

Programowanie obiektowe w Javie

dzień 2

v3.1

Plan

1. Dziedziczenie
2. Zaawansowana obiektowość

Dziedziczenie

Dziedziczenie

Dziedziczenie jest sposobem na to, aby jedna klasa mogła przejmować funkcjonalność innej klasy.

Mówimy w takim przypadku, że klasa A dziedziczy po klasie B.

Klasa dziedzicząca ma wszystkie pola i metody klasy, którą rozszerza.

Gdy projektujemy relację dziedziczenia, należy sobie zadać pytanie, czy każda podklasa, może być też traktowana jak obiekt klasy rodzica?

Na przykład każdy ebook jest też książką, zatem ebook może dziedziczyć po książce.

Każdy kot perski jest też po prostu kotem, zatem kot perski może dziedziczyć po kocie.

Ale żaden kot nie jest psem, zatem kot nie może dziedziczyć po psie.

Dziedziczenie

Dziedziczenie deklarujemy podczas definiowania klasy.

Zapis schematyczny:

```
class A extends B
```

extends – to słowo kluczowe, którego używamy do zadeklarowania dziedziczenia

W przypadku dziedziczenia mówimy też często o relacji rodzic–dziecko.

W przypadku Javy rodzic może mieć wiele dzieci, ale dziecko może mieć tylko jednego rodzica.

- **Klasa bazowa (nadrzędna)** – bardziej ogólna.
- **Klasa dziedzicząca (potomna, pochodna)** – bardziej specjalistyczna.

Dziedziczenie

Dziedziczenie to podstawowy mechanizm programowania obiektowego.

Główne zalety:

- wielokrotne wykorzystanie klas,
- spójna hierarchia,
- łatwiejsza modyfikacja.

Klasa może być zarówno klasą bazową dla innych, jak i dziedziczącą.

```
class A {}  
class B extends A {}  
class C extends B {}
```

Dziedziczenie

Dziedziczenie to podstawowy mechanizm programowania obiektowego.

Główne zalety:

- wielokrotne wykorzystanie klas,
- spójna hierarchia,
- łatwiejsza modyfikacja.

Klasa może być zarówno klasą bazową dla innych, jak i dziedziczącą.

```
class A {}  
class B extends A {}  
class C extends B {}
```

A – klasa bazowa dla klasy **B**

B – klasa potomna z klasy **A**

Dziedziczenie

Dziedziczenie to podstawowy mechanizm programowania obiektowego.

Główne zalety:

- wielokrotne wykorzystanie klas,
- spójna hierarchia,
- łatwiejsza modyfikacja.

Klasa może być zarówno klasą bazową dla innych, jak i dziedziczącą.

```
class A {}  
class B extends A {}  
class C extends B {}
```

A – klasa bazowa dla klasy **B**

B – klasa potomna z klasy **A**

B – klasa bazowa dla klasy **C**

C – klasa potomna z klasy **B**

Dziedziczenie – przykład

Na poniższych przykładach widać korzyści płynące z wykorzystywania dziedziczenia.

Bez dziedziczenia:

```
public class Book {  
    public String name;  
    public double price;  
    public String author;  
}  
  
public class Ebook {  
    public String name;  
    public double price;  
    public String author;  
    public int sizeInMB;  
}
```

Z dziedziczeniem:

```
public class Book {  
    public String name;  
    public double price;  
    public String author;  
}  
  
public class Ebook extends Book {  
    public int sizeInMB;  
}
```

Dziedziczenie – przykład

Na poniższych przykładach widać korzyści płynące z wykorzystywania dziedziczenia.

Bez dziedziczenia:

```
public class Book {  
    public String name;  
    public double price;  
    public String author;  
}  
public class Ebook {  
    public String name;  
    public double price;  
    public String author;  
    public int sizeInMB;  
}
```

Tutaj nie używamy dziedziczenia i musimy deklarować od nowa wszystkie atrybuty.

Z dziedziczeniem:

```
public class Book {  
    public String name;  
    public double price;  
    public String author;  
}  
public class Ebook extends Book {  
    public int sizeInMB;  
}
```

Dziedziczenie – przykład

Na poniższych przykładach widać korzyści płynące z wykorzystywania dziedziczenia.

Bez dziedziczenia:

```
public class Book {  
    public String name;  
    public double price;  
    public String author;  
}  
  
public class Ebook {  
    public String name;  
    public double price;  
    public String author;  
    public int sizeInMB;  
}
```

Tutaj nie używamy dziedziczenia i musimy deklarować od nowa wszystkie atrybuty.

Z dziedziczeniem:

```
public class Book {  
    public String name;  
    public double price;  
    public String author;  
}  
  
public class Ebook extends Book {  
    public int sizeInMB;  
}
```

Tutaj dziedziczymy po klasie **Book**, musimy zatem dopisać tylko nowe atrybuty, bo przejmujemy wszystkie atrybuty istniejące w klasie **Book**.

Konstruktor przy dziedziczeniu

- Podczas tworzenia obiektu wywoływany jest bezargumentowy konstruktor.
- Konstruktor bez argumentów rodzica zostanie użyty domyślnie w klasie dziecka.
- Konstruktory z argumentami klas nadrzędnych nie zostaną użyte (chyba że jasno wymusimy ich użycie).
- Wywołanie konstruktora klasy bazowej realizujemy z wykorzystaniem słowa kluczowego **super**.

Przykład

```
public class Ebook extends Book {  
    public int sizeInMB;  
    public Ebook(){  
        super();  
    }  
}
```

Konstruktor przy dziedziczeniu

- Podczas tworzenia obiektu wywoływany jest bezargumentowy konstruktor.
- Konstruktor bez argumentów rodzica zostanie użyty domyślnie w klasie dziecka.
- Konstruktory z argumentami klas nadrzędnych nie zostaną użyte (chyba że jasno wymusimy ich użycie).
- Wywołanie konstruktora klasy bazowej realizujemy z wykorzystaniem słowa kluczowego **super**.

Przykład

```
public class Ebook extends Book {  
    public int sizeInMB;  
    public Ebook(){  
        super();  
    }  
}
```

W ten sposób wywołamy bezargumentowy konstruktor klasy **Book**.

Konstruktor bezargumentowy

```
public class Parent {  
    public Parent() {  
        System.out  
            .println("Parent constructor");  
    }  
}  
  
public class Child extends Parent{  
}
```

```
public class Main {  
    public static  
        void main(String[] args) {  
            Child child = new Child();  
        }  
}
```

Konstruktor bezargumentowy

```
public class Parent {  
    public Parent() {  
        System.out  
            .println("Parent constructor");  
    }  
}  
  
public class Child extends Parent{  
}
```

Konstruktor klasy **Parent** bez argumentów.

```
public class Main {  
    public static  
    void main(String[] args) {  
        Child child = new Child();  
    }  
}
```


Konstruktor bezargumentowy

```
public class Parent {  
    public Parent() {  
        System.out  
            .println("Parent constructor");  
    }  
}  
  
public class Child extends Parent{  
}
```

Konstruktor klasy **Parent** bez argumentów.

Klasa **Child** dziedziczy po **Parent**.

```
public class Main {  
    public static  
    void main(String[] args) {  
        Child child = new Child();  
    }  
}
```


Konstruktor bezargumentowy

```
public class Parent {  
    public Parent() {  
        System.out  
            .println("Parent constructor");  
    }  
}  
  
public class Child extends Parent{  
}
```

Konstruktor klasy **Parent** bez argumentów.

Klasa **Child** dziedziczy po **Parent**.

```
public class Main {  
    public static  
        void main(String[] args) {  
            Child child = new Child();  
        }  
}
```

Utworzenie obiektu **child** klasy **Child**.

Konstruktor bezargumentowy

```
public class Parent {  
    public Parent() {  
        System.out  
            .println("Parent constructor");  
    }  
}  
  
public class Child extends Parent{  
}
```

Konstruktor klasy **Parent** bez argumentów.

Klasa **Child** dziedziczy po **Parent**.

```
public class Main {  
    public static  
        void main(String[] args) {  
            Child child = new Child();  
        }  
}
```

Utworzenie obiektu **child** klasy **Child**.

Wynik na konsoli: **Parent constructor**

Konstruktor z parametrami

```
public class Book {  
    public String name;  
    public double price;  
    public String author;  
    public Book(String name,  
                 double price,  
                 String author) {  
        this.name = name;  
        this.price = price;  
        this.author = author;  
    }  
}
```

```
public class Ebook extends Book {  
    public int sizeInMB;  
    public Ebook(){  
        super("Thinking in JAVA",  
              95.99,  
              "Bruce Eckel");  
    }  
}
```

Konstruktor z parametrami

```
public class Book {  
    public String name;  
    public double price;  
    public String author;  
    public Book(String name,  
                  double price,  
                  String author) {  
        this.name = name;  
        this.price = price;  
        this.author = author;  
    }  
}
```

Konstruktor klasy **Book** ustawiający wartości atrybutów przy tworzeniu obiektu.

```
public class Ebook extends Book {  
    public int sizeInMB;  
    public Ebook(){  
        super("Thinking in JAVA",  
              95.99,  
              "Bruce Eckel");  
    }  
}
```

Konstruktor z parametrami

```
public class Book {  
    public String name;  
    public double price;  
    public String author;  
    public Book(String name,  
                  double price,  
                  String author) {  
        this.name = name;  
        this.price = price;  
        this.author = author;  
    }  
}
```

Konstruktor klasy **Book** ustawiający wartości atrybutów przy tworzeniu obiektu.

```
public class Ebook extends Book {  
    public int sizeInMB;  
    public Ebook(){  
        super("Thinking in JAVA",  
              95.99,  
              "Bruce Eckel");  
    }  
}
```

Wywołanie konstruktora klasy **Book**. Argumenty **name**, **price**, **author** są klasy nadrzędnej.

Konstruktor z parametrami

Mamy możliwość utworzenia konstruktora klasy podrzędnej – **Ebook**, który przyjmuje i ustawia wszystkie atrybuty klasy **Ebook**, z jednoczesnym wykorzystaniem konstruktora klasy nadrzędnej – **Book**.

```
public class Ebook extends Book {  
    public int sizeInMB;  
    public Ebook(String name, double price, String author, int sizeInMB) {  
        super(name, price, author);  
        this.sizeInMB = sizeInMB;  
    }  
}
```

Konstruktor z parametrami

Mamy możliwość utworzenia konstruktora klasy podrzędnej – **Ebook**, który przyjmuje i ustawia wszystkie atrybuty klasy **Ebook**, z jednoczesnym wykorzystaniem konstruktora klasy nadrzędnej – **Book**.

```
public class Ebook extends Book {  
    public int sizeInMB;  
    public Ebook(String name, double price, String author, int sizeInMB) {  
        super(name, price, author);  
        this.sizeInMB = sizeInMB;  
    }  
}
```

Konstruktor klasy **Ebook** ustawiający wartości atrybutów przy tworzeniu obiektu.

Konstruktor z parametrami

Mamy możliwość utworzenia konstruktora klasy podrzędnej – **Ebook**, który przyjmuje i ustawia wszystkie atrybuty klasy **Ebook**, z jednoczesnym wykorzystaniem konstruktora klasy nadrzędnej – **Book**.

```
public class Ebook extends Book {  
    public int sizeInMB;  
    public Ebook(String name, double price, String author, int sizeInMB) {  
        super(name, price, author);  
        this.sizeInMB = sizeInMB;  
    }  
}
```

Konstruktor klasy **Ebook** ustawiający wartości atrybutów przy tworzeniu obiektu.

Metoda **super** przekazuje wszystkie dane do klasy rodzica. Wywołanie konstruktora klasy nadrzędnej **Book**.

Nadpisywanie metod

Nadpisywanie/przedefiniowywanie niestatycznych metod (**overriding**) służy stworzeniu nowej funkcjonalności dla obiektów klasy podrzędnej.

W praktyce polega to na stworzeniu metody w klasie podrzędnej, która ma taką samą nazwę, listę parametrów oraz zwracany typ.

Nadpisywanie metod

Kod

```
class Book {  
    public String name;  
    public double price;  
    public String author;  
    public void printBook(){  
        System.out  
            .println("Drukowanie ...");  
    }  
}
```

Kod

```
public class Ebook extends Book {  
    public int sizeInMB;  
    public void printBook() {  
        System.out  
            .println("Drukowanie" +  
                "ebook");  
        super.printBook();  
    }  
}
```

Nadpisywanie metod

Kod

```
class Book {  
    public String name;  
    public double price;  
    public String author;  
    public void printBook(){  
        System.out  
            .println("Drukowanie ...");  
    }  
}
```

Kod

```
public class Ebook extends Book {  
    public int sizeInMB;  
    public void printBook() {  
        System.out  
            .println("Drukowanie" +  
                "ebook");  
        super.printBook();  
    }  
}
```

Słowo kluczowe **super** spowoduje wywołanie metody **printBook** z klasy **Book** (po której dziedziczy **Ebook**).

Nadpisywanie metod

Kod – wywołanie

```
public static void main(String[] args) {  
    Ebook ebook = new Ebook();  
    ebook.printBook();  
}
```

Wynik

```
Drukowanie ebook  
Drukowanie ...
```

Nadpisywanie metod

Kod – wywołanie

```
public static void main(String[] args) {  
    Ebook ebook = new Ebook();  
    ebook.printBook();  
}
```

Utworzenie obiektu klasy **Ebook**.

Wynik

```
Drukowanie ebook  
Drukowanie ...
```

Nadpisywanie metod

Kod – wywołanie

```
public static void main(String[] args) {  
    Ebook ebook = new Ebook();  
    ebook.printBook();  
}
```

Utworzenie obiektu klasy **Ebook**.

Wywołanie metody **printBook()**.

Wynik

```
Drukowanie ebook  
Drukowanie ...
```

Nadpisywanie metod

Kod – wywołanie

```
public static void main(String[] args) {  
    Ebook ebook = new Ebook();  
    ebook.printBook();  
}
```

Utworzenie obiektu klasy **Ebook**.

Wywołanie metody **printBook()**.

Wynik

```
Drukowanie ebook  
Drukowanie ...
```

Pochodzi z metody **printBook()** klasy **Ebook**.

Nadpisywanie metod

Kod – wywołanie

```
public static void main(String[] args) {  
    Ebook ebook = new Ebook();  
    ebook.printBook();  
}
```

Utworzenie obiektu klasy **Ebook**.

Wywołanie metody **printBook()**.

Wynik

```
Drukowanie ebook  
Drukowanie ...
```

Pochodzi z metody **printBook()** klasy **Ebook**.

Pochodzi z metody **printBook()** klasy **Book**.

Adnotacja @Override

Adnotacje są to specjalne konstrukcje, które pozwalają na przekazanie dodatkowych informacji na temat kodu.

Nad nadpisywaną metodą możemy umieścić adnotację o nazwie **@Override**.

Adnotacja ta nie jest wymagana, za jej pomocą informujemy kompilator, że nadpisujemy metodę.

Przykład

```
public class Ebook extends Book {  
    public int sizeInMB;  
  
    @Override  
    public void printBook() {  
        System.out.println("Drukowanie  
                             ebook");  
    }  
}
```

Adnotacja @Override

Przykład

```
public class Ebook extends Book {  
    public int sizeInMB;  
  
    @Override  
    public void printBook(int pages) {  
        System.out  
        .println("Drukowanie ebook:  
                pages" + pages);  
    }  
}
```

W przypadku oznaczenia taką adnotacją metody, która nie spełnia warunków nadpisania, np. zawiera inne parametry, otrzymamy wyjątek typu:

Exception in thread "main"
java.lang.Error: Unresolved
compilation problem: The method
printBook(int) of type Ebook must
override or implement a supertype
method

Adnotacja @Override

Przykład

```
public class Ebook extends Book {  
    public int sizeInMB;  
  
    @Override  
    public void printBook(int pages) {  
        System.out  
        .println("Drukowanie ebook:  
                pages" + pages);  
    }  
}
```

Metoda zawiera dodatkowy parametr typu **int**, którego nie ma metoda z klasy **Book**.

W przypadku oznaczenia taką adnotacją metody, która nie spełnia warunków nadpisania, np. zawiera inne parametry, otrzymamy wyjątek typu:

Exception in thread "main"
java.lang.Error: Unresolved
compilation problem: The method
printBook(int) of type Ebook must
override or implement a supertype
method

final

Oznaczenie metody słowem **final** powoduje, że klasy dziedziczące nie mogą jej przesłonić.

```
public class Book {  
    final void name() {  
        System.out.println("test1");  
    }  
}
```

Niedozwolona jest w związku z tym następująca konstrukcja:

```
public class Ebook extends Book {  
    final void name() {  
        System.out.println("test2");  
    }  
}
```

final

Final jest przydatny kiedy w naszej klasie nie chcemy by opracowany przez nas algorytm zmienił działanie w klasie pochodnej.

```
public class Chess {  
    final public String  
        getStartingPlayer() {  
        return "white";  
    }  
}
```

Niedozwolona jest w związku z tym następująca konstrukcja:

```
public class DarkChess extends Chess{  
    @Override  
    public String getStartingPlayer() {  
        return "black";  
    }  
}
```

final

Final jest też przydatny gdy używamy metody w konstruktorze. Spójrz na poniższy przykład:

```
public class Parent {
    public int parentData = 123;
    public Parent() {
        System.out
            .println("Parent constructor");
        showInfo();
    }
    public void showInfo() {
        System.out
            .println("Wartość parentData:"
                + parentData);
    }
}
```

Przeciążenie metody **showInfo()** jest nie zalecane, ponieważ doprowadzi do błędów w naszym kodzie.

```
public class Child extends Parent{
    public int childData = 42;
    public Child() {
        System.out
            .println("Child constructor");
    }
    @Override
    public void showInfo() {
        System.out
            .println("childData:"
                + childData);
    }
}
```

final

```
public class Main {  
    public static void main(String[] args) {  
        Child child = new Child();  
    }  
}
```

final

```
public class Main {  
    public static void main(String[] args) {  
        Child child = new Child();  
    }  
}
```

Tworzymy nowy obiekt klasy **Child**

final

```
public class Main {  
    public static void main(String[] args) {  
        Child child = new Child();  
    }  
}
```

Tworzymy nowy obiekt klasy **Child**

Zostanie użyty konstruktor bez argumentów rodzica - a w nim jest przesłonięta metoda **showInfo()**.

final

```
public class Main {  
    public static void main(String[] args) {  
        Child child = new Child();  
    }  
}
```

Tworzymy nowy obiekt klasy **Child**

Zostanie użyty konstruktor bez argumentów rodzica - a w nim jest przesłonięta metoda **showInfo()**.

Konstruktor rodzica nie ma w sobie zmiennej **childData** - skąd będzie wiedział, jaką wartość ma ta zmienna?

final

```
public class Main {  
    public static void main(String[] args) {  
        Child child = new Child();  
    }  
}
```

Tworzymy nowy obiekt klasy **Child**

Zostanie użyty konstruktor bez argumentów rodzica - a w nim jest przesłonięta metoda **showInfo()**.

Konstruktor rodzica nie ma w sobie zmiennej **childData** - skąd będzie wiedział, jaką wartość ma ta zmienna?

Spójrz na wynik na konsoli:

Parent constructor

childData:0

Child constructor

final

```
public class Main {  
    public static void main(String[] args) {  
        Child child = new Child();  
    }  
}
```

Tworzymy nowy obiekt klasy **Child**

Zostanie użyty konstruktor bez argumentów rodzica - a w nim jest przesłonięta metoda **showInfo()**.

Konstruktor rodzica nie ma w sobie zmiennej **childData** - skąd będzie wiedział, jaką wartość ma ta zmienna?

Spójrz na wynik na konsoli:

Parent constructor

childData:0

Child constructor

Rozwiązaniem tego problemu jest uczynienie metody **showInfo()** final

Przeciążanie metod

W jednej klasie nie mogą znajdować się dwie metody (lub więcej) o takiej samej nazwie i takich samych parametrach (nazwa i typ).

Możemy utworzyć w klasie kilka metod o takich samych nazwach, ale różnych parametrach.

Java rozpoznaje na tej podstawie, która z metod ma być wywołana.

```
public void printReview(int points) {  
    System.out.println(  
        "Książka " + this.name + ", ocena: " + points + " punktów.");  
}  
public void printReview(int points, String review) {  
    System.out.println("Książka " + this.name + ", ocena: "  
        + points + " punktów. Dodatkowo została przesłana recenzja: " + review);  
}
```

Przeciążanie metod

W zależności od tego z jakimi parametrami zostanie wywołana metoda, zostanie użyta odpowiednia jej implementacja.

Wywołanie metody, która nie przyjmuje parametrów:

```
ebook.printBook();
```

Wywołanie metody przyjmującej parametr typu **String**:

```
ebook.printBook("Druk");
```

Wywołanie metody przyjmującej parametr typu **int**:

```
ebook.printBook(333);
```

Zadania

Wykonaj zadania z działu

Dziedziczenie

Zaawansowana obiektoowość

Klasa Object

Wszystkie klasy zawsze są podklasami klasy **Object**. Dzięki temu możemy w każdej klasie korzystać z metod, które są zawarte w klasie **Object**:

```
Object newObject = new Ebook();
```

Object – referencja typu **Object**

= – znak przypisania referencji typu **Object** do obiektu klasy **Ebook**

new Ebook() – tworzymy obiekt klasy **Ebook**

```
newObject.toString();
```

Mimo, że nie deklarowaliśmy metody **toString()** w klasie **Ebook**, możemy wywołać ją na obiekcie klasy **Ebook**, ponieważ dziedziczy ona po klasie **Object**.

Klasa Object

Nie zadziała to jednak w drugą stronę – pojawi się błąd, jeśli spróbujemy wywołać dla tego obiektu metodę z klasy **Ebook**:

```
newObject.getName();
```

Wywołanie metody **getName()** klasy **Ebook** spowoduje wystąpienie poniższego wyjątku:

```
Exception in thread "main" java.lang.Error:  
Unresolved compilation problem:  
The method getName() is undefined for the type Object
```

Dzieje się tak dlatego, że metoda **getName()** pochodzi z klasy **Ebook**.

Klasa Object

Klasa **Object** zawiera metody dostępne we wszystkich klasach, są to m.in.:

Object clone()

Tworzy obiekt identyczny jak ten, na rzecz którego została wywołana.

boolean equals(Object obj)

Sprawdza równość obiektów.

Class getClass()

Zwraca klasę obiektu.

String toString()

Zwraca ciąg znaków opisujących obiekt.

Więcej informacji o metodach klasy **Object**, a także o niej samej można znaleźć pod adresem:

<https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html>

Metoda equals

Poniżej różnica w działaniu metody **equals** i operatora "==" przy porównywaniu dwóch obiektów:

Metoda **equals** sprawdza czy obiekty mają równą zawartość:

```
String s1 = new String("abc");  
String s2 = new String("abc");  
System.out.println(s1.equals(s2));  
//zwróci true
```

Operator "==" sprawdza, czy wskazują one na ten sam obiekt:

```
String s1 = new String("abc");  
String s2 = new String("abc");  
System.out.println(s1==s2);  
//zwróci false
```

Standardowe klasy **Javy** dostarczają implementacji metody **equals**. Jeśli chcemy mieć tę metodę w naszych klasach, musimy sami zadbać o jej dodanie.

Dokładniej o metodzie **equals** możesz poczytać w dokumentacji:

<https://docs.oracle.com/javase/8/docs/api/java/util/Objects.html#equals-java.lang.Object-java.lang.Object->

Operacje rzutowania

- Rzutowanie to jawna (określona przez programistę) konwersja jednego typu do drugiego.
- Rzutowanie typów prostych może powodować utratę danych.
- Rzutowanie wykonujemy, określając w nawiasach okrągłych typ, jaki chcielibyśmy otrzymać.

Schematyczny opis rzutowania:

zmienna = (typNaKtoryRzutuujemy) wartość;

```
int intValue = 12;  
short shortValue = (short) intValue;
```

Operacje rzutowania

- Rzutowanie to jawna (określona przez programistę) konwersja jednego typu do drugiego.
- Rzutowanie typów prostych może powodować utratę danych.
- Rzutowanie wykonujemy, określając w nawiasach okrągłych typ, jaki chcielibyśmy otrzymać.

Schematyczny opis rzutowania:

zmienna = (typNaKtoryRzutuujemy) wartość;

```
int intValue = 12;  
short shortValue = (short) intValue;
```

(short) – typ, na który chcemy rzutować.

Operacje rzutowania – utrata informacji

W pewnych przypadkach możemy się spotkać z utratą danych podczas rzutowania, np.:

➤ Przekroczenie zakresu:

rzutujemy maksymalną wartość typu **int** na typ **short** – co przekracza jego zakres.

Przykład:

```
int intValue = 2147483647;  
short shortValue = (short) intValue;  
System.out.println(shortValue);
```

➤ Utrata precyzji:

odcięcie części ułamkowej w przypadku rzutowania typu zmiennoprzecinkowego na całkowity.

Przykład:

```
double doubleValue = 11.20;  
int intVal = (int) doubleValue;  
System.out.println(intVal);
```

Operacje rzutowania – utrata informacji

W pewnych przypadkach możemy się spotkać z utratą danych podczas rzutowania, np.:

➤ Przekroczenie zakresu:

rzutujemy maksymalną wartość typu **int** na typ **short** – co przekracza jego zakres.

Przykład:

```
int intValue = 2147483647;  
short shortValue = (short) intValue;  
System.out.println(shortValue);
```

Otrzymamy wynik **-1** – oznacza to utratę informacji.

➤ Utrata precyzji:

odcięcie części ułamkowej w przypadku rzutowania typu zmiennoprzecinkowego na całkowity.

Przykład:

```
double doubleValue = 11.20;  
int intVal = (int) doubleValue;  
System.out.println(intVal);
```


Operacje rzutowania – utrata informacji

W pewnych przypadkach możemy się spotkać z utratą danych podczas rzutowania, np.:

➤ Przekroczenie zakresu:

rzutujemy maksymalną wartość typu **int** na typ **short** – co przekracza jego zakres.

Przykład:

```
int intValue = 2147483647;  
short shortValue = (short) intValue;  
System.out.println(shortValue);
```

Otrzymamy wynik **-1** – oznacza to utratę informacji.

➤ Utrata precyzji:

odcięcie części ułamkowej w przypadku rzutowania typu zmiennoprzecinkowego na całkowity.

Przykład:

```
double doubleValue = 11.20;  
int intVal = (int) doubleValue;  
System.out.println(intVal);
```

Otrzymamy wynik **11**.

Operacje rzutowania

Nie zawsze zamiana typu jest możliwa!

Poniższe próby rzutowania zwrócą błąd podczas kompilacji:

```
String str1 = "abc";  
char charValue = (char) str1;  
int number = (int) "123";  
char c = 'a';  
String str2 = (String) c;
```

```
Exception in thread "main" java.lang.Error:  
Unresolved compilation problems:  
Cannot cast from String to char  
Cannot cast from String to int  
Cannot cast from char to String
```

Słowo kluczowe static

Użycie static

Wszystkie przykłady, które pokazywaliśmy do tej pory, działały tylko przy użyciu obiektów.

Jest jednak możliwość wywoływania pewnych funkcji przy odwołaniu się do klasy, a nie do danego obiektu.

Taką funkcjonalność daje nam słowo kluczowe **static**.

Jeżeli dodamy do metody słowo kluczowe **static**, to będzie ona działała dla klasy.

Przyjęło się, że umieszczamy słowo **static** za modyfikatorem dostępu.

Metoda statyczna nie może się odnosić do zmiennej, która nie jest statyczna.

Metody static – podsumowanie

- Metodę możemy określić jako statyczną (nazywaną również klasową), dodając przed jej nazwą słowo kluczowe **static**.
- Metody statyczne można wywołać bez tworzenia obiektu.

Przykład:

```
System.out.println("test");
```

Zwróćmy uwagę, że nie definiujemy nowego obiektu przy użyciu słowa kluczowego **new**.

Deklarowanie metod statycznych – przykłady:

```
static void method1() {  
    System.out.println("Metoda nic" +  
        " nie zwraca");  
}  
  
static int method2() {  
    return 321;  
}  
  
static String method3() {  
    return "Coderslab";  
}
```

Metody static – wywoływanie

Zdefiniowaną metodę możemy wywołać na przykład w dowolnym miejscu metody **main**.

```
public static void main(String[] args) {  
    method1();  
}  
static void method1() {  
    System.out.println("Metoda nic nie zwraca");  
}  
static String method3() {  
    method1();  
    return "Coderslab";  
}
```

Metody static – wywoływanie

Zdefiniowaną metodę możemy wywołać na przykład w dowolnym miejscu metody **main**.

```
public static void main(String[] args) {  
    method1();  
}  
static void method1() {  
    System.out.println("Metoda nic nie zwraca");  
}  
static String method3() {  
    method1();  
    return "Coderslab";  
}
```

Wywołujemy metodę.

Metody static – wywoływanie

Zdefiniowaną metodę możemy wywołać na przykład w dowolnym miejscu metody **main**.

```
public static void main(String[] args) {  
    method1();  
}  
static void method1() {  
    System.out.println("Metoda nic nie zwraca");  
}  
static String method3() {  
    method1();  
    return "Coderslab";  
}
```

Wywołujemy metodę.

Definiujemy metodę.

Metody static – wywoływanie

Zdefiniowaną metodę możemy wywołać na przykład w dowolnym miejscu metody **main**.

```
public static void main(String[] args) {  
    method1();  
}  
static void method1() {  
    System.out.println("Metoda nic nie zwraca");  
}  
static String method3() {  
    method1();  
    return "Coderslab";  
}
```

Wywołujemy metodę.

Definiujemy metodę.

Wywołujemy metodę.

Metody static – wywoływanie

```
public static void main(String[] args) {  
    int variable = method2();  
}  
static int method2() {  
    return 321;  
}
```

Metody static – wywoływanie

```
public static void main(String[] args) {  
    int variable = method2();  
}  
static int method2() {  
    return 321;  
}
```

variable – tworzymy zmienną oraz przypisujemy jej wynik metody **method2()**

Metody static – wywoływanie

```
public static void main(String[] args) {  
    int variable = method2();  
}  
static int method2() {  
    return 321;  
}
```

variable – tworzymy zmienną oraz przypisujemy jej wynik metody **method2()**

method2() – wywołanie metody

Metody static – wywoływanie

```
public static void main(String[] args) {  
    int variable = method2();  
}  
static int method2() {  
    return 321;  
}
```

variable – tworzymy zmienną oraz przypisujemy jej wynik metody **method2()**

method2() – wywołanie metody

return 321 – definiujemy metodę

Metody static – wywoływanie

Z innych miejsc w kodzie, np. innych klas – wywołujemy metodę według schematu:

NazwaKlasy.nazwaMetody

Przykład:

```
class Book {  
    public static void showTitle() {  
        //ciało metody  
    }  
}  
class Author {  
    public static void main(String[] args) {  
        Book.showTitle();  
    }  
}
```

Słowo kluczowe static – atrybuty

Jeżeli dodamy słowo kluczowe **static** do atrybutu, to utworzymy zmienną, która jest współdzielona przez wszystkie obiekty danej klasy.

Wewnątrz klasy możemy się odnieść do takiej zmiennej po prostu przez jej nazwę:

```
number;
```

Z innych miejsc w kodzie przez konstrukcję:

NazwaKlasy.nazwaZmiennej

```
Book.number;
```

```
class Book {  
    static int number = 0;  
    public void increaseNumber() {  
        number++;  
        Book.number++;  
    }  
}
```

Słowo kluczowe static – atrybuty

Jeżeli dodamy słowo kluczowe **static** do atrybutu, to utworzymy zmienną, która jest współdzielona przez wszystkie obiekty danej klasy.

Wewnątrz klasy możemy się odnieść do takiej zmiennej po prostu przez jej nazwę:

```
number;
```

Z innych miejsc w kodzie przez konstrukcję:

NazwaKlasy.nazwaZmiennej

```
Book.number;
```

```
class Book {  
    static int number = 0;  
    public void increaseNumber() {  
        number++;  
        Book.number++;  
    }  
}
```

Wewnątrz klasy możemy się do jej metody statycznej odwołać na oba sposoby.

Zmienna liczba parametrów

Tworząc metodę, możemy zadeklarować, że przyjmuje ona dowolną liczbę parametrów określonego typu. Jest to konstrukcja zwana **varargs**.

Przykład:

```
displayText(String... words) {}
```

displayText – nazwa metody,

String – typ parametrów,

... – deklaracja dowolnej liczby parametrów,

words – zmienna przechowująca przekazane parametry.

Zmienna liczba parametrów

```
public static void displayText(String... words) {  
    for (String word : words) {  
        System.out.println(word);  
    }  
}
```

Wywołanie

```
displayText();  
displayText("Witaj");  
displayText("Witaj", "na");  
displayText("Witaj", "na", "kursie");  
displayText("Witaj", "na", "kursie", "Java");
```

Automatyczna konwersja typów prostych

W wyniku operacji arytmetycznych występować może automatyczna konwersja typów, której główne zasady zawierają się w czterech następujących punktach:

1. Wartości **byte**, **short** oraz **char** są konwertowane podczas obliczeń do typu **int**.
2. Jeśli jeden z elementów działania jest typu **long**, to całe działanie (pozostałe wartości) jest konwertowane do **long**.
3. Jeśli jeden z elementów działania jest typu **float**, to całość jest konwertowana do **float**.
4. Jeśli jeden z elementów jest typu **double**, to całość jest konwertowana do **double**.

Automatyczna konwersja typów prostych

Trzeba pamiętać o tym, że wynik działania musimy przypisać do poprawnego typu, inaczej może nastąpić utrata danych.

Poprawny zapis:

```
int a = 2;  
double b = 1.5;  
double c = b/a;
```

Niepoprawny zapis:

```
int a = 2;  
double b = 1.5;  
int c = b/a;
```

Przykład operacji

Zwróćmy uwagę na wyniki jakie otrzymamy wykonując poniższy kod:

```
double ex1 = 3/2;  
System.out.println(ex1);  
// 1.0  
double ex2 = 3/2.0;  
System.out.println(ex2);  
//1.5  
double ex3 = 3/2*2;  
System.out.println(ex3);  
//2.0  
double ex4 = 3.0/2*2;  
System.out.println(ex4);  
//3.0
```

Przykład operacji

Zwróćmy uwagę na wyniki jakie otrzymamy wykonując poniższy kod:

```
double ex1 = 3/2;  
System.out.println(ex1);  
// 1.0  
double ex2 = 3/2.0;  
System.out.println(ex2);  
//1.5  
double ex3 = 3/2*2;  
System.out.println(ex3);  
//2.0  
double ex4 = 3.0/2*2;  
System.out.println(ex4);  
//3.0
```

Obie strony operacji są typu **int**, więc wynik też jest typu **int**. Dopiero wynik operacji jest konwertowany na typ **double** – nastąpiła utrata danych.

Przykład operacji

Zwróćmy uwagę na wyniki jakie otrzymamy wykonując poniższy kod:

```
double ex1 = 3/2;  
System.out.println(ex1);  
// 1.0  
double ex2 = 3/2.0;  
System.out.println(ex2);  
//1.5  
double ex3 = 3/2*2;  
System.out.println(ex3);  
//2.0  
double ex4 = 3.0/2*2;  
System.out.println(ex4);  
//3.0
```

Obie strony operacji są typu **int**, więc wynik też jest typu **int**. Dopiero wynik operacji jest konwertowany na typ **double** – nastąpiła utrata danych.

Jedna ze stron operacji jest typu **double**, wynik też jest typu **double** – otrzymujemy poprawny wynik.

Przykład operacji

Zwróćmy uwagę na wyniki jakie otrzymamy wykonując poniższy kod:

```
double ex1 = 3/2;  
System.out.println(ex1);  
// 1.0  
double ex2 = 3/2.0;  
System.out.println(ex2);  
//1.5  
double ex3 = 3/2*2;  
System.out.println(ex3);  
//2.0  
double ex4 = 3.0/2*2;  
System.out.println(ex4);  
//3.0
```

Obie strony operacji są typu **int**, więc wynik też jest typu **int**. Dopiero wynik operacji jest konwertowany na typ **double** – nastąpiła utrata danych.

Jedna ze stron operacji jest typu **double**, wynik też jest typu **double** – otrzymujemy poprawny wynik.

Wynik dzielenia **3/2** wynosi **1**, pomnożony przez **2** – daje **2**. Dopiero teraz następuje konwersja na **double** – otrzymujemy niepoprawny wynik.

Przykład operacji

Zwróćmy uwagę na wyniki jakie otrzymamy wykonując poniższy kod:

```
double ex1 = 3/2;  
System.out.println(ex1);  
// 1.0  
double ex2 = 3/2.0;  
System.out.println(ex2);  
//1.5  
double ex3 = 3/2*2;  
System.out.println(ex3);  
//2.0  
double ex4 = 3.0/2*2;  
System.out.println(ex4);  
//3.0
```

Obie strony operacji są typu **int**, więc wynik też jest typu **int**. Dopiero wynik operacji jest konwertowany na typ **double** – nastąpiła utrata danych.

Jedna ze stron operacji jest typu **double**, wynik też jest typu **double** – otrzymujemy poprawny wynik.

Wynik dzielenia **3/2** wynosi **1**, pomnożony przez **2** – daje **2**. Dopiero teraz następuje konwersja na **double** – otrzymujemy niepoprawny wynik.

Poprawny wynik.

Przykład operacji

Zamiast dopisywać ".0" możemy na końcu wartości liczbowej wskazać jawnie jej typ, np.:

Dodajemy literę **d**, jeśli ma to być typ **double**:

```
double ex1 = 3/2d;
```

Dodajemy literę **f**, jeśli chcemy uzyskać **float**:

```
float someFloat = 123123123f;
```

Dodajemy literę **L** dla typu **long**:

```
long someLong = 123123123L;
```

Wielkość litery nie ma znaczenia, jednakże dla typu **long** stosujemy zazwyczaj wielką literę ze względu na podobieństwo małej litery **l** do liczby **1**.

Konwersja na String

Dość częstym przypadkiem jest konwersja wartości liczbowej do wartości **String**. Tego typu konwersję możemy wykonać na kilka sposobów:

```
int number = 123;  
String str1 = Integer.toString(number);  
String str2 = String.valueOf(number);  
String str3 = "value of number = " + number;
```

Konwersja na String

Dość częstym przypadkiem jest konwersja wartości liczbowej do wartości **String**. Tego typu konwersję możemy wykonać na kilka sposobów:

```
int number = 123;  
String str1 = Integer.toString(number);  
String str2 = String.valueOf(number);  
String str3 = "value of number = " + number;
```

Wykona się automatyczna konwersja.

Zadania

Wykonaj zadania z działu
Zaawansowana obiektowość