtedy, gdy klasa jest pierwszy raz ładowana do pamięci (gdy nie ma jeszcze żadnego obiektu danej klasy). Każda klasa może zawierać dowolną liczbę inicjatorów statycznych. Inicjatory statyczne wykonywane są w kolejności ich wystąpienia w definicji klasy.

```
class MojaKlasa {  
    static int licznik;  
    static byte flaga;  
    static  
    {    licznik=0;  
        flaga=1;  
    }  
}
```

Klasy zagnieżdżone

Klasa zagnieżdżona jest członkiem innej klasy. Klasy zagnieżdżone, mogą mieć modyfikator `static`, wskazujący że mają takie same właściwości jak klasa zewnętrzna. Jeśli nie są statyczne, wówczas określa się je jako wewnętrzne. Oprócz tego, klasy zagnieżdżone mogą być oczywiście opatrzone modyfikatorami: `private`, `protected` i `public` oraz `abstract` i `final` - a znaczenia tych modyfikatorów są takie same jak w przypadku zwykłych klas.

```
class EnclosingClass{
    . . .
    static class AStaticNestedClass { //zagnieżdżona statyczna
        . . .
    }
    class InnerClass { //wewnętrzna
        . . .
    }
}
```

Klasy wewnętrzne a statyczne zagnieżdżone

Klasy zagnieżdżone statyczne:

- nie mogą bezpośrednio odwoływać się do atrybutów klasy zewnętrznej (muszą używać kwalifikowanego odnośnika)
- obiekty tych klas mogą istnieć nawet gdy nie istnieją obiekty klas zewnętrznych,

Klasy wewnętrzne

- mają nieograniczony dostęp do atrybutów klasy zewnętrznej,
- obiekty tych klas istnieją tylko gdy istnieją obiekty klas zewnętrznych,
- są często wykorzystywane w mechanizmie obsługi zdarzeń AWT

Klasy wewnętrzne - przykład uchwytu


Założmy że do istniejącej klasy kontenerowej Stack chcemy dodać nową cechę - pozwolić innym klasom na zliczanie elementów na stosie wykorzystując interfejs `java.util.Enumeration`. Interfejs zawiera dwie deklaracje metod:

```
public boolean hasMoreElements();  
public Object nextElement();  
i definiuje interfejs dla pętli po elementach:  
while (hasMoreElements()) nextElement()
```

Jeżeli Stack zaimplementuje Enumeration w sobie, nie będzie można zliczyć zawartości stosu więcej niż raz (zostanie wyczyszczony), jak również zastosować dwóch wyliczeń równolegle. Musi istnieć klasa pomocnicza (adapter), w tym przypadku wewnętrzna, która ma dostęp do wszystkich elementów ponieważ klasa Stack wspiera tylko kolejki LIFO.

Klasy wewnętrzne - przykład uchwytu

```
public class Stack {  
    private Vector items; //przechowuje elementy stosu  
    ...//konstruktory i metody klasy Stack...  
  
    public Enumeration enumerator() { // zwraca obiekt klasy  
        return new StackEnum(); //implementującej interfejs Enumeration  
    }  
    class StackEnum implements Enumeration {  
        int currentItem = items.size() - 1;  
        public boolean hasMoreElements() {  
            return (currentItem >= 0);  
        }  
        public Object nextElement() {  
            if (!hasMoreElements())  
                throw new NoSuchElementException();  
            else  
                return items.elementAt(currentItem--);  
        }  
    }  
}
```



Klasy anonimowe

Klasa anonimowa jest to klasa wewnętrzna zadeklarowana bez nazwy i konstruktora (tylko przy użyciu new). Ze względu na nieczytelność kodu jest stosowana rzadko.

```
public class Stack {  
    private Vector items;    ...//elementy klasy Stack...  
  
    public Enumeration enumerator() {  
        return new Enumeration() { //operator new przy interfejsie!  
            int currentItem = items.size() - 1;  
            public boolean hasMoreElements() {  
                return (currentItem >= 0);  
            }  
            public Object nextElement() {  
                if (!hasMoreElements())  
                    throw new NoSuchElementException();  
                else  
                    return items.elementAt(currentItem--);  
            }  
        }  
    }  
}
```



Klasy lokalne

Klasy lokalne to klasy zdefiniowane w bloku programu Javy. Klasa taka może odwoływać się do wszystkich zmiennych widocznych w miejscu wystąpienia jej definicji. Klasa lokalna jest widoczna, i może zostać użyta, tylko w bloku w którym została zdefiniowana.

```
class Test
{
    void test()
    {
        // definicja klasy lokalnej
        class KlasaLokalna
        { }
        // deklaracja obiektu typu: KlasaLokalna
        KlasaLokalna l = new KlasaLokalna();
    }
}
```

Autoreferencja

Słowo kluczowe `this` oznacza autoreferencję.

```
class Miejsce
{
    public int X;
    String nazwa;
    public Miejsce(int X, String xnazwa)
    {
        // gdyby argumenty i pola danych miały różne nazwy
        // użycie this nie było by wymagane,
        this.X=X;
        nazwa=xnazwa; //tu nie jest wymagane
    }
    public Miejsce(int X)
    {
        this(X,"Miejsce Bez Nazwy");
    }
    public Miejsce()
    {
        this(10,"Miejsce Bez Nazwy");
    }
    public Miejsce PodajMiejsce()
    {
        return this;
    }
    }//koniec klasy
```

Usuwanie obiektów

Java nie wymaga definiowania destruktorów, gdyż istnieje mechanizm automatycznego zarządzania pamięcią (ang. garbage collection). Obiekt istnieje w pamięci dopóki istnieje do niego jakakolwiek referencja w programie, w tym sensie, że gdy referencja do obiektu nie jest już przechowywana przez żadną zmienną obiekt jest automatycznie usuwany, a zajmowana przez niego pamięć zwalniana.

Proces zbierania nieużytków jest włączany okresowo, uwalniając pamięć zajmowaną przez obiekty, które nie są już potrzebne. W czasie działania programu przeglądany jest obszar pamięci przydzielanej dynamicznie, zaznaczane są obiekty, do których istnieją referencje. Po prześledzeniu wszystkich możliwych ścieżek referencji do obiektów, te obiekty, które nie są zaznaczone (tzn. do których nie ma referencji) zostają usunięte.

Usuwanie obiektów

Program w Javie może jawnie uruchomić mechanizm zbierania nieużytków poprzez wywołanie metody `System.gc()`. Mechanizm oczyszczania pamięci z nieużytków działa w wątku o niskim priorytecie synchronicznie lub asynchronicznie, zależnie od sytuacji i środowiska systemu operacyjnego na którym wykonywany jest program w Javie. W systemach, które pozwalają środowisku przetwarzania Javy sprawdzać, kiedy wątek się rozpoczął i przerwał wykonanie innego wątku (takich jak np. Windows 2000), mechanizm czyszczenia pamięci działa asynchronicznie w czasie bezczynności systemu.

Mimo że Java nie wymaga destruktorów, istnieje możliwość deklaracji specjalnej metody `finalize` (zdefiniowanej w klasie `java.lang.Object`), która będzie wykonywana przed usunięciem obiektu z pamięci. Deklaracja takiej metody ma zastosowanie, gdy nasz obiekt np.: ma referencje do urządzeń wejścia-wyjścia i przed usunięciem obiektu należy je zamknąć.

Usuwanie obiektów - finalizacja

```
class OtwórzPlik {  
    FileInputStream m_plik = null;  
    OtwórzPlik(String nazwaPliku)  
    {  
        //Otwarcie pliku  
        try  
        { m_plik = new FileInputStream(nazwaPliku); }  
        //Obsługa wyjątku  
        catch (java.io.FileNotFoundException e)  
        { System.err.println("Nie moge otworzyc pliku" + nazwaPliku);}  
    }  
    protected void finalize () throws Throwable  
    {  
        if (m_plik != null)  
        {  
            m_plik.close();  
            m_plik = null;  
        }  
    }  
}
```