Klasy

Klasy

```
[ModyfikatorKlasy] class NazwaKlasy [extends Nadklasa]
      [implements ListaInterfejsów] {
  // Lista metod i pól
}
```

- ModyfikatorKlasy może być kombinacją wyrażeń:
  - abstract - klasa zawiera metody abstrakcyjne
  - final - nie może posiadać podklas
  - public - może być używana w kodzie poza klasą, jedna klasa publiczna w pliku (nazwa pliku = nazwa klasy)
  - private - dostępna tylko w obrębie pliku
  - <puste> - dostęp w ramach pakietu, w którym występuje

Metody

```
[ModyfikatorMetody] TypZwrotny Nazwa (Typ arg1, ...) {
  // implementacja metody
}
```

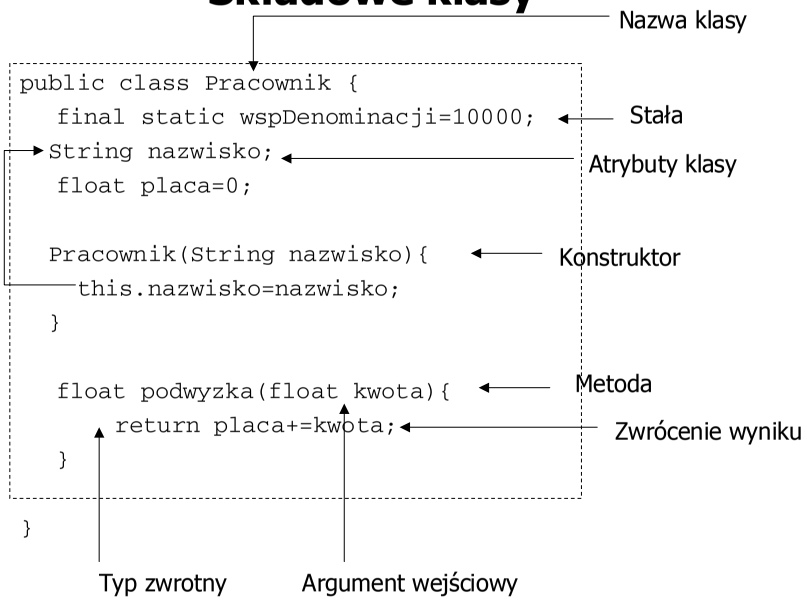
- ModyfikatorMetody może być kombinacją wyrażeń:
  - modyfikator widzialności
    - public - dostępna dla metod spoza klasy
    - protected - dostępna w klasach z pakietu i wszystkich podklasach
    - <puste> - dostępna w klasach z pakietu, w którym występuje
    - private - dostępna tylko dla metod z tej samej klasy
  - final - metoda nie może zostać przesłonięta w podklasie
  - static - wspólna dla wszystkich wystąpień obiektu
  - synchronized - blokuje dostęp do obiektu na czas wykonywania
  - native - zaimplementowana w innym języku
  - abstract - metoda bez implementacji

Pola

```
[ModyfikatorPola] Typ Nazwa [ = wartość];
```

- ModyfikatorPola może być kombinacją wyrażeń:
  - modyfikator widzialności
    - public - dostępna dla metod spoza klasy
    - protected - dostępna w klasach z pakietu i wszystkich podklasach
    - <puste> - dostępna w klasach z pakietu, w którym występuje
    - private - dostępna tylko dla metod z tej samej klasy
  - static - wspólne dla wszystkich wystąpień obiektu
  - final - stała, musi być zainicjalizowana

Składowe klasy



Konstruktor

- Konstruktor - metoda służącą do inicjowania obiektów klasy
- Jego nazwa musi pokrywać się z nazwą klasy
- Konstruktor nie może posiadać typu zwrotnego
- Konstruktory (i inne metody) mogą być przeciążane
- Przy braku definicji konstruktora, kompilator dostarcza domyślny, pusty konstruktor bezargumentowy

Niszczenie obiektu

- Nie ma w Javie destruktora
- Obiekt jest niszczony przez mechanizm *garbage collection*
- Przed zniszczeniem obiektu wywoływana jest metoda `finalize()`

Tworzenie obiektów, aktywowanie metod

- Obiekty tworzone są dynamicznie, za pomocą operatora `new`
- Obiekty są dostępne przez referencje
- Obiekty są niszczone przez *garbage collection*, gdy nie ma do nich referencji
- Odwołanie do składowych obiektu odbywa się przez operator kropki (w przypadku składowych statycznych zamiast nazwy obiektu można podać nazwę klasy)

```
{
      Pracownik jezierski = new Pracownik("Jezierski");
      Pracownik koszlajda= new Pracownik("Koszlajda");
      jezierski.podwyzka(1000);
      jezierski=null; //obiekt zostanie zniszczony

} //obiekt koszlajda zostanie zniszczony
```


Przekazywanie parametrów do metod

- Zmienne typów prostych przekazywane są przez wartość
- Obiekty są przekazywane przez referencję
  - można je zmieniać poprzez przekazaną referencję
  - referencja jest przekazywana przez wartość

```
class X{
      static void brakEfektow(Pracownik p){
            p = new Pracownik("Koszlajda");
      }
}
...
Pracownik jezierski = new Pracownik("Jezierski");
X.brakEfektow(jezierski);
System.out.println(jezierski.nazwisko); // ->"Jezierski"
...
```

Hermetyzacja

```
public class Pracownik {
  private String nazwisko;
  private float placa=0;

  public Pracownik(String nazwisko){
    this.nazwisko=nazwisko;
  }

  public float podwyzka(float kwota){
           return placa+=kwota;
     }
}
...
Pracownik p = new Pracownik("Kowalski");
p.nazwisko="Nowak";
```

nazwisko has private access in Pracownik

Składowe statyczne

- Współdzielone przez wszystkie obiekty danej klasy
- Statyczne metody aktywowane na rzecz klasy

```
public class Pracownik {
     private static int liczbaPracownikow;
      public Pracownik(String nazwisko){
          this.nazwisko=nazwisko;
          liczbaPracownikow++;
      }
    public static int iluPracownikow(){
          return liczbaPracownikow;
    }
    static{       //blok wykonywany jednorazowo przy ładowaniu klasy
      liczbaPracownikow=0;
    }
}
Pracownik p1=new Pracownik("Kowalski");
System.out.println(Pracownik.iluPracownikow()); //p1.iluPracownikow()
```


Stałe

```
public class Pracownik {
     private final Osoba matka;
     private final Osoba ojciec;

     public static final int PI=3.14;

     public Pracownik(Osoba matka, Osoba ojciec){
          this.matka=matka;
          this.ojciec=ojciec;
      }
}
```


Przeciążanie metod

```
public class Pracownik {
...
      public Pracownik(String nazwisko){
          this.nazwisko=nazwisko;
      }
      public Pracownik(String nazwisko, float placa){
            this(nazwisko);
            this.placa=placa;
      }
      public Pracownik(String nazwisko, int wiek){
            this(nazwisko);
            this.wiek=wiek;
      }
      public float podwyzka(float kwota){return placa+=kwota;}
      public void podwyzka(float kwota){placa+=kwota;}
      }
}
...
Pracownik p1=new Pracownik("Kowalski");
Pracownik p2=new Pracownik("Kowalski", 25L);
Pracownik p3=new Pracownik("Kowalski", 1000F);
```

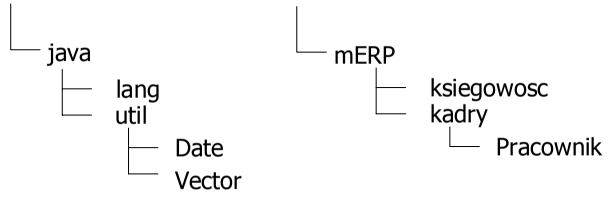
podwyzka(float) is already defined in Pracownik

Grupowanie klas w pakiety

- Informacja o przynależności do pakietu w pierwszej linii jednostki kompilacji (przy braku - pakiet domyślny)

```
package myERP.kadry;
class Pracownik { ... }
```

- Lokalizacja pliku `.class` w drzewie katalogów musi odpowiadać zadeklarowanej nazwie pakietu



- W praktyce, pakiety klas są kompresowane do plików ZIP lub JAR

Importowanie klas

- Odwołanie do klasy w pakiecie wymaga podania ścieżki:

```
myERP.kadry.Pracownik jezierski;
```

- Dzięki zastosowaniu polecenia `import`, możliwe jest odwoływanie się przez samą nazwę klasy (nie wymagane dla java.lang):

```
import myERP.kadry.Pracownik;
```

```
import myERP.kadry.*;
```

- W celu załadowania wskazywanej klasy, JVM przeszukuje wszystkie lokalizacje zapisane w zmiennej środowiskowej CLASSPATH; jeżeli CLASSPATH odwołuje się do pliku ZIP/JAR, wtedy plik ten jest automatycznie rozpakowywany w pamięci operacyjnej

```
CLASSPATH=.:/java/lib/classes12.zip:/home/jj/myJAR.jar
```


Dziedziczenie

```
public class Pracownik {
  protected  String nazwisko;
  protected float placa=0;
...
}
```

Nazwa nowej klasy (potomnej) Nazwa nadklasy (macierzystej, bazowej)

```
public class Dydaktyk extends Pracownik {
   protected int pensum=210;
   public   Dydaktyk (String nazwisko){
        super(nazwisko);
   }

   public float stawkaGodzinowa(){
        return placa/pensum;
   }
}
```


Hermetyzacja - suplement

	klasa	pakiet	klasa potomna w pakiecie	klasa potomna poza pakietem	świat
private	OK	-	-	-	-
protected	OK	-	OK	OK	-
domyślny	OK	OK	OK	-	-
public	OK	OK	OK	OK	OK

Nadpisanie i odsłanianie

```
public class Dydaktyk extends Pracownik {
   protected int pensum=210;
   public   Dydaktyk (String nazwisko){
        super(nazwisko);
   }

public float podwyzka(float stawkaGodzinowa) {
    return super.podwyzka (pensum*stawkaGodzinowa);
  }
}
```

Nadpisanie (przysłonięcie) definicji metody

Odsłonięcie przysłoniętej metody

Referencje *this* i *super*

- W metodach klasy można używać referencji **`this`** i **`super`**
- **`this`** jest referencją do bieżącego obiektu
- **`super`** udostępnia przesłonięte składowe nadklasy
- Z konstruktora klasy można wywołać inny konstruktor tej klasy (`this(...)`) lub konstruktor nadklasy (`super(...)`), w pierwszej instrukcji ciała konstruktora

Składowe abstrakcyjne

- Deklaracja własności, szkieletów
- Klasy z składowymi abstrakcyjnymi nie mogą posiadać wystąpień
- Wymagają implementacji w klasach potomnych

```
public abstract class Instrument {
   public String nazwa(){ return "instrument";}
   public abstract void strojenie();
   public abstract void graj(Nutka nutka);
}

public class Fortepian extends Instrument{
   public String nazwa(){ return "fortepian";}
   public void strojenie(){ ... };
   public void graj(Nutka nutka){ // la la
}
```


Interfejsy

- Interfejs jest w pełni abstrakcyjną klasą
- Wszystkie jego metody muszą być abstrakcyjne (domyślnie `public abstract`)
- Nie może posiadać zmiennych wystąpień (wszystkie pola są domyślnie `public static final`)
- Interfejsy kompensują brak dziedziczenia wielobazowego

```
[public] interface NazwaInterfejsu
      [extends ListaInterfejsów] {
  // Lista metod i pól statycznych
}
```

Tablice

- Tablice są obiektami
  - tworzone dynamicznie (`new`)
  - zmienne tablicowe są referencjami
- Rozmiar tablic specyfikowany przy tworzeniu
- Indeksowanie tablicy od zera
- Elementy są automatycznie inicjalizowane (dla liczb: zerami, dla obiektów: referencjami pustymi `null`)
- Odwołanie się poprzez niewłaściwy indeks generuje wyjątek
- Rozmiar tablicy dostępny jako pole o nazwie `length`
- Tablice wielowymiarowe jako tablice tablic

Korzystanie z tablic

```
int [] tab = new int[10];
for (int i=0; i < tab.length; i++)
  tab[i] = i;

int [][] chessBoard;
chessBoard = new int[8][8];

int [] primes = {1,2,3,5,5+2};
String [] verbs = {"go", "sleep"};
```

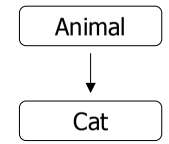

Klasy opakowujące (ang. *wrapper*)

- Dla każdego typu prostego w Javie istnieje odpowiadająca mu klasa "wrapper" (`Integer`, `Float`, ...)
- Klasy "wrapper" zastępują zmienne proste w miejscach gdzie spodziewane są obiekty
- Zawierają metody zwracające proste wartości (`intValue()`, `floatValue()`, ...)
- Zawierają przydatne metody konwersji typów np.
  - `Integer.parseInt(String)` - String -> int
  - `Integer.toString(int)` - int -> String

Hierarchia dziedziczenia

- Każda klasa może posiadać jedną nadklasę
- Korzeniem hierarchii dziedziczenia jest klasa `Object`
- Gdy dla danej klasy nie zostanie podana jej nadklasa, domyślnie przyjmowana jest klasa `Object`
- Konwersja typów (jawna lub niejawna) możliwa jest w ramach gałęzi dziedziczenia

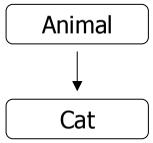
```
Cat filemon = new Cat();
Animal a = filemon; // OK
filemon = a; // error
filemon = (Cat) a; // OK
```



Zastępowanie referencji do obiektów

- Obiekt podklasy może być użyty tam, gdzie spodziewany jest obiekt nadklasy
- Prawdziwy typ obiektu może być określony przy użyciu operatora `instanceof`
- Powyższe stwierdzenia odnoszą się również do interfejsów implementowanych przez klasę obiektu

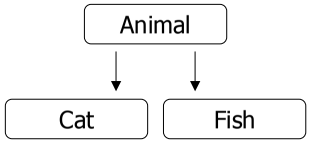
```
public void rename (Animal a) {
  if (a instanceof Cat)
  {
    ((Cat) a).catMethod();
  }
...
}
```



Polimorfizm

- Polega na dynamicznym (późnym) wiązaniu metod
- W przypadku wywołania metody obiektu podklasy poprzez referencję typu nadklasy, wywoływana jest metoda z jego klasy (na podstawie jego faktycznego typu)

```
Animal [] zoo = {
                new Cat(...),
                new Fish(...),
                ...
                };
for (int i=0; i < zoo.length; i++)
  zoo[i].feed();
```



Klasy zagnieżdżone

- Klasy można definiować jako składowe innych klas
  - statyczne klasy zagnieżdżone
    - mają dostęp jedynie do składowych statycznych klasy otaczającej
    - służą głównie do wprowadzenia dodatkowego poziomu grupowania klas w biblioteki (oprócz pakietów)
  - klasy wewnętrzne
    - ich obiekty występują wewnątrz obiektów klasy otaczającej i mają swobodę dostępu do ich składowych

```
class EnclosingClass{ . . .
  static class AStaticNestedClass { ...
  }
  class InnerClass { ...
  }
}
```

Statyczna klasa zagnieżdżona

Klasa wewnętrzna

- Klasy zagnieżdżone można definiować również w metodach lub blokach kodu

Anonimowe klasy wewnętrzne

- Przydatne do definiowania klas *jednorazowego użytku* np. do implementacji obiektów *nasłuchujących* zdarzeń generowanych przez GUI

```
 class ButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      ...
    }
  }
private ButtonListener bl = new ButtonListener();
//lub
ActionListener bl = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      ...
}
```

Wyjątki

- Stanowią wydajny i przejrzysty mechanizm obsługi błędów
- Przykłady (podklasy klasy `Throwable`)
  - błędy (podklasy `Error`) np.:
    - `OutOfMemoryError`, `InternalError`
  - wyjątki kontrolowane (podklasy `Exception`), np.:
    - `MalformedURLException`, `IOException`
  - wyjątki czasu wykonania (podklasy `RuntimeException`), np.:
    - `ArrayIndexOutOfBoundsException`, `ArithmeticException`
- Wszystkie wyjątki kontrolowane (poza wyjątkami czasu wykonania) muszą być obsłużone lub wymienione w klauzuli `throws` w deklaracji metody

```
public void wstawDoTabeli() throws SQLException {...}
```


Przechwytywanie i obsługa wyjątków

```
try {
  // instrukcje, w których może pojawić się wyjątek
}
catch (ExceptionClass1 e)
{ ... }
catch (ExceptionClass2 e)
{ ... }
...
finally
{
  // wykonuje się zawsze (!), sekcja opcjonalna
}
```


Obsługa zdarzeń wyjątkowych - przykład

```
public class SayHello {
   public static void main(String[] args) {
      int  x=10, y=0;
      int  wynik;

      System.out.println("x/y=");

      try {
         wynik = x/y;
         System.out.println(wynik);}
      catch (ArithmeticException e)
        {System.out.println("Blad arytmetyczny!");
         System.out.println(e);}
   }
}
```

Rodzaj błędu

Komunikat błędu

Wywoływanie wyjątków

- Wyjątek można wywołać jawnie

```
throw new IOException();
```

- Metody, które mogą zgłaszać wyjątki, muszą zawierać informację o tym w deklaracji

```
public void method1() throws IOException {
  // ...
}
```

Przykład definicji wyjątku użytkownika

```
public class TestException extends Exception {
   public TestException() {} ;

   public TestException (Object o, String method, Exception e)
   { this(e.getClass() + " in <" + o.getClass() + "> ( " +
       method + ") :: " + e.getMessage());
   e.printStackTrace();
   }

   public TestException(String msg) { super(msg); }
}
```