PROGRAMOWANIE OBIEKTOWE JAVA – LABORATORIUM

PRZESŁANIANIE METOD (OVERRIDING), PRZECIĄŻANIE METOD, WYJĄTKI

Przesłanianie metod (Overriding)

Taka sama metoda w klasie bazowej i pochodnej

Wyobraźmy sobie sytuację, w której zarówno w klasie bazowej, jak i klasie pochodnej mamy metodę o tej samej sygnaturze. Dodatkowo niech obie metody mają albo dokładnie ten sam typ zwracanej wartości, albo niech metoda klasy pochodnej zwraca typ rozszerzający typ zwracany przez metodę w klasie bazowej.

Ten sam typ wartości zwracanej:

```
public class Item {

    String name;

    public void setName(String
name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

```
public class DocumentItem extends
Item {

    @Override
    public String getName() {
        return "Document Item";
    }
}
```

Typ rozszerzający (String) typ wartości zwracanej (Object):

```
public class Item {

    String name;

    public void setName(String
name) {
        this.name = name;
    }

    public Object getName() {
        return name;
    }
}
```

```
public class DocumentItem extends
Item {

    @Override
    public String getName() {
        return "Document Item";
    }
}
```

Właśnie zaimplementowano mechanizm nazywający się przesłonięciem metody. Metoda getName znajdująca się w klasie pochodnej przesłania metodę getName znajdującą się w klasie bazowej. Teraz zobaczmy, jakie to ma konsekwencje w kontekście wykonywanego programu.

```
public class Start {

    public static void main(String[] args) {

        Item item = new Item();
        item.setName("Simple Item");
        System.out.println(item.getName());
```

```
        Item item2 = new DocumentItem();
        item2.setName("Simple Item");
        System.out.println(item2.getName());
    }
}
```

Uruchamiając taki kod, najpierw wydrukujemy na konsoli tekst *"Simple Item"*, a następnie...*"Document Item"*. Dzieje się tak, mimo że wcześniej zarówno w obiekcie item jaki i w obiekcie item2 ustawiliśmy tę samą nazwę - "Simple Item". Powstaje więc pytanie, co się właściwie stało?

Odpowiedzią jest właśnie termin przesłonięcie metody. Zauważmy bowiem, że obiekt item, to instancja klasy Item, podczas gdy obiekt item2 to instancja klasy DocumentItem. Skoro więc przesłoniliśmy metodę getName w klasie DocumentItem, to w przypadku stworzenia obiektu tego typu, wykonana zostanie metoda przesłaniająca, a ona zwraca ustawiony na sztywno tekst "Document Item".

Super metoda w superklasie

Klasa bazowa nazywana jest czasem superklasą. Nie ma w tym nic dziwnego, bo żeby powstała jakakolwiek klasa pochodna musi istnieć najpierw klasa wzorcowa - superklasa. Termin ten przyda się teraz, aby zrozumieć, do czego służy w Javie słowo kluczowe o nazwie super.

Jeśli w podklasie chcemy wywołać kod przesłoniętej metody z superklasy, wówczas używamy słowa kluczowego super:

```
public class Item {

    String name;

    public void setName(String
name) {
        this.name = name;
    }

    public String getName() {
        System.out.println("Super
metoda w superklasie");
        return name;
    }
}
```

```
public class DocumentItem extends
Item {

    @Override
    public String getName() {
        super.getName();
        return "Document Item";
    }
}
```

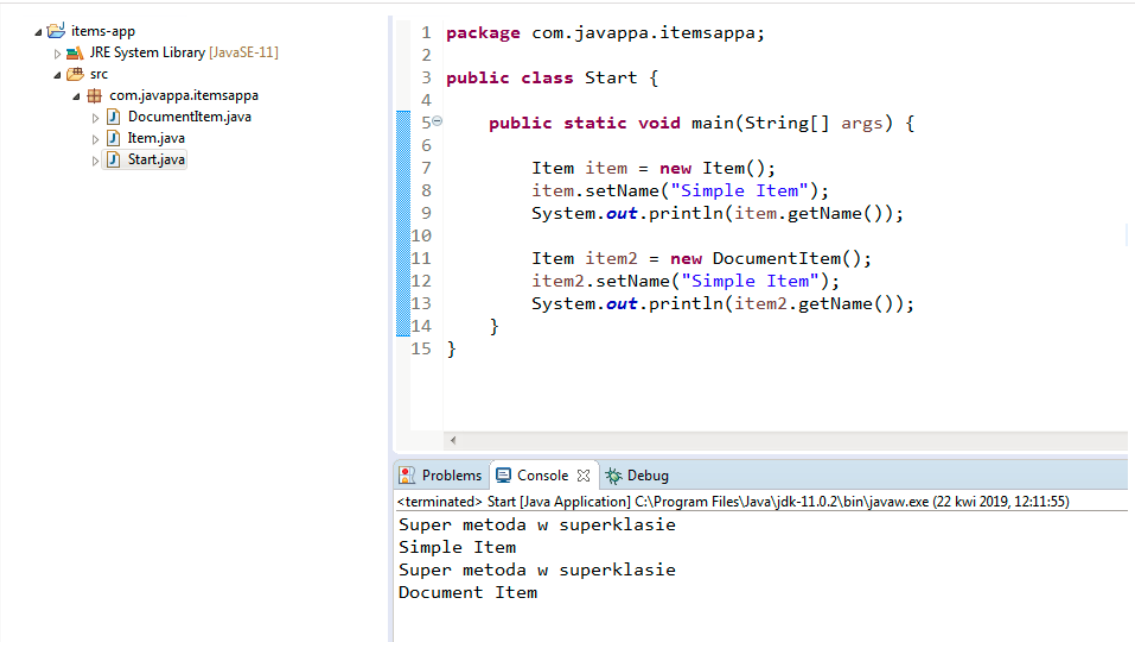
Wywołanie programu klasy Start (zdefiniowanej tak jak poprzednio) będzie szczególnie ciekawe w przypadku obiektu item2. Mamy tu do czynienia z instancją klasy DocumentItem, a to jak wiemy oznacza, że wywołujemy kod metody przesłaniającej. W obecnym przypadku pierwszą instrukcją tej metody jest "nakaz" wywołania metody z klasy bazowej (z superklasy). Z tego powodu proces wykonawczy programu przejdzie do metody getName w klasie Item i wydrukuje na konsoli tekst "Super metoda w superklasie".

Wykona się tu cała metoda getName, a więc jako wynik tej metody zwrócona zostanie nazwa ustawiona w polu name. Nie będzie to jednak miało wpływu na dalszą część programu, gdyż obecnie nie przechwytujemy tego, co zwraca ta metoda. Zaraz po tym proces wróci do klasy DocumentItem i zwróci jako wynik (wykonania metody przesłaniającej) tekst "Document Item", który następnie zostanie wydrukowany na konsoli.

Przykład

```
public class Start {

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

```
        Item item = new Item();
        item.setName("Simple Item");
        System.out.println(item.getName());

        Item item2 = new DocumentItem();
        item2.setName("Simple Item");
        System.out.println(item2.getName());
    }
}
```



Przeciążanie metod

Przeciążanie metod (ang. overloading) to tworzenie metod o tych samych nazwach ale różnych parametrach. Przeciążanie jest przydatne wtedy, kiedy chcemy stworzyć nową wersję tej samej metody.

Przykład:

```
public class Overloading
{
    public int sum(int a, int b) // parametry typu int
    {
        return a + b;
    }

    public double sum(double a, double b) // zmieniamy parametry na double
    {
        return a + b;
    }

    public static void main(String[] args)
    {
        Overloading o = new Overloading();
        System.out.println(o.sum(4.5,6)); // wynik = 10.5
    }
}
```

Pierwsza wersja metody sum przyjmuje dwa parametry a i b typu całkowitego int i zwraca zsumowaną wartość również tego samego typu. W drugiej metodzie (o tej samej nazwie) zmieniamy typ danych na double (liczba zmiennoprzecinkowa).

Konstruktor wywołujący inny konstruktor

W naszym przykładzie można łatwo zauważyć, że konstruktor przyjmujący jeden parametr w postaci identyfikatora, uruchamia konstruktor z większą listą parametrów. To jest dosyć często stosowana praktyka.

Przykładowo, jeśli tylko chcemy, możemy utworzyć obiekt klasy Item, korzystając z konstruktora jednoargumentowego, który sam ustawi domyślną wartość dla parametru z nazwą itema (name). Jeśli jednak wolimy własnoręcznie nadać nazwę, wtedy wykorzystujemy konstruktor z dwoma parametrami. Oczywiście możemy wprowadzać konstruktory również z większą ilością parametrów.

```
public class Item {

    int id;
    String name;
    String description;

    public Item(int id) {
        this(id, "Appa Item no. " + id);
    }

    public Item(int id, String name) {
        this.id = id;
        this.name = name;
    }

    ...
}
```

W trakcie wywołania jednego konstruktora z drugiego konstruktora nie podajemy nazwy konstruktora. W zamian za to używamy słowa kluczowego this. Inaczej jest w przypadku zwykłych metod.

Metoda wywołująca inną metodę

Wywołanie metody przeciążającej z poziomu metody przeciążonej nie wymaga użycia specjalnego słowa kluczowego. Wystarczy użyć nazwy docelowej metody z wybraną przez nas listą parametrów. Przyjrzyjmy się teraz dokładnie, jak to wygląda w przypadku kodu naszej klasy Item.

Najpierw definiujemy podstawową, bezparametrową wersję metody. W ramach niej uruchamiamy metodę przeciążającą, do której przekazujemy parametr (w postaci tekstu z opisem):

```
public void setDescription() {
     setDescription("Appa " + id + " description");
}
```

Przyjrzyjmy się wywoływanej metodzie przeciążającej. W ramach tej metody uruchamiamy kolejną metodę przeciążającą, tym razem dwuparametrową, gdzie drugi parametr określamy "na sztywno" (autor JavAPPa):

```
public void setDescription(String description) {
    setDescription(description, "JavAPPa");
}
```

Uruchomienie tej metody skutkuje już wykonaniem finalnej części procesu:

```
public void setDescription(String description, String author) {
    this.description = description + " created by " + author;
}
```

Wywołanie pierwszej, bezparametrowej metody, zawsze będzie ustawiało ten sam tekst oraz tego samego autora. Bezpośrednie wywołanie drugiej metody pozwoli zdefiniować tekst, ale autora nie będziemy musieli ustawiać. Jest to pomocne, gdy chcemy stworzyć kilka obiektów typu Item i każdy z nich ma mieć domyślnego autora. Nie ma wtedy sensu za każdym razem pisać tego samego:

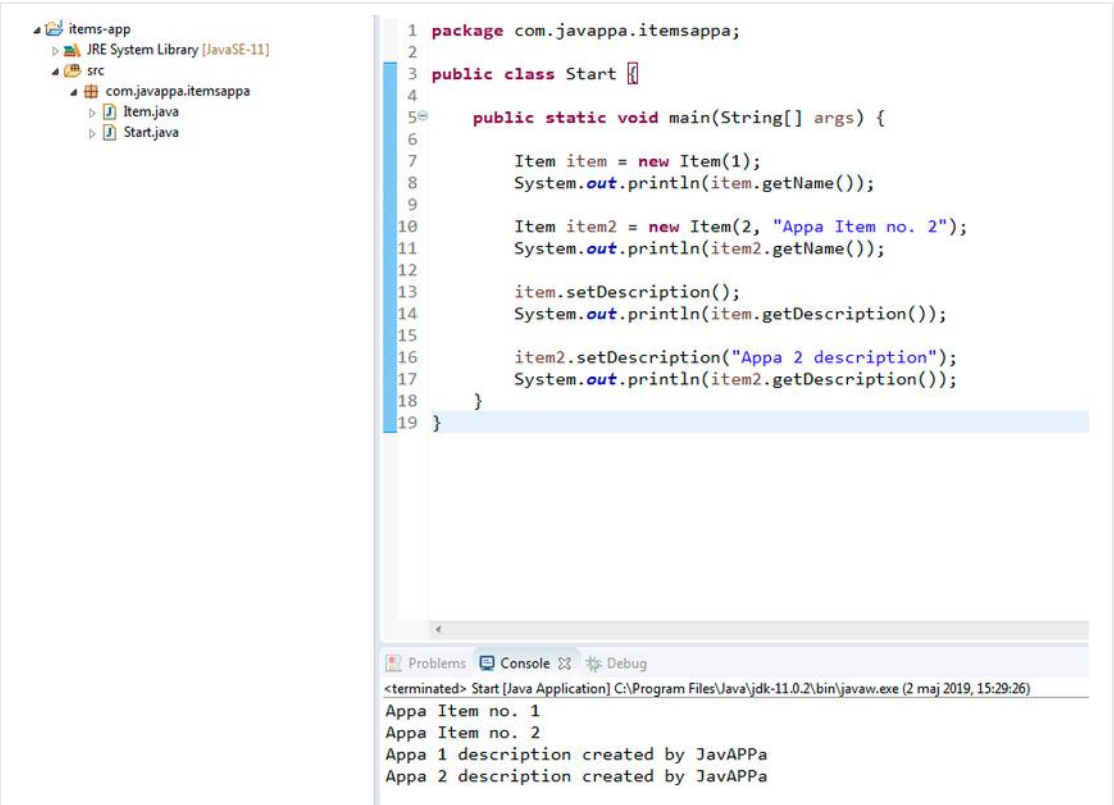
```
item1.setDescription("Some description...", "JavAPPa");
item2.setDescription("Another description...", "JavAPPa");
item3.setDescription("And another one...", "JavAPPa");
```

Lepiej wywołać naszą drugą metodę:

```
item1.setDescription("Some description...");
item2.setDescription("Another description...");
item3.setDescription("And another one...");
```

Przykład

```
public class Start {

    public static void main(String[] args) {

        Item item = new Item(1);
        System.out.println(item.getName());

        Item item2 = new Item(2, "Appa Item no. 2");
        System.out.println(item2.getName());

        item.setDescription();
        System.out.println(item.getDescription());

        item2.setDescription("Appa 2 description");
        System.out.println(item2.getDescription());
    }
}
```



Wyjątki

Wyjątek to sytuacja, w której program nie może zostać skompilowany z powodu wystąpienia w nim błędu (np.: złej składni, braku połączenia z serwerem, nie istniejącego pliku itd). Domyślnie Java wyświetla komunikaty o błędach, ale możemy sami przewidzieć niektóre sytuacje i im zapobiegać poprzez obsługę wyjątków.

Aby obsłużyć wyjątek stosuje się blok instrukcji try … catch:

```
try {

// kod programu ...

} catch (Exception ex) {

// komunikat o błędzie lub obsługa wyjątku ...

}
```

Dla przykładu napiszmy prosty programik do obliczania ilorazu liczb całkowitych. Z lekcji matematyki pamiętamy, że liczby nie można dzielić przez 0 i taki wyjątek zostanie zgłoszony w sytuacji podania przez użytkownika błędnego argumentu.

```
public class Wyjatki
{
    public int iloraz(int a, int b) { // oblicza iloraz dwóch liczb
całkowitych
        return a/b;
    }

    public static void main(String args[])
    {
        try // obsługa wyjątku
        {
            Wyjatki w = new Wyjatki();
            System.out.println(w.iloraz(10,0)); // podstawiamy argumenty:
10 i 0

        } catch (Exception ex)
        {
            System.out.println("Wprowadź poprawny dzielnik!");
        }
    }
}
```

Do bloków try oraz catch można dołożyć blok finalny deklarowany poprzez słowo kluczowe finally. Blok finalny wykona się niezależnie od tego, czy wyjątek wystąpi czy też nie.

```
public class Wyjatki
{
    public int iloraz(int a, int b) {
        return a/b;
    }

    public static void main(String args[])
    {
        try
        {
            Wyjatki w = new Wyjatki();
            System.out.println(w.iloraz(10,0));

        } catch (Exception ex)
        {
```

```
            System.out.println("Wprowadź poprawny dzielnik!");

        } finally
        {
            System.out.println("Kod finalny");
        }
    }
}

/* Wyświetli:
Wprowadź poprawny dzielnik!
Kod finalny */
```

Wyjątki w Runtime Exceptions

- **ArithmeticException** - Arithmetic error, such as divide-by-zero.
- **ArrayIndexOutOfBoundsException** - Array index is out-of-bounds.
- **ArrayStoreException** - Assignment to an array element of an incompatible type.
- **ClassCastException** - Invalid cast.
- **IllegalArgumentException** - Illegal argument used to invoke a method.
- **IllegalMonitorStateException** - Illegal monitor operation, such as waiting on an unlocked thread.
- **IllegalStateException** - Environment or application is in incorrect state.
- **IllegalThreadStateException** - Requested operation not compatible with the current thread state.
- **IndexOutOfBoundsException** - Some type of index is out-of-bounds.
- **NegativeArraySizeException** - Array created with a negative size.
- **NullPointerException** - Invalid use of a null reference.
- **NumberFormatException** - Invalid conversion of a string to a numeric format.
- **SecurityException** - Attempt to violate security.
- **StringIndexOutOfBounds** - Attempt to index outside the bounds of a string.
- **UnsupportedOperationException** - An unsupported operation was encountered.

Following is the list of Java Checked Exceptions Defined in java.lang.

- **ClassNotFoundException** - Class not found.
- **CloneNotSupportedException** - Attempt to clone an object that does not implement the Cloneable interface.
- **IllegalAccessException** - Access to a class is denied.
- **InstantiationException** - Attempt to create an object of an abstract class or interface.
- **InterruptedException** - One thread has been interrupted by another thread.
- **NoSuchFieldException** - A requested field does not exist.
- **NoSuchMethodException** - A requested method does not exist.

Zadania do samodzielnego rozwiązania:

Zadanie 1.

Napisz program składający się z dwóch klas. Pierwsza niech zawiera kilka metod o nazwie dodaj(), ale zwracających różne typy wyników i przyjmujących po minimum dwa parametry typów liczbowych wybranych przez Ciebie. Ich zadaniem jest zwrócenie, lub wyświetlanie sumy podanych argumentów. W drugiej klasie Testowej utwórz obiekt tej klasy i sprawdź działanie swoich metod, wyświetlając wyniki działań na ekranie. Dodatkowo każda z metod niech wyświetla swój typ zwracany i rodzaj argumentów, abyś wiedział, która z nich zadziałała.

Zadanie 2.

Napisz program, który pobierze od użytkownika liczbę i wyświetli jej pierwiastek. Do obliczenia pierwiastka możesz użyć istniejącej metody java.lang.Math.sqrt(). Jeśli użytkownik poda liczbę ujemną rzuć wyjątek java.lang.IllegalArgumentException. Obsłuż sytuację, w której użytkownik poda ciąg znaków, który nie jest liczbą

Zadanie 3.

Napisz metodę, która jako parametr będzie przyjmowała napis i wypisywała na standardowe wyjście jego długość.

1. Przekaż do tej metody null i zobacz, jaki wyjątek został zgłoszony.
2. Otocz wywołanie metody blokiem try-catch, przechwyć ten wyjątek i wypisz na standardowe wyjście ślad stosu wywołań z chwili zgłoszenia wyjątku.
3. Bezpośrednio po wypisaniu jego śladu zgłoś obsługiwany wyjątek ponownie.
4. Czy ślady stosu wypisane przez ciebie w bloku catch i przez maszynę wirtualną w chwili przerwania programu są takie same?
5. Przed zgłoszeniem ponownie obsługiwanego wyjątku wykonaj na nim metodę fillInStackTrace().
6. Zamiast zgłaszać ponownie obsługiwany wyjątek, zgłoś nowy wyjątek klasy Exception.
7. Dołącz obsługiwany wyjątek do nowo tworzonego wyjątku Exception jako przyczynę jego powstania.

Zadanie 4.

Stwórz klasę własnego wyjątku MyException. Stwórz klasę Samochod posiadającą pole predkosc oraz metodę przyspiesz(int), która przyspiesza/zwalnia o podaną wartość. W przypadku zwolnienia poniżej 0 ma zostać zgłoszony wyjątek. Stwórz klasy Osobowy i Ciezarowy dziedziczące po klasie Samochod, które zawierają redeklarację metody przyspiesz(int). W przypadku, gdy podany argument jest liczbą parzystą, wtedy predkosc=predkosc_max/argument (uwzględnić możliwość wystąpienia wyjątku ArithmeticException). W przeciwnym wypadku należy wywołać metodę przyspiesz(int) z klasy bazowej. Prędkość maksymalna jest stała i wynosi 150 dla obiektu klasy Ciezarowy i 250 dla obiektu klasy Osobowy. Przyspieszenie przez samochody powyżej prędkości maksymalnej ma spowodować zgłoszenie wyjątku MyException.

Zadanie 5.

Należy zaimplementować aplikację wykorzystującą: kompozycję i dziedziczenie, wykorzystać rzutowanie, wykorzystać this oraz super.

- Księgarnia, Podręcznik, Powieść, Klient, Książka
- Kino, Przygodowy, Romans, KinoManiak, Film