

Podstawy języka JAVA

Wykład nr 7 – Konstruktory i ich przeciążanie w języku Java. Bloki inicjalizujące i niszczenie obiektów.

Dziedziczenie w języku Java.



Java – konstruktory

- **Konstruktor** jest specjalną metodą danej klasy, który wywoływany jest podczas tworzenia instancji klasy.
- **Konstruktor** ma (i musi mieć) taką samą nazwę jak klasa.
- **Konstruktor** posiada zasadniczą różnicę w stosunku do metody – można go wywołać tylko za pomocą operatora **new**.
- Jedna klasa może posiadać więcej niż jeden konstruktor.
- Konstruktor może przyjmować zero lub więcej parametrów.
- Konstruktor nie zwraca żadnej wartości.



Java – konstruktory

```
public class Figure {  
    int a;  
    int b;  
    int c;  
    int d;  
    public Figure()  
    {  
  
    }  
    public Figure(int a, int b)  
    {  
        this.a=a;  
        this.b=b;  
    }  
    public Figure (int a, int b, int c, int d)  
    {  
        this.a=a;  
        this.b=b;  
        this.c=c;  
        this.d=d;  
    }  
}
```

```
public class MainClass {  
    public static void main(String[] args) {  
        Figure f1 = new Figure();  
        Figure f2 = new Figure(10,10);  
        Figure f3 = new Figure(5,10,15,20);  
    }  
}
```



Java – przeciążanie konstruktorów

- **Przeciążanie** to sytuacja, w której kilka metod ma taką samą nazwę, ale różne parametry.
- Kompilator musi zdecydować, którą wersję wywoła.
- Decyzję tą podejmuje na podstawie dopasowania typów parametrów w nagłówkach różnych metod do typów wartości przekazanych w konkretnym wywołaniu.
- W przypadku kiedy nie istnieje takie dopasowanie lub jest ich kilka to występuje błąd kompilacji.
- Proces taki nazywa się **rozstrzygnięciem przeciążania** (ang. overloading resolution)



Java – wartości domyślne pól

- Jeżeli wartość pola nie zostanie jawnie ustawiona w konstruktorze, to pole to automatycznie przyjmuje wartość domyślną.
- Dla typów liczbowych takie pola ustawiane są na wartość 0, wartość logiczna na false, a referencje do obiektów na wartość null.
- Inicjowanie pól wartościami domyślnymi nie jest jednak dobrym stylem programowania.
- Dużo prościej jest zrozumieć kod, w którym wartości pól są jawnie inicjowane.



Java – konstruktor bezargumentowy

- Wiele klas zawiera **konstruktor bezargumentowy**, który pozwala na tworzenie obiektów z wartościami domyślnymi.
- Dla klasy Figure taki konstruktor może wyglądać następująco:

```
public class Figure {  
    int a;  
    int b;  
    int c;  
    int d;  
    String description;  
    public Figure()  
    {  
        this.a=0;  
        this.b=0;  
        this.c=0;  
        this.d=0;  
        this.description="";  
    }  
}
```



Java – konstruktor bezargumentowy

- Jeżeli pole description nie zostałoby zainicjowane pustym ciągiem znaków, to w dalszej części programu można by spróbować wykonać odwołanie do wartości null.
- **Konstruktor domyślny (bezargumentowy)** jest stosowany, jeżeli programista nie utworzy żadnego konstruktora.
- **Konstruktor domyślny** ustawia wszystkie pola na wartości domyślne.
- Jeżeli klasa ma przynajmniej jeden konstruktor, ale nie ma konstruktora domyślnego to nie można utworzyć obiektu tej klasy bez podania odpowiednich parametrów konstrukcyjnych.



Java – konstruktor bezargumentowy

```
public class Figure {  
    int a;  
    int b;  
    int c;  
    int d;  
    String description;  
    public Figure (int a, int b, int c, int d, String description)  
    {  
        this.a=a;  
        this.b=b;  
        this.c=c;  
        this.d=d;  
        this.description=description;  
    }  
}
```

```
public class MainClass {  
    public static void main(String[] args) {  
        Figure f = new Figure(5,10,15,20,"To jest kwadrat");  
        Figure f1 = new Figure();  
    }  
}
```



Java – jawna inicjacja pól

- Dzięki możliwości **przeciążania konstruktorów** początkowy stan obiektu klasy może być ustawiany na wiele sposobów.
- Bez względu na ilość przekazywanych parametrów, dobrą praktyką jest ustawienie wartości każdego pola na „sensowną” wartość.
- Dobrym rozwiązaniem jest przypisanie każdemu polu w definicji klasy jakiejś wartości.

```
public class Figure {  
    int a=0;  
    int b=0;  
    int c=0;  
    int d=0;  
    String description="";  
}
```



Java – jawna inicjacja pól

- Przypisanie z poprzedniego slajdu następuje przed wywołaniem konstruktora.
- Składnia ta jest również pożyteczna w przypadku kiedy wszystkie konstruktory klasy muszą ustawiać określoną składową na tę samą wartość.
- Wartość inicjująca pole nie musi być jednak stała.
- Może być ustawiana za pomocą wywołania metody np. ustawienie unikalnego numeru id dla każdej figury.



Java – jawna inicjacja pól

```
public class Figure {  
    private int a=0;  
    private int b=0;  
    private int c=0;  
    private int d=0;  
    String description="";  
    private static int nextID;  
    public int ID=setId();  
    private static int setId()  
    {  
        int r=nextID;  
        nextID++;  
        return r;  
    }  
}
```

```
public class MainClass {  
    public static void main(String[] args) {  
        Figure f1 = new Figure();  
        Figure f2 = new Figure();  
        System.out.println(f1.ID);  
        System.out.println(f2.ID);  
    }  
}
```

Console ✕

<terminated>

0

1



Java – nazywanie parametrów

- Przy pisaniu konstruktorów bardzo często pojawia się problem nazywania parametrów.
- Część programistów wykorzystuje **jednolite nazwy jednoliterowe** – w każdym konstruktorze ten sam parametr ma taką samą nazwę jednoliterową.
- Inne osoby stawiają **przedrostek a** przed każdym parametrem – dzięki temu nazwa parametru od razu pozwala zrozumieć jego przeznaczenie.
- Często wykorzystywany jest również fakt, że zmienne parametryczne przesłaniają składowe obiektów o tej samej nazwie.



Java – nazywanie parametrów

- Przykładowo jeżeli zostanie wywołany parametr o nazwie `wiek`, to `wiek` będzie się odnosił do parametru, a nie składowej obiektu.
- W celu uzyskania dostępu do składowej w takim przypadku, należy wykorzystać słowo kluczowe **this** i po kropce napisać składową obiektu.
- Przypominając – **this** oznacza parametr niejawny, to znaczy obiekt, który jest właśnie konstruowany.



Java – nazywanie parametrów

```
public class Figure {  
    private int pole;  
    private int obwod;  
    String description="";  
    public Figure (int aPole, int aObwod, String aDescription)  
    {  
        pole=aPole;  
        obwod=aObwod;  
        description=aDescription;  
    }  
}
```

```
public class Figure {  
    private int pole;  
    private int obwod;  
    String description="";  
    public Figure (int pole, int obwod, String description)  
    {  
        this.pole=pole;  
        this.obwod=obwod;  
        this.description=description;  
    }  
}
```

```
public class Figure {  
    private int pole;  
    private int obwod;  
    String description="";  
    public Figure (int p, int o, String d)  
    {  
        pole=p;  
        obwod=o;  
        description=d;  
    }  
}
```



Java – wywoływanie innego konstruktora

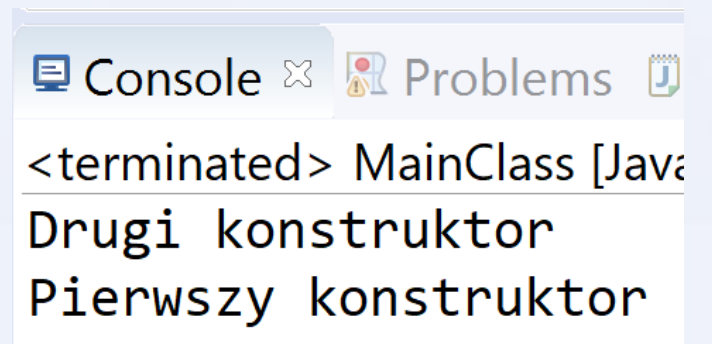
- Słowo kluczowe **this** odwołuje się do parametru niejawnego metody.
- Ma ono jednak jeszcze jedno zastosowanie.
- Jeśli pierwsza instrukcja konstruktora ma postać **this(...)**, to konstruktor ten wywołuje inny konstruktor tej samej klasy.
- Słowo kluczowe **this** w takim przypadku jest bardzo przydatne, ponieważ pozwala wspólny kod kilku konstruktorów napisać tylko jeden raz.
- Jeżeli słowo kluczowe **this(...)** nie zostanie użyte jako pierwsza instrukcja konstruktora to wystąpi błąd kompilatora.



Java – wywoływanie innego konstruktora

```
public class Figure {  
    private int pole;  
    private int obwod;  
    String description="";  
  
    public Figure (int pole, int obwod)  
    {  
        this(pole,obwod,"Opis figury");  
        System.out.println("Pierwszy konstruktor");  
    }  
  
    public Figure (int pole, int obwod, String description)  
    {  
        this.pole=pole;  
        this.obwod=obwod;  
        this.description=description;  
        System.out.println("Drugi konstruktor");  
    }  
}
```

```
public class MainClass {  
    public static void main(String[] args) {  
        Figure f1 = new Figure(10,20);  
    }  
}
```



Java – bloki inicjujące

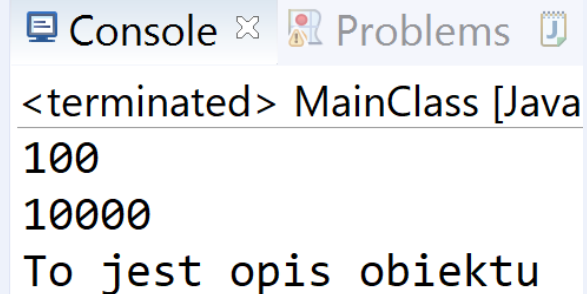
- Przedstawione zostały już dwa sposoby inicjacji pól danych:
 - poprzez ustawienie wartości w konstruktorze,
 - poprzez przypisanie wartości w deklaracji.
- Istnieje również trzeci sposób, polegający na wykorzystaniu **bloku inicjującego**.
- W deklaracji klasy mogą znajdować się dowolne bloki kodu, a zawarte w nich instrukcje są wywoływane za każdym razem, gdy konstruowany jest obiekt danej klasy.



Java – bloki inicjujące

```
public class Figure {  
    public int pole;  
    public int obwod;  
    public String description="";  
    {  
        pole=100;  
        obwod=10000;  
    }  
    {  
        description="To jest opis obiektu";  
    }  
    public Figure ()  
    {  
    }  
}
```

```
public class MainClass {  
    public static void main(String[] args) {  
        Figure f1 = new Figure();  
        System.out.println(f1.pole);  
        System.out.println(f1.obwod);  
        System.out.println(f1.description);  
    }  
}
```



The screenshot shows an IDE window with two tabs: 'Console' and 'Problems'. The 'Console' tab is active and displays the output of the program. The text '<terminated> MainClass [Java' is at the top, followed by the printed values: '100', '10000', and 'To jest opis obiektu'.

```
<terminated> MainClass [Java  
100  
10000  
To jest opis obiektu
```



Java – bloki inicjujące

- Pierwszym krokiem przy powoływaniu obiektu jest wykonanie **bloków inicjujących**.
- Następnie dopiero wykonywane są instrukcje zawarte w **konstruktorze**.
- Sposób inicjowanie pól za pomocą bloków inicjujących nigdy nie jest niezbędny i nie jest stosowany zbyt często przez programistów.
- Prościej po prostu jest umieścić kod inicjujący wewnątrz odpowiedniego konstruktora.
- Zastosowanie wszystkich przedstawionych metod inicjacji pól może bardzo negatywnie wpłynąć na czytelność kodu.



Java – bloki inicjujące

- Kiedy wywoływany jest konstruktor to mają miejsce następujące zdarzenia:
 1. Wszystkie pola inicjowane są **wartościami domyślnymi** (0, false lub null).
 2. Wszystkie **inicjatory i bloki inicjujące** są wykonywane w takiej kolejności w jakiej znajdują się w klasie.
 3. Jeżeli w pierwszym wierszu konstruktora znajduje się wywołanie innego **konstruktora**, to wykonywane są instrukcje innego konstruktora.
 4. Wykonywane jest ciało **konstruktora**.



Java – bloki inicjujące

- Pola statyczne można zainicjować, podając wartość początkową lub korzystając ze statycznego bloku inicjującego.
- Pierwszy sposób został już omówiony w trakcie wykładów:
private static int nextId=1;
- Jeżeli inicjacja pola statycznego odbywa się za pomocą bardziej złożonych instrukcji, to można wykorzystać do tego celu statyczny blok inicjujący.
- Odpowiedni kod należy umieścić w bloku opatrzonym etykietą **static**.
- Przykładowo chcemy, aby numery Id zaczynały się od wartości losowej, ale mniejszej od 10000.

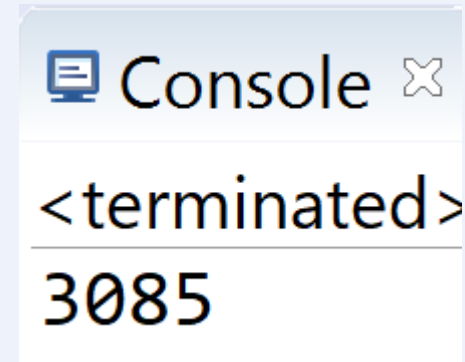


Java – bloki inicjujące

```
import java.util.Random;

public class Figure {
    private static int nextID;
    public int ID;
    static
    {
        Random r = new Random();
        nextID=r.nextInt(10000);
    }
    public Figure ()
    {
        ID=nextID;
        nextID++;
    }
}
```

```
public class MainClass {
    public static void main(String[] args) {
        Figure f1 = new Figure();
        System.out.println(f1.ID);
    }
}
```



Console

<terminated>

3085



Java – bloki inicjujące

- Inicjacja statyczna następuje w chwili pierwszego załadowania klasy.
- **Pola statyczne**, podobnie jak zmienne składowe, przybierają wartości domyślne 0, false lub null, w przypadku kiedy nie zostaną im nadane w jawny sposób inne wartości.
- Inicjatory pól statycznych i statyczne bloki inicjujące są wykonywane w takiej kolejności, w jakiej znajdują się w deklaracji klasy.



Java – niszczenie obiektów i metoda finalize

- W wielu językach programowania dostępne są tzw. **destrukторы**.
- Metody te wykonują pewne operacje porządkowe w przypadku kiedy dany obiekt wyjdzie z użytku.
- Ich podstawowym zadaniem jest przywracanie pamięci, która była przydzielona do obiektów.
- Ponieważ w języku Java zastosowano mechanizm automatycznego usuwania nieużytków, nie trzeba tego robić ręcznie.
- Wynika z tego jeszcze jedna cecha języka Java.
- **W Java nie ma destruktorów.**



Java – niszczenie obiektów i metoda finalize

- Niektóre obiekty korzystają jednak z innych zasobów niż tylko pamięć np. pliku lub uchwytów do innych obiektów, które wykorzystują zasoby systemowe.
- W takiej sytuacji trzeba zwrócić do ponownego użytku wykorzystywany **zasób**, wtedy gdy przestaje być potrzebny.
- Do każdej klasy w Java można dodać metodę finalize.
- Jest ona wywoływana przed usunięciem obiektu przez system zbierania nieużytków.
- Nie należy jednak polegać na metodzie finalize do przywracania zasobów, których jest mało.
- Nigdy nie wiadomo kiedy finalize zostanie wywołane.



Java – niszczenie obiektów i metoda finalize

- Jeżeli dany zasób musi być zamknięty natychmiast po zakończeniu jego używania to należy o to zadbać we własnym zakresie.
- Przykładem tutaj jest połączenie do bazy danych, które należy zamknąć jeżeli nie są już planowane dalsze operacje na bazie danych.
- Do zamykania zasobów służy metoda **close**, którą programista wywołuje w celu skasowania określonych zasobów i gdy skończy pracę z określonym obiektem.



Java – dziedziczenie

- **Dziedziczenie (ang. Inheritance)** – jest to technika umożliwiająca tworzenie nowych klas na bazie klas już istniejących.
- Klasa, która dziedziczy po innej klasie przejmuje jej metody i pola oraz może dodawać własne metody i pola, które służą jej przystosowaniu do nowych zadań.
- Należy wyobrazić sobie powiązanie pomiędzy klasą zwierzę, a klasą pies.
- Łatwo zauważyć, że pies **jest** specjalnym typem zwierzęcia.
- Właśnie związek „jest” pomiędzy klasami stanowi cechę charakterystyczną dziedziczenia.



Java – dziedziczenie

- Dziedziczenie w Java określone jest słowem kluczowym `extends`.
- Oznacza ono, że nowa klasa jest tworzona na podstawie istniejącej klasy.
- Klasę, która już istnieje nazywa się **klasą bazową** (ang. base class), **nadklasą** (ang. superclass) lub **klasą macierzystą** (ang. parent class).
- Nowo tworzoną klasę nazywa się **podklasą** (ang. subclass) lub **klasą potomną** (ang. child class).



Java – dziedziczenie

```
public class Zwierze {  
    String nazwa;  
    int konczyny;  
    public Zwierze(String nazwa, int konczyny)  
    {  
        this.nazwa=nazwa;  
        this.konczyny=konczyny;  
    }  
    public void Oddychaj()  
    {  
  
    }  
}
```

```
public class Ryba extends Zwierze{  
  
    public Ryba(String nazwa)  
    {  
        super(nazwa,0);  
    }  
    public void Plyn()  
    {  
  
    }  
}
```

```
public class Pies extends Zwierze {  
    public Pies(String nazwa)  
    {  
        super(nazwa,4);  
    }  
  
    public void Biegaj()  
    {  
  
    }  
    public void MachajOgonem()  
    {  
  
    }  
}
```



Java – dziedziczenie

- Czasami metoda z klasy nadrzędnej nie ma odpowiedniego zachowania dla klasy podrzędnej.
- W takim przypadku można **przesłonić (ang. override)** taką metodę w klasie podrzędnej.
- Przesłonięcie polega na napisaniu metody o tej samej nazwie w klasie podrzędnej, która ma być przesłonięta z klasy nadrzędnej.
- W przypadku konieczności wykorzystania metody z klasy nadrzędnej należy wykorzystać słowo kluczowe **super**.
- Pozwala ono na odwołania do klasy nadrzędnej.

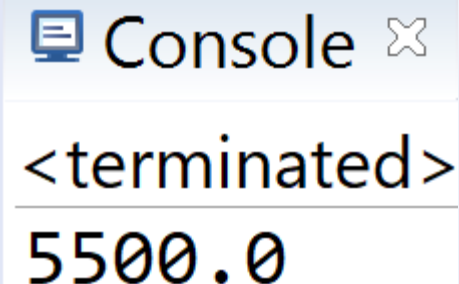


Java – dziedziczenie

```
public class Employee {  
    private double salary;  
  
    public Employee(double salary)  
    {  
        this.salary=salary;  
    }  
  
    public double getSalary()  
    {  
        return this.salary;  
    }  
}
```

```
public class Manager extends Employee {  
  
    double bonus;  
    public Manager(double salary, double bonus)  
    {  
        super(salary);  
        this.bonus=bonus;  
    }  
    public double getSalary()  
    {  
        double salary = super.getSalary();  
        return this.bonus+salary;  
    }  
}
```

```
public class MainClass {  
    public static void main(String[] args) {  
        Manager m = new Manager(5000,500);  
        System.out.println(m.getSalary());  
    }  
}
```



Console ✕

<terminated>

5500.0



Java – dziedziczenie

- Słowo kluczowe **super** w konstruktorze umożliwia wywołanie konstruktora klasy nadrzędnej.
- Należy pamiętać, aby to wywołanie za pomocą **super** było pierwszą linią w konstruktorze klasy podrzędnej.
- Jeżeli konstruktor klasy podrzędnej nie wywołuje w jawny sposób konstruktor klasy nadrzędnej to wywoływany jest domyślny **(bezparametrowy) konstruktor nadklasy**.
- Jeżeli klasa podrzędna nie wywołuje jawnie żadnego konstruktora klasy nadrzędnej, a klasa nadrzędna nie ma konstruktora domyślnego to kompilator zgłosi błąd.

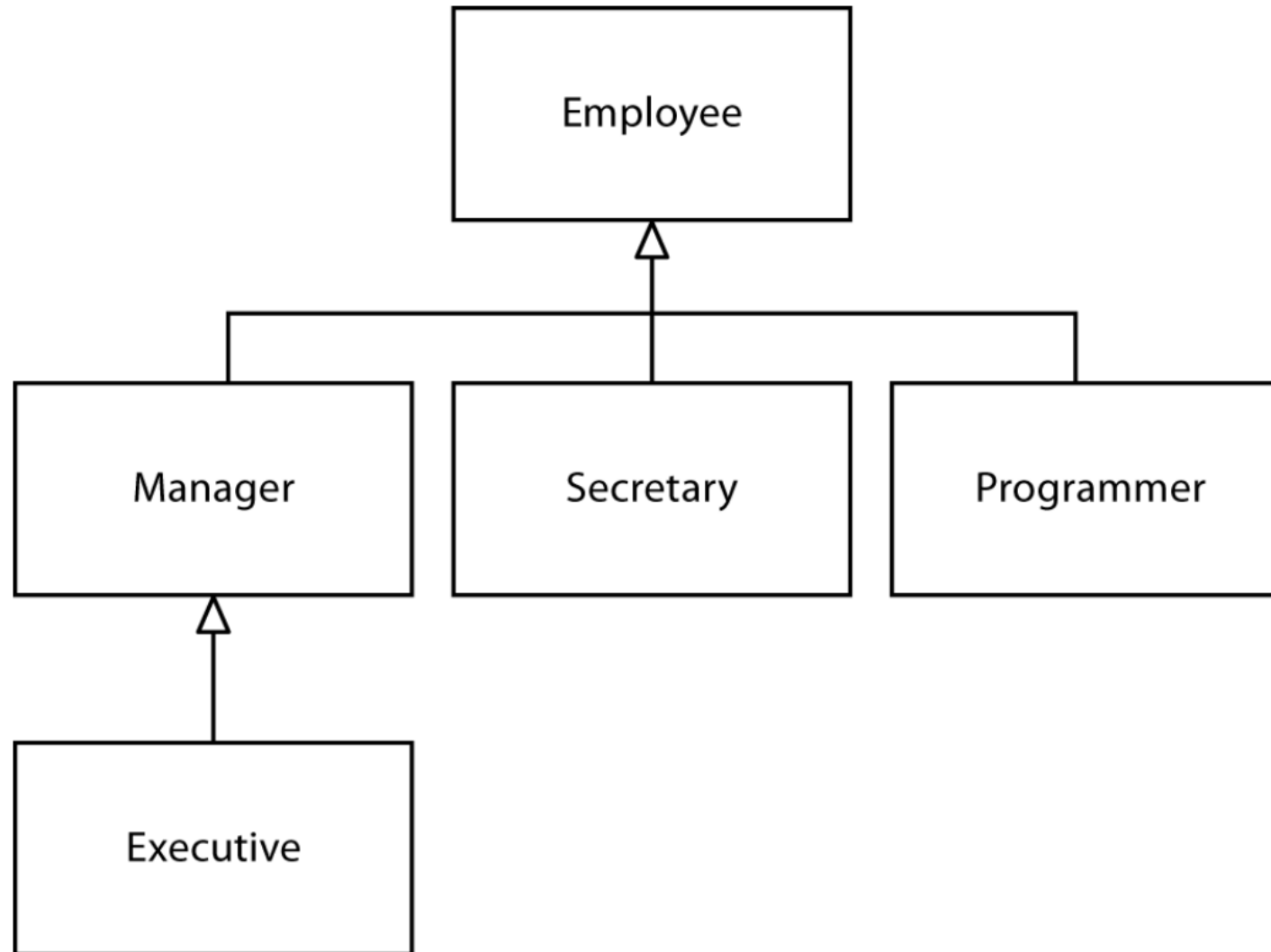


Java – hierarchia dziedziczenia

- Dziedziczenie nie dotyczy jedynie jednego poziomu klas.
- Przykładowo rozszerzeniem klasy Manager może być Executive (dyrektor).
- Strukturę klas i ich drogi dziedziczenia od wspólnej klasy bazowej nazywa się **hierarchią dziedziczenia** (ang. Inheritance hierarchy).
- Ścieżka od danej klasy do jej przodków w hierarchii dziedziczenia ma nazwę **łańcucha dziedziczenia** (ang. Inheritance chain).
- Łańcuchów dziedziczenia może być wiele np. klasę Employee mogą rozszerzać klasa Programmer oraz Secretary, które nie mają nic wspólnego z klasą Manager.



Java – hierarchia dziedziczenia

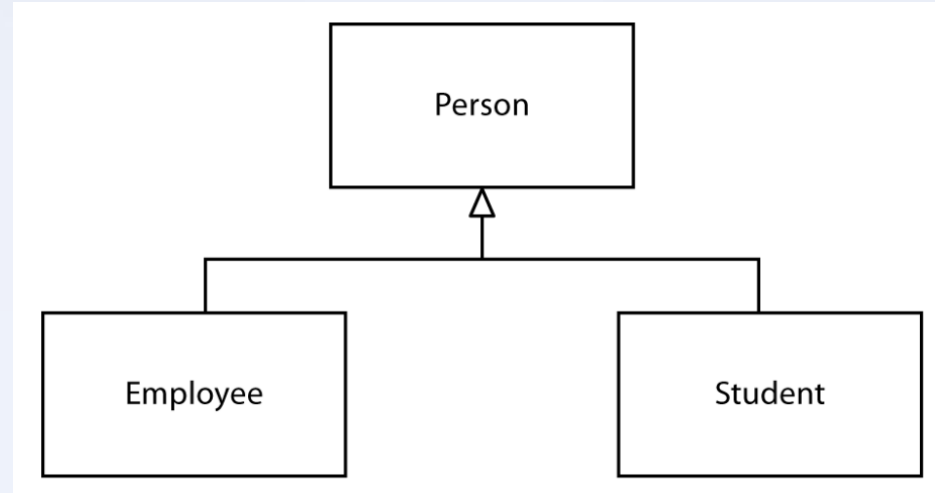


Java – klasy abstrakcyjne

- Im wyżej programista znajduje się w hierarchii dziedziczenia tym klasy stają się co raz bardziej ogólne, abstrakcyjne.
- W pewnym momencie klasa nadrzędna staje się tak abstrakcyjna, że traktowana jest jako podstawa do tworzenia innych klas, a nie do powoływania obiektów tej klasy.
- Przykładowo klasy Employee jak i Student mogą posiadać nadklasę o nazwie Person.
- Zarówno pracownik jak i student są osobami. Natomiast osoba jest bytem, który posiada imię i nazwisko. W związku z tym te pola oraz metody służące do ich pobierania można przenieść do klasy Person.



Java – klasy abstrakcyjne



- Zarówno klasa Employee jak i Student mogą posiadać metodę getDescription, która zwraca krótki opis pracownika lub studenta.
- W takim przypadku czy klasa Person również musi posiadać taką metodę? Co powinna ona zwracać? Pusty łańcuch znaków?



Java – klasy abstrakcyjne

- Dzięki wykorzystaniu słowa kluczowego **abstract** nie ma w ogóle potrzeby implementowania tej metody w klasie nadrzędnej.

public abstract String getDescription();

- Klasa, która zawiera przynajmniej jedną metodę **abstrakcyjną**, sama również musi być **abstrakcyjna**.

```
public abstract class Person {  
    String name;  
    String surname;  
    public abstract String getDescription();  
}
```



Java – klasy abstrakcyjne

- Klasa abstrakcyjna może zawierać pola i metody konkretne.

```
public abstract class Person {  
    private String name;  
    private String surname;  
    public abstract String getDescription();  
    public String getName()  
    {  
        return name+" "+surname;  
    }  
}
```

- Metody abstrakcyjne pełnią rolę symbolu zastępczego dla metod, które są implementowane w podklasach.
- Przy rozszerzaniu klas abstrakcyjnych, programista ma dwie odmienne metody dalszego postępowania.



Java – klasy abstrakcyjne

- Może pozostawić niezdefiniowane niektóre lub wszystkie metody nadklasy – wtedy podklasa musi również być **abstrakcyjna**.
- Może zdefiniować wszystkie metody i wtedy podklasa nie jest **abstrakcyjna**.
- **Podstawową cechą klas abstrakcyjnych jest to, że nie można tworzyć jej obiektów.**
- Możliwe jest jednak utworzenie **zmiennej obiektowej klasy abstrakcyjnej**, ale musi ona odwoływać się do **obiektu nieabstrakcyjnej podklasy**.

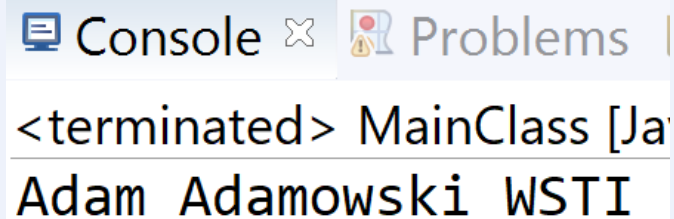


Java – klasy abstrakcyjne

```
public abstract class Person {  
    private String name;  
    private String surname;  
    public abstract String getDescription();  
    public String getNames()  
    {  
        return name+" "+surname;  
    }  
    public void setName(String name)  
    {  
        this.name=name;  
    }  
    public void setSurname(String surname)  
    {  
        this.surname=surname;  
    }  
}
```

```
public class MainClass {  
    public static void main(String[] args) {  
        Person p = new Student("Adam","Adamowski","WSTI");  
        System.out.println(p.getDescription());  
    }  
}
```

```
public class Student extends Person {  
    private String college;  
    public Student(String name, String surname, String college)  
    {  
        super.setName(name);  
        super.setSurname(surname);  
        this.college=college;  
    }  
    public String getDescription()  
    {  
        return super.getNames()+" "+this.college;  
    }  
}
```



The screenshot shows an IDE interface with two tabs: 'Console' and 'Problems'. The 'Console' tab is active and displays the output of the program. The text in the console is: '<terminated> MainClass [Ja', 'Adam Adamowski WSTI'. The text is partially cut off at the end of the line.

```
<terminated> MainClass [Ja  
Adam Adamowski WSTI
```



Java – klasa bazowa Object

- Klasa **Object** jest podstawą wszystkich pozostałych klas w języku Java.
- Każda klasa (wbudowana i pisana przez programistę) rozszerza klasę **Object**.
- Nigdy jednak nie trzeba w jawny sposób pisać `extends Object`.
- Wynika to z faktu, że jeżeli dla klasy nie jest jawnie podana żadna nadklasa, to automatycznie jest nią klasa `Object`.
- Za pomocą zmiennej typu `Object` można odwoływać się do wszystkich typów obiektów.

```
Object o = new Manager(5000,500);  
Object o1 = new Employee(3000);
```



Java – klasa bazowa **Object**

- W Java tylko typy podstawowe (liczby, znaki, wartości logiczne) nie są obiektami.
- Wszystkie typy tablicowe – niezależnie od tego czy przechowują typy podstawowe czy obiekty – są typami klasowymi rozszerzającymi klasę **Object**.
- W klasie **Object** dostępna jest metoda **equals**, która dokonuje porównania dwóch obiektów.
- Jej implementacja sprawdza czy dwie referencje do obiektów są identyczne.



Java – klasa bazowa Object

```
public class MainClass {  
    public static void main(String[] args) {  
        Object o = new Manager(5000,500);  
        Object o1 = new Employee(3000);  
        Object o2 = new Manager(5000,500);  
        boolean p = o.equals(o1);  
        boolean p1 = o.equals(o2);  
        boolean p2 = o1.equals(o2);  
        boolean p3 = o.equals(o);  
        System.out.println(p+" "+p1+" "+p2+" "+p3);  
    }  
}
```

Console Problems Debug Shell
<terminated> MainClass [Java Application]
false false false true



Java – klasa bazowa Object

- Programista może w swojej klasie nadpisać metodę **equals**, która może np. porównywać wartości składowych obiektu, a nie referencje.
- W jaki sposób powinna zachować się nadpisana metoda equals w przypadku kiedy parametr jawny i parametr niejawny nie należą do tej samej klasy?
- Bardzo często programiści wykorzystują słowo kluczowe **instanceof**, które sprawdza czy dany obiekt należy do konkretnej klasy lub jej podklasy.



Java – klasa bazowa Object

```
public class MainClass {  
    public static void main(String[] args) {  
        Object o = new Manager(5000,500);  
        Object o1 = new Employee(3000);  
        boolean p = (o instanceof Manager);  
        boolean p1 = (o instanceof Employee);  
        boolean p2 = (o1 instanceof Manager);  
        boolean p3 = (o1 instanceof Employee);  
        System.out.println(p+" "+p1+" "+p2+" "+p3);  
    }  
}
```

Console Problems Debug Shell
<terminated> MainClass [Java Application] (
true true false true



Java – klasa bazowa Object

- Pisząc własną metodę equals należy spełnić następujące własności:
 - **Zwrotność:** `x.equals(x)` powinno zwracać `true`, jeśli `x` nie ma wartości `null`.
 - **Symetria:** dla dowolnych referencji `x` i `y`, `x.equals(y)` powinno zwrócić wartość `true` wtedy i tylko wtedy, gdy `y.equals(x)` zwróci wartość `true`.
 - **Przemienność:** dla dowolnych referencji `x`, `y` i `z`, jeżeli `x.equals(y)` zwraca `true` i `y.equals(z)` zwraca `true`, to `x.equals(z)` zwraca też wartość `true`.



Java – klasa bazowa Object

- **Niezmiennność:** jeżeli obiekty, do których odwołują się `x` i `y` nie zmieniły się to kolejne wywołanie `x.equals(y)` musi zwrócić tę samą wartość.
- Dla każdego `x` różnego od `null`, wywołanie `x.equals(null)` powinno zwrócić wartość `false`.
- Powyższe reguły ułatwiają pisanie programów, ponieważ programista nie musi się zastanawiać czy wywołać `x.equals(y)` czy też może `y.equals(x)`.



Java – klasa bazowa Object

- **Kod mieszający** (ang. Hash code) to skrót do obiektu w postaci pochodzącej od niego liczby całkowitej.
- Kody mieszające powinny mieć różne wartości.
- Oznacza to, że jeżeli x i y to dwa, różne obiekty to powinno istnieć wysokie prawdopodobieństwo, że $x.hashCode()$ i $y.hashCode()$ to dwie różne liczby.
- Metoda `hashCode` znajduje się w klasie `Object`. Dzięki temu każdy obiekt ma domyślny kod mieszający, który jest uzyskiwany z adresu obiektu w pamięci.
- Jednak klasy dziedziczące po klasie `Object` bardzo często mają nadpisaną metodę `hashCode`.



Java – klasa bazowa Object

- Przykładowo klasa String oblicza hash code następująco:

```
int hash = 0;
for (int i = 0; i < length(); i++)
    hash = 31 * hash + charAt(i);
```

- Jak widać powyżej, wartość hash code obliczana jest na podstawie zawartości obiektu klasy String.

```
public class MainClass {
    public static void main(String[] args) {
        String s = "Lukasz";
        String k = "Lukasz";
        String z = "Lukasz1";
        String l = "Podstawy programowania w języku Java";
        System.out.println(s.hashCode()+" "+k.hashCode()+" "+
            +z.hashCode()+" "+l.hashCode());
    }
}
```

```
Console Problems Debug Shell
<terminated> MainClass [Java Application] C:\Program Files\Java
-2007815322 -2007815322 -2112732789 629983695
```



Java – klasa bazowa Object

- W przypadku przeddefiniowania metody **equals** należy również przeddefiniować metodą **hashCode** dla klasy.
- Metoda **hashCode** powinna zwracać liczbę całkowitą – może to być zarówno wartość dodatnia jak i ujemna.
- W celu zapewnienia, aby hash code dla różnych obiektów był różny, należy wykorzystać kombinację kodów mieszających pól tych obiektów.
- Jeżeli `x.equals(y)` to trzeba zapewnić, aby `x.hashCode()` oraz `y.hashCode()` były ze sobą równe.



Java – klasa bazowa Object

- Metoda **toString** również jest metodą pochodzącą z klasy **Object**.
- Metoda ta zwraca obiekt reprezentujący wartość obiektu.
- Większość metod **toString** ma następujący format: nazwa klasy plus wartości pól wymienione w nawiasach kwadratowych.

```
public class Employee {  
    private double salary;  
    private String name;  
    private String surname;  
    public Employee(double salary)  
    {  
        this.salary=salary;  
        this.name="";  
        this.surname="";  
    }  
  
    public double getSalary()  
    {  
        return this.salary;  
    }  
  
    public String toString()  
    {  
        return "Employee[salary="+salary+",name="+name+",surname="+surname+"]";  
    }  
}
```



Java – klasa bazowa Object

- Metodę **toString** można ulepszyć poprzez wykorzystanie **getClass().getName()** zamiast wpisywać na sztywno nazwę klasy.
- Dzięki takiej modyfikacji, **toString()** będzie również działało w klasach potomnych.

```
public class Employee {  
    private double salary;  
    private String name;  
    private String surname;  
    public Employee(double salary)  
    {  
        this.salary=salary;  
        this.name="";  
        this.surname="";  
    }  
  
    public double getSalary()  
    {  
        return this.salary;  
    }  
  
    public String toString()  
    {  
        return getClass().getName()+"[salary="+salary+",name="+name+", "  
            + "surname="+surname+"]";  
    }  
}
```



Java – klasa bazowa Object

- Programista piszący klasy dziedziczące powinien jednak zdefiniować własną metodę **toString**.
- Należy w niej uwzględnić pola, która są specyficzne dla klasy dziedziczącej.
- Jeżeli w nadklasie użyto wywołania `getClass().getName()` to w podklasie można wywołać **super.toString()**.



Java – klasa bazowa Object

```
public class Employee {
    private double salary;
    private String name;
    private String surname;
    public Employee(double salary)
    {
        this.salary=salary;
        this.name=" ";
        this.surname=" ";
    }

    public double getSalary()
    {
        return this.salary;
    }

    public String toString()
    {
        return getClass().getName()+"[salary="+salary+",name="+name+", "
            + "surname="+surname+"]";
    }
}
```

```
public class MainClass {
    public static void main(String[] args) {
        Employee e = new Employee(1000);
        Manager m = new Manager(1000,500);
        System.out.println(e.toString());
        System.out.println(m.toString());
    }
}
```

```
public class Manager extends Employee {

    double bonus;
    public Manager(double salary, double bonus)
    {
        super(salary);
        this.bonus=bonus;
    }
    public double getSalary()
    {
        double salary = super.getSalary();
        return this.bonus+salary;
    }

    public String toString()
    {
        return super.toString()+"[bonus="+bonus+"]";
    }
}
```

```
Console Problems Debug Shell
<terminated> MainClass [Java Application] C:\Program Files\Java\jdk-11\
Employee[salary=1000.0,name= ,surname= ]
Manager[salary=1000.0,name= ,surname= ][bonus=500.0]
```



Java – klasa bazowa Object

- Metoda **toString** jest bardzo często używana z jednego ważnego powodu: za każdym razem, gdy obiekt jest łączony z łańcuchem za pomocą operatora +, kompilator automatycznie wywołuje metodę **toString**, aby utworzyć łańcuchową reprezentację obiektu.
- Łączenie za pomocą operatora + najczęściej wykorzystywane jest przy wypisywaniu wartości na ekran za pomocą **System.out**.



Kolejny wykład:
Hierarchia klas,
pakiety oraz polimorfizm w języku Java.
Wprowadzenie do wzorców projektowych.



DZIĘKUJĘ ZA UWAGĘ

