Beata Jankowska,
Instytut Automatyki, Robotyki i Inżynierii Informatycznej

# XI. Java po C++.
# Implementacja klas (i obiektów) w Javie

## 1. Klasa a obiekt

**1.1.** Klasa w języku Java ma składnię i semantykę zbliżoną do jej odpowiednika w języku C++. Jednak, odmiennie niż w C++, w Javie nie wolno rozdzielać deklaracji klasy i implementacji jej metod (muszą znajdować się w tym samym miejscu). Prowadzi to w pewnych sytuacjach do powstawania dużych plików źródłowych. Klasa obejmuje dane, czyli zmienne oraz kod, sformułowany w postaci metod. Powyższe zmienne i metody nazywają się składowymi klasy.

**1.2.** Definicja klasy jest – jak we wszystkich językach obiektowych – definicją typu danych stanowiącego podstawę do tworzenia obiektów będących egzemplarzami tej klasy. Proces tworzenia obiektów jest dwuetapowy. Najpierw deklaruje się zmienną danego typu, po czym konstruuje rzeczywistą, fizyczną wersję obiektu i przypisuje ją do zmiennej.

**1.3.** Tworzenie obiektów klasy odbywa się zawsze przy użyciu operatora `new`. Operator ten, w trakcie działania programu, dokonuje dynamicznej alokacji pamięci niezbędnej do reprezentacji obiektu i zwraca referencję do tego obiektu. Referencja w Javie stanowi odpowiednik wskaźnika w C++. Istotna różnica między oboma polega na tym, że referencji nie można modyfikować tak jak wskaźników języka C++.
Bezpośrednio po zaalokowaniu pamięci odbywa się inicjalizacja obiektu. Proces inicjalizacji zachodzi przy użyciu (sparametryzowanego, bądź nie) konstruktora.

**1.4.** Zarządzanie pamięcią dynamiczną przebiega w Javie odmiennie niż w C++. Choć bowiem obiekty alokowane są w obu wypadkach jawnie, przy użyciu operatora `new`, Java stosuje inne podejście do zwalniania pamięci zajmowanej przez obiekty nieużyteczne. W maszynie Javy działa mechanizm automatycznego zwalniania

pamięci, który włącza się sporadycznie, przy wyczerpywaniu się tego zasobu. W programach Javy nie zachodzi więc potrzeba jawnego niszczenia obiektów.

**1.5.** W związku ze szczególnym podejściem do zwalniania pamięci, Java nie przewiduje stosowania destruktorów. W ich miejsce oferuje mechanizm zwany finalizacją lub dokończeniem. Umożliwia on zdefiniowanie działań, które będą wykonane tuż przed usunięciem obiektu przez mechanizm odzyskiwania pamięci. Działania te specyfikuje się wewnątrz metody `finalize`. Należy bezwzględnie pamiętać, że na tej metodzie nie można polegać podczas normalnej pracy programu.

**1.6.** Oto przykład klasy `Stack`, implementującej stos liczb całkowitych:

```
class Stack {
  int stack[] = new int[10];
  int ref;
  Stack() {
    ref = -1;
  }

  void push(int item) {
     if(ref == 9)
       System.out.println(”Stos jest pelny.”);
     else
       stack[++ref] = item;

  }
  int pop() {
    if(ref <0) {
      System.out.println
           (”Stos nie zawiera zadnych elemetow.”);
      return 0;
    }
    else
      return stack[ref--];
  }
}
```

oraz klasy `TestStack`, wykorzystującej `Stack`:

```
class TestStack {
  public static void main(String args[]) {
    Stack mystack = new Stack();
    for(int i=0; i<10; i++) mystack.push(i);
    System.out.println(”Stos mystack”);
    for(int i=0; i<10; i++)
      System.out.println(mystack.pop());
   }
 }
```

# 2. Modyfikatory dostępu i obsługa metod w klasach Javy

**2.1.** Java oferuje zestaw trzech tradycyjnych modyfikatorów dostępu do składowych klasy: `public`, `protected` i `private`. Zezwala ponadto na stosowanie domyślnego poziomu dostępu. Jeśli nie zastosuje się żadnego modyfikatora, to składowa klasy jest dostępna publicznie dla wszystkich innych klas swojego pakietu, ale nie jest dostępna dla kodu z innych pakietów (pakiet to swoista grupa klas).

**2.2.** Język Java dopuszcza przeciążanie metod składowych klasy (także konstruktora). Przeciążanie to odbywa się w sposób analogiczny, jak w programach języka C++.

**2.3.** Parametry metod składowych mogą być zarówno typów prostych, jak i złożonych. W szczególności, dopuszcza się parametry typów klasowych.

**2.4.** Dwa używane w Javie sposoby przekazywania parametrów, przez wartość i przez referencję, są równoważne ich odpowiednikom w języku C++. Wartości typów prostych są przekazywane do metod przez wartość, zaś wszystkie obiekty – przez referencję.

**2.5.** W J2SE 5.0 poszerzono funkcjonalność języka o opcję tworzenia metod, które mogą przyjmować zmienną liczbę argumentów. Metody te nazywamy zmienno-argumentowymi lub metodami varargs, np.

```
class Varargs {
  static void vaTest(int ... v) {
    System.out.print(”L.argumentow: ” +
                     v.length + ”Zawartosc: ”);
    for (int x : v)
      System.out.print(x + ” ”);
    System.out.println();
  }
public static void main(String args[])
  {
    vaTest(10);
    vaTest(1, 2, 3);
    vaTest();
  }
}
```

**2.6.** Metody varargs można przeciążać w analogiczny sposób, jak metody tradycyjne. Trzeba jednak pamiętać o unikaniu w samych definicjach, a także przy wywołaniach niejednoznaczności.

**2.7.** Java dopuszcza rekurencję.

# 3. Statyczne składowe klasy

**3.1.** W tradycyjnym podejściu obiektowym, składowa klasy jest dostępna tylko w odniesieniu do pewnej instancji klasy (obiektu). Java, podobnie jak C++, zezwala na utworzenie składowej statycznej, udostępnianej bez podania referencji do konkretnego obiektu. Deklarację takiej składowej należy koniecznie poprzedzić słowem kluczowym `static`. Taką składową można odczytywać lub modyfikować zanim powstanie jakakolwiek instancja klasy – odnosi się ona do samej klasy, nie do jej instancji.

**3.2.** Zmienne składowe zadeklarowane jako statyczne są w istocie zmiennymi globalnymi. Przy tworzeniu obiektu klasy, jej zmienne statyczne nie podlegają kopiowaniu. Wszystkie obiekty klasy współdzielą zmienne statyczne.

**3.3.** Metody statyczne mogą być używane tylko przy spełnieniu następujących ograniczeń:
– wywołują tylko metody statyczne,
– mają dostęp tylko do danych zadeklarowanych jako `static`,
– nie korzystają ze słów kluczowych `this` i `super`.

**3.4.** Każdą inicjalizację zmiennej statycznej która musi być poprzedzona obliczeniami, należy zrealizować w specjalnie zadeklarowanym bloku `static`, np.

```
class UseStatic {
  static int a=3;
  static int b;
  static void method(int x) {
    System.out.println(”x = ” + x);
    System.out.println(”a = ” + a);
    System.out.println(”b = ” + b);
  }
  static {
    System.out.println
         (”Inicjalizacja w bloku statycznym.”);
    b=a*4;
  }
  public static void main(String args[]) {
    method(42);
  }
}
```

**3.5.** Przy użyciu składowych statycznych, Java implementuje łatwą do sterowania wersję globalnych metod i zmiennych.

# 4. Klasy zagnieżdżone

**4.1.** Jedną klasę można zdefiniować wewnątrz innej klasy. Zagnieżdżona klasa ma dostęp do wszystkich składowych, także prywatnych, należących do klasy zewnętrznej. Klasa zewnętrzna nie ma specjalnego dostępu do składowych klasy zagnieżdżonej. Klasa zagnieżdżona znana jest tylko w zawierającym ją zakresie. Klasy

zagnieżdżone definiuje się zwykle jako klasy niestatyczne (bez składowych statycznych).

**4.2.** Klasy zagnieżdżone można definiować także wewnątrz bloku metody lub nawet wewnątrz bloku pętli, np.

```
class Outer {
  int outer_x = 100;
  void test() {
    for (int i=0; i<5; i++) {
      class Inner {
        void display() {
          System.out.println
             (”display: outer_x = ” + outer_x);
        }
      }
      Inner inner = new Inner();
      inner.display();
    }
  }
}

class InnerClassDemo {
  public static void main(String args[]) {
    Outer outer = new Outer();
    outer.test();
  }
}
```

# 5. Klasa String

**5.1.** Klasa `String` służy do obsługi obiektów łańcuchowych. Każdy tekst w kodzie programu (w tym – stała tekstowa) jest zamieniany na obiekt typu `String`. Obiekty typu `String` charakteryzują się niezmiennością: po utworzeniu takiego obiektu, jego zawartość nie może być zmodyfikowana. Takie modyfikacje można przeprowadzać jedynie przy użyciu pomocniczej klasy `StringBuffer`, tworząc w trakcie operacji nowy obiekt `String`,

np.

```
String s = ”Jan ”;
s = s + ”Kowalski”;
```

Użyty w instrukcji operator `’+’` – jedyny operator do działań na łańcuchach – jest operatorem konkatencji tekstów. W wyniku wykonania powyższej instrukcji powstanie więc tekst `”Jan Kowalski”`. Zostanie on zapisany w nowo powstałym obiekcie, którego referencja będzie przypisana do `s`. Omawiana instrukcja ma w istocie następujący przebieg:

```
s = new
    StringBuffer(s).append(”Kowalski”).toString().
```

**5.2.** Oprócz operatora konkatencji `’+’`, klasa `String` zawiera szereg innych przydatnych metod, m.in.:

```
boolean equals(String obiekt)
int length()
char charAt(int indeks)
```

## 6. Dziedziczenie klas

**6.1.** W Javie w pełni zrealizowano paradygmat dziedziczenia. Klasę pochodną (podklasę), dziedziczącą po klasie bazowej (nadrzędnej) definiuje się przy użyciu słowa kluczowego `extends`, np.

```
class A {
  int i, j;
  void showij() {
    System.out.println(”i i j:” + i + ” ” + j);
  }
}
class B extends A {
  int k;
  void showk() {
    System.out.println(”k: ” + k);
  }
}
```

**6.2.** Java, umożliwiając tworzenie liniowych łańcuchów dziedziczenia, nie udostępnia (znanej z języka C++) możliwości wielodziedziczenia. Problem widoczności składowych klasy bazowej w obrębie klasy pochodnej rozwiązano analogicznie jak w C++ przy założeniu dziedziczenia w trybie `public`.

**6.3.** Do zmiennej typu klasowego bazowego wolno przypisać referencję do obiektu typu klasowego pochodnego. Należy przy tym pamiętać, że to typ zmiennej referencyjnej, a nie typ obiektu wskazywanego przez zmienną wyznacza zakres działań, które można podejmować w odniesieniu do obiektu. Nie da się natomiast zrealizować operacji przypisania obiektu typu klasowego bazowego do zmiennej typu klasowego pochodnego.

# 7. Polimorfizm w Javie

**7.1.** W Javie zaimplementowano mechanizm przesłaniania metod. Mechanizm, stanowiący odpowiednik metod wirtualnych (i polimorfizmu), umożliwia dynamiczne przydzielanie metod w trakcie działania programu. W tym wypadku, o wersji wybranej metody decyduje typ obiektu, na który wskazuje referencja, nie zaś typ zmiennej referencyjnej.
Przesłanianie odbywa się bez użycia specjalnego słowa kluczowego (brak odpowiednika `virtual` z C++).

**7.2.** W sytuacji, gdy implementacja metody wirtualnej zrealizowana jest dopiero na poziomie klas pochodnych, w klasie bazowej tę metodę trzeba zadeklarować jako metodę abstrakcyjną, np.

```
abstract class A {
  abstract void callme();
  void callmetoo() {
    System.out.println
         (”To jest konkretna metoda.”);
  }
}
```

```
class B extends A {
  void callme() {
    System.out.println
         (”Implementacja callme() z klasy B.”);
  }
}
class AbstractDemo {
  public static void main(String args[]) {
    B b = new B();
    b.callme();
    b.callmetoo();
  }
}
```

**7.3.** Klasa zawierająca przynajmniej jedną metodę abstrakcyjną musi być oznaczona słowem `abstract`. Nie można tworzyć obiektów klasy abstrakcyjnej, lecz można posługiwać się zmiennymi referencyjnymi o typie klasy abstrakcyjnej.

## 8. Konstrukt `super`

**8.1.** Za pomocą konstruktu `super` można w konstruktorze klasy pochodnej wymusić wywołanie konstruktora klasy bazowej. Konstrukt ten przyjmuje postać składniową:

`super (`*lista_argumentów*`)`,

gdzie lista argumentów oznacza listę parametrów aktualnych odpowiadających parametrom formalnym stosownego konstruktora.
Zastosowanie tego konstruktu jest konieczne w wypadku, gdy konstruktor klasy bazowej jest sparametryzowany. Konstrukt `super` musi poprzedzać właściwe instrukcje konstruktora klasy pochodnej.

**8.2.** Słowa kluczowego `super` można też używać dopełniająco w stosunku do konstruktu `this` (o semantyce identycznej jak w języku C++). Referencja w postaci:

`super.skladowa`

odnosi się zawsze do tej składowej obiektu typu klasowego pochodnego, która pochodzi z nadklasy, czyli klasy bazowej dla tej klasy pochodnej. Stosuje się ją najczęściej w sytuacjach, w których składowe klasy pochodnej zasłaniają składowe klasy bazowej.

# 9. Słowo `final`

**9.1.** Słowo kluczowe `final` ma trzy zastosowania:
- pozwala deklarować i używać zmiennych o charakterze stałych,
- umożliwia tworzenie metod, których (zgodnie z intencją) nie da się przesłonić,
- umożliwia definiowanie klas, po których (zgodnie z intencją) nie wolno dziedziczyć.

# 10. Klasa `Object`

10.1. Klasa `Object` jest szczególną klasą Javy: dziedziczą po niej wszystkie klasy programu. `Object` jest więc klasą bazową wszystkich innych klas. Do zmiennej referencyjnej typu `Object` można przypisać obiekt dowolnej klasy, a także obiekt-tablicę.