# Programowanie obiektowe w Javie

dzień 1

v3.1

# Plan

1. Idea programowania obiektowego
2. Obiekty i klasy

# Idea programowania obiektowego

# Programowanie proceduralne

Skrypt wywołuje funkcje, a osobne kawałki kodu mają za zadanie wykonać jakąś czynność na podstawie danych wejściowych. To jest właśnie programowanie proceduralne.

Ogólnie możemy opisać to jako proces składający się z następujących punktów:

- uruchom funkcję A, przekaż do funkcji – w postaci argumentów – wszystkie potrzebne dane,
- pobierz wynik funkcji A,
- uruchom funkcję B, przekaż do funkcji – w postaci argumentów – wszystkie potrzebne dane.

Niektóre języki – np. **PHP** – dopuszczają pisanie kodu w sposób proceduralny (czyli bez użycia klas i obiektów, a z użyciem tylko samych funkcji) lub połączenia programowania obiektowego z proceduralnym.

# Klasa a obiekt

**W programowaniu obiektowym rozdzielamy dwie główne idee: obiekty i klasy.**

## Klasa

Jest to schemat opisujący atrybuty i funkcje, które można wywołać na jakimś obiekcie. Można powiedzieć, że jest to szablon, względem którego tworzymy później poszczególne obiekty.

Przykładowo klasa samochód definiuje, że każdy samochód musi mieć markę, liczbę drzwi, moc silnika oraz funkcje przyspieszania, hamowania i skręcania.

## Obiekt

Jest to dokładna instancja danej klasy posiadająca wszystkie atrybuty, na której można wywoływać metody.

Przykładowo obiekt `redCar` ma markę Honda, czworo drzwi i moc silnika 120 KM.

# Obiekty

- Obiekty najczęściej pochodzą z rzeczywistości, np. obiekt – samochód.
- Taki obiekt ma swoje **właściwości** i **zachowania**.

| WŁAŚCIWOŚCI: | ZACHOWANIA: |
|---|---|
| kolor | przyspiesz |
| moc silnika | zwolnij |
| liczba miejsc | skręć w lewo |
| waga | zatrzymaj się |
| napęd | zredukuj bieg |

# Cztery założenia programowania obiektowego

**ABSTRAKCJA**

**DZIEDZICZENIE**

**HERMETYZACJA**

**POLIMORFIZM**

# Cztery założenia programowania obiektowego

## Abstrakcja

Odwołujemy się tutaj do założenia, że klasa ma „ukrywać” logikę, dzięki której otrzymujemy jakąś funkcjonalność.

Podobnie jak w przypadku samochodu – kierowca nie interesuje się silnikiem. Wie, że po wciśnięciu odpowiedniego pedału, samochód zwiększy prędkość.

## Dziedziczenie

Obiekty mogą po sobie dziedziczyć, przejmują swoje właściwości i funkcjonalności.

Na przykład obiekt klasy amfibia dziedziczy po klasie samochód. Dzięki temu wiemy, że obiekt amfibia będzie umiał jeździć (odziedziczył to po klasie samochód) i rozszerzamy go jeszcze o umiejętność pływania.

# Cztery założenia programowania obiektowego

## Hermetyzacja

Założenie to mówi, że klasa powinna ukrywać kod i dane przed niezaplanowanym użyciem.

Klasa powinna w pełni kontrolować swój stan.

Na przykład klasa samochód nie powinna pozwalać nam usuwać kół, hamulców, itp.

## Polimorfizm

Polimorfizm to możliwość podszywania się pod inne klasy.

Oznacza to, że pracując na jakimś interfejsie (zbiorze funkcjonalności), nie przejmujemy się dokładnym typem klasy, tylko tym, żeby miała te funkcjonalności.

Np. amfibia może podszywać się pod samochód (implementuje wszystkie jego funkcje).

# Dodatkowe założenia programowania obiektowego

Dwa założenia dopisane później mówią nam o zależnościach między obiektami.

## Generalizacja

Opisuje relacje między obiektami.

Na przykład owoc jest generalizacją zbioru: jabłko, gruszka, pomarańcza.

## Specjalizacja

Oznacza, że każda klasa obiektów powinna się w czymś specjalizować. Jeżeli klasa obiektów nie ma swojej specjalizacji, to powinna zostać wchłonięta przez klasę obiektów znajdujących się wyżej w hierarchii.

# Zalety programowania obiektowego

Programowanie obiektowe ma wiele zalet, m.in.:

- możliwość użycia naszego kodu w różnych projektach,
- wytwarzanie kodu łatwiejszego w utrzymaniu,
- lepsza abstrakcja kodu od problemu, dająca większą możliwość podmiany kodu przy zmianie założeń biznesowych,
- modularność kodu (umożliwiająca łatwiejszą podmianę części kodu przy zmianie systemów).

# Obiekty i klasy

# Nasza pierwsza klasa

W języku Java klasę definiujemy za pomocą słowa kluczowego **`class`**, po którym podajemy nazwę naszej klasy.

Nazwa musi zaczynać się od litery, ale może zawierać litery, cyfry i niektóre znaki specjalne (np. "_").

Kod tworzący klasę znajduje się w nawiasach klamrowych **{…}**. Jest to tak zwane ciało klasy.

**Definicja klasy o nazwie Book**

```
class Book {
    //tutaj znajdzie się ciało klasy
}
```

# Nasze pierwsze obiekty

Do tworzenia nowych obiektów używamy operatora "**new**".

Schematycznie możemy przedstawić to w poniższy sposób:

```
NazwaKlasy nazwaObiektu = new NazwaKlasy();
```

Tworzenie obiektu klasy Book:

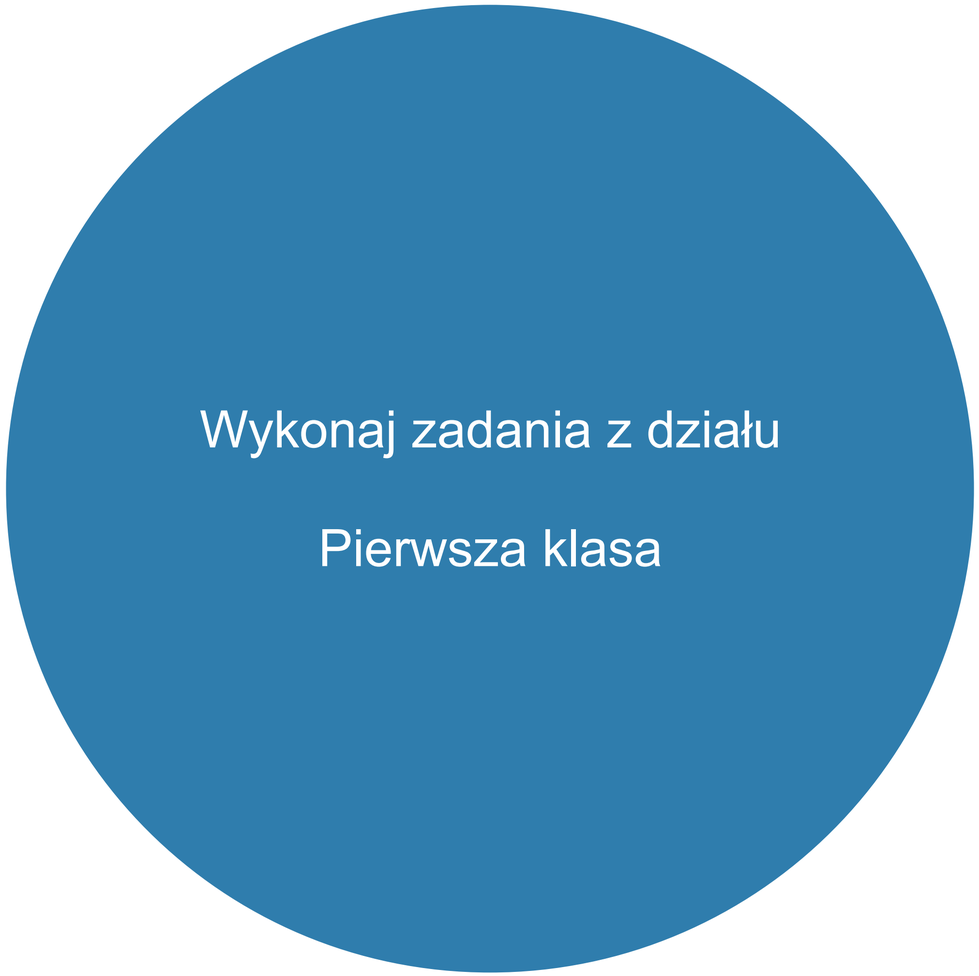
```
Book book = new Book();
```

Nowy obiekt jest tworzony, jest dla niego alokowana pamięć.

**Tworzenie zmiennych typów prostych nie wymaga operatora new.**

```
int myVal = 12;
```

# Zadania



# Modyfikatory dostępu

Klasy mogą mieć wewnętrzne zmienne (zwane atrybutami).

Atrybuty służą nam do trzymania danych w naszym obiekcie.

W Javie atrybuty poprzedzamy jednym z modyfikatorów dostępu do takiego atrybutu.

Mamy do wyboru:

- **`private`**
- **`protected`**
- **`public`**
- **`default`**

# Modyfikatory dostępu

**private**

Atrybut prywatny jest dostępny tylko z wnętrza danej klasy.

**protected**

Atrybut jest widoczny tylko z wnętrza danej klasy i klas dziedziczących oraz innych klas zawartych w pakiecie.

**public**

Atrybut publiczny jest dostępny z każdego miejsca.

**default**

Jeśli nie wpiszemy jawnie modyfikatora, atrybut będzie miał modyfikator domyślny (tzw. pakietowy).

# Modyfikatory dostępu

Poniższa tabela pokazuje dostęp do składowych dla każdego modyfikatora.

| Widoczność | Public | Protected | Default | Private |
|---|---|---|---|---|
| Ta sama klasa | T | T | T | T |
| Inna klasa z tego samego pakietu | T | T | T | N |
| Klasa pochodna z tego samego pakietu | T | T | T | N |
| Klasa pochodna z innego pakietu | T | T | N | N |
| Inna klasa z innego pakietu | T | N | N | N |

# Atrybuty klas

## Schematycznie definiowanie atrybutu

**`[public/private/protected] [static] [final] typ_zmiennej nazwa_zmiennej;`**

Nawiasy kwadratowe "**`[]`**" oznaczają, że dany element jest opcjonalny. Często w ten sposób oznaczamy elementy niewymagane.

```
class Book {
    public String name;
    public double price;
    public String author;
    private int catalogNumber;
    public String toString() {
        return "This is my Book";
    }
}
```

# Atrybuty klas

## Schematycznie definiowanie atrybutu

**`[public/private/protected] [static] [final] typ_zmiennej nazwa_zmiennej;`**

Nawiasy kwadratowe "**`[]`**" oznaczają, że dany element jest opcjonalny. Często w ten sposób oznaczamy elementy niewymagane.

```
class Book {
    public String name;
    public double price;
    public String author;
    private int catalogNumber;
    public String toString() {
        return "This is my Book";
    }
}
```

Te trzy atrybuty są dostępne wszędzie.

# Atrybuty klas

## Schematycznie definiowanie atrybutu

**`[public/private/protected] [static] [final] typ_zmiennej nazwa_zmiennej;`**

Nawiasy kwadratowe "**`[]`**" oznaczają, że dany element jest opcjonalny. Często w ten sposób oznaczamy elementy niewymagane.

```
class Book {
    public String name;
    public double price;
    public String author;
    private int catalogNumber;
    public String toString() {
        return "This is my Book";
    }
}
```

Te trzy atrybuty są dostępne wszędzie.

Ten atrybut będzie dostępny tylko z wnętrza klasy.

# Atrybuty: final i static

Atrybuty **`final`** i **`static`** były omawiane przez nas już w pierwszym tygodniu.
Poniżej krótkie przypomnienie ich znaczenia.

## Atrybut final

Gdy oznaczymy atrybut klasy słowem **`final`**, to otrzymamy zmienną, której wartości – po zainicjalizowaniu – nie można już zmodyfikować.

Wartość zazwyczaj nadajemy podczas deklaracji.

```
public class User {
    final int MIN_AGE = 18;
}
```

# Atrybuty: final i static

## Atrybut static

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

Przyjęło się, że to słowo kluczowe umieszczamy przed deklaracją typu zmiennej.

Atrybut określony jako **`static`** nazywamy klasowym.

```
class Book {
    static int number = 0;
}
```

# Atrybuty klas – operator kropki

Do atrybutów klas możemy się odwołać, używając operatora kropki: **"."**

**Przykład**

Tworzymy nowy obiekt klasy Book:

```
Book book = new Book();
String bookName = book.name;
```

Za pomocą operatora kropki odwołujemy się do atrybutu **`name`**, który posiada nasz utworzony obiekt **`book`**.

W naszym przykładzie właściwość **`name`** na razie będzie pusta, czyli **`null`** – aby to zmienić, należy ją zainicjować.

# Operator kropki – static

Z wnętrza klasy możemy się odnieść do takiej zmiennej po prostu przez wpisanie jej nazwy:

```
number;
```

Z innych miejsc w kodzie przez konstrukcję:

**NazwaKlasy.nazwaZmiennej**

```
Book.number;
```

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

# Operator kropki – static

Z wnętrza klasy możemy się odnieść do takiej zmiennej po prostu przez wpisanie jej nazwy:

```
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.

# Atrybuty klas – inicjalizacja

## Przykład

Tworzymy nowy obiekt klasy **Book** i przypisujemy wartości jego atrybutom:

```
Book book1 = new Book();
book1.name = "Rok 1984";
book1.price = 21.50;
book1.author = "George Orwell";
book1.catalogNumber = 12345;  //błąd
```

## Pobranie właściwości

```
String bookName = book1.name;
double price = book1.price;
String author = book1.author;
```

# Atrybuty klas – inicjalizacja

## Przykład

Tworzymy nowy obiekt klasy **Book** i przypisujemy wartości jego atrybutom:

```
Book book1 = new Book();
book1.name = "Rok 1984";
book1.price = 21.50;
book1.author = "George Orwell";
book1.catalogNumber = 12345;  //błąd
```

Błąd. Odwołujemy się do prywatnego atrybutu klasy (zakładając, że nie tworzymy obiektu klasy w niej samej).

## Pobranie właściwości

```
String bookName = book1.name;
double price = book1.price;
String author = book1.author;
```

# Atrybuty klas – wartości domyślne

## Kod

```
class Book {
    public String name = "Thinking" +
                         " in Java";
    public double price = 95.99;
    public String author = "Bruce" +
                           " Eckel"
    public int catalogNumber = 1;
}
```

**Atrybuty klas mogą mieć przypisane wartości domyślne.**

Wartości te będą ustawiane dla każdego obiektu podczas jego tworzenia.

## Pobranie właściwości:

```
Book book1 = new Book();
String bookName = book1.name;
double price = book1.price;
String author = book1.author;
```

## Wyświetlenie:

```
System.out.println(book1.name);
System.out.println(book1.price);
System.out.println(book1.author);
```

## Otrzymujemy:

```
Thinking in Java
95.99
Bruce Eckel
```

# Atrybuty klas – wartości domyślne

Wartości domyślne mogą zostać zmienione w taki sam sposób jak nadawaliśmy im wartość:

```
Book book2 = new Book();
book2.price  = 21.50;
book2.author  = "George Orwell";
```

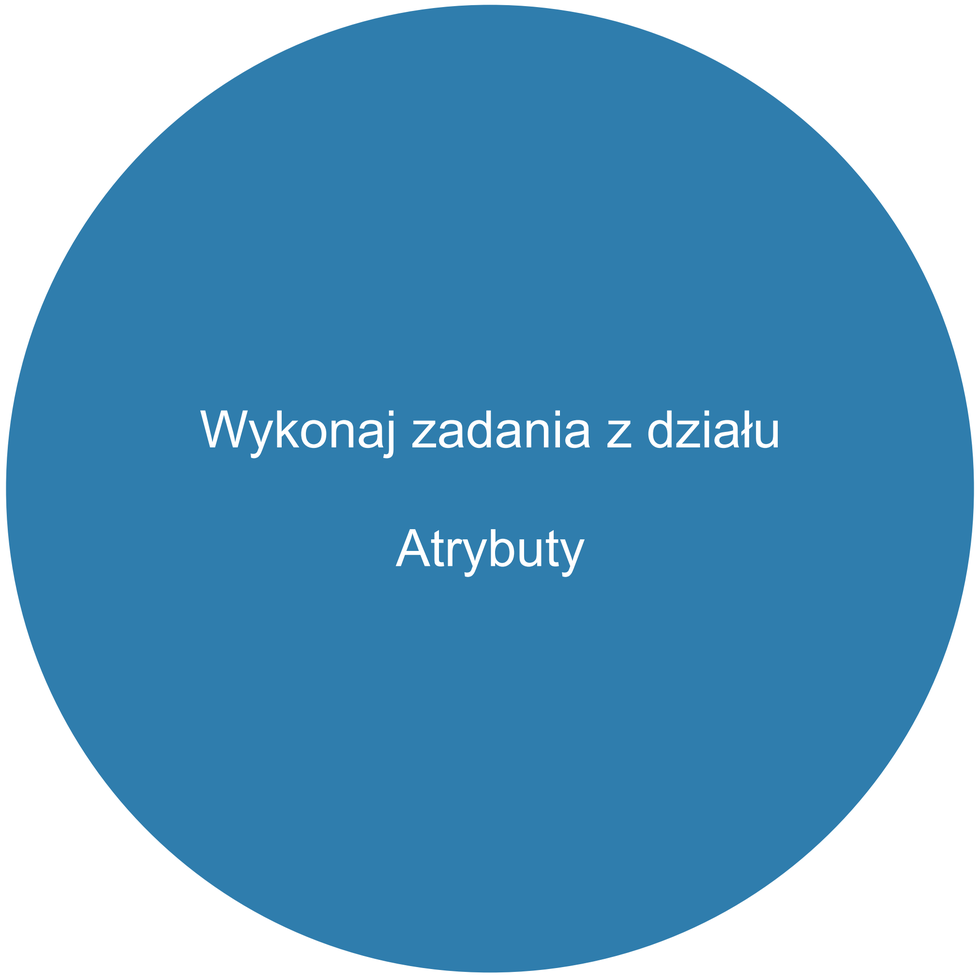
**Pobranie właściwości**

```
String bookName = book2.name;
double price = book2.price;
String author = book2.author;
```

Po wyświetleniu wartości w konsoli otrzymamy:

```
Thinking in Java
21.50
George Orwell
```

# Zadania



# Metody klas

Klasy mogą mieć wewnętrzne funkcje (zwane metodami).

Zarówno metody, jak i atrybuty poprzedzamy jednym z modyfikatorów dostępu do takiego atrybutu.

Metody mogą mieć takie same modyfikatory jak atrybuty.

**Statyczne metody nie mają dostępu do niestatycznych właściwości klasy.** Odpowiedni przykład zobaczymy na kolejnych slajdach.

```
class Book {
    public void printInfo() {
      //tutaj znajdzie się ciało metody
    }
}
```

# Metody klas

Klasy mogą mieć wewnętrzne funkcje (zwane metodami).

Zarówno metody, jak i atrybuty poprzedzamy jednym z modyfikatorów dostępu do takiego atrybutu.

Metody mogą mieć takie same modyfikatory jak atrybuty.

**Statyczne metody nie mają dostępu do niestatycznych właściwości klasy.** Odpowiedni przykład zobaczymy na kolejnych slajdach.

```
class Book {
    public void printInfo() {
        //tutaj znajdzie się ciało metody
    }
}
```

Publiczna metoda naszej klasy **Book**.

# Metody klas – słowo kluczowe this

W metodach mamy do dyspozycji nową pseudozmienną – **`this`**.

Umożliwia ona odnoszenie się do obiektu, na którym wywołaliśmy daną metodę.

Możemy o niej myśleć jak o zmiennej, w której znajduje się obiekt, na którym wywołaliśmy daną metodę.

- **`this`** – oznacza obiekt, na rzecz którego metoda jest wykonywana.
- do jego atrybutów również odwołujemy się przy użyciu kropki:

```
this.author
this.name
```

# Metody klas – słowo kluczowe this

```
class Book {
    public String name = "Thinking in Java";
    public double price = 95.99;
    public String author = "Bruce Eckel";
    public void printInfo() {
        String bookInfo = this.author + " " + this.name;
        System.out.println(bookInfo);
    }
}
```

# Metody klas – słowo kluczowe this

```
class Book {
    public String name = "Thinking in Java";
    public double price = 95.99;
    public String author = "Bruce Eckel";
    public void printInfo() {
        String bookInfo = this.author + " " + this.name;
        System.out.println(bookInfo);
    }
}
```

Tworzymy metodę korzystającą z atrybutów klasy.

# Metody klas – słowo kluczowe this

```
class Book {
    public String name = "Thinking in Java";
    public double price = 95.99;
    public String author = "Bruce Eckel";
    public void printInfo() {
        String bookInfo = this.author + " " + this.name;
        System.out.println(bookInfo);
    }
}
```

Tworzymy metodę korzystającą z atrybutów klasy.

**this** – oznacza obiekt, na którym wykonujemy metodę **printInfo()**

# Słowo kluczowe this i metoda static

**Błędne odwołanie:**

```
class Book {
    public String name = "Thinking in Java";
    public double price = 95.99;
    public String author = "Bruce Eckel";
    public static void printInfo() {
        String bookInfo = this.author + " " + this.name;
        System.out.println(bookInfo);
    }
}
```

# Słowo kluczowe this i metoda static

**Błędne odwołanie:**

```
class Book {
    public String name = "Thinking in Java";
    public double price = 95.99;
    public String author = "Bruce Eckel";
    public static void printInfo() {
        String bookInfo = this.author + " " + this.name;
        System.out.println(bookInfo);
    }
}
```

Taka definicja nie jest możliwa, gdyż odwołujemy się w statycznej metodzie do niestatycznych zmiennych.

# Słowo kluczowe this i metoda static

**Błędne odwołanie:**

```
class Book {
    public String name = "Thinking in Java";
    public double price = 95.99;
    public String author = "Bruce Eckel";
    public static void printInfo() {
        String bookInfo = this.author + " " + this.name;
        System.out.println(bookInfo);
    }
}
```

Taka definicja nie jest możliwa, gdyż odwołujemy się w statycznej metodzie do niestatycznych zmiennych.

Niestatyczne atrybuty.

# Wywoływanie metod

Metody naszej klasy możemy później wywołać na obiekcie tej klasy.

Robimy to poprzez użycie operatora kropki.

Należy jednak pamiętać, że metoda wywoływana na danym obiekcie zawsze będzie miała dostęp do wszystkich danych, które zawierają się w tym obiekcie.

```
Book book3 = new Book();
book3.printInfo();
```

# Wywoływanie metod

Metody naszej klasy możemy później wywołać na obiekcie tej klasy.

Robimy to poprzez użycie operatora kropki.

Należy jednak pamiętać, że metoda wywoływana na danym obiekcie zawsze będzie miała dostęp do wszystkich danych, które zawierają się w tym obiekcie.

```
Book book3 = new Book();
book3.printInfo();
```

Wywołanie metody **`printInfo()`** na obiekcie **`book3`**.

# Settery i gettery

Przy programowaniu obiektowym bardzo często używa się tzw. **`setterów`** i **`getterów`**.

Są to specjalne funkcje, dające nam dostęp do prywatnych atrybutów naszej klasy.

Służą one kontroli dostępu do prywatnych atrybutów klasy.
Dzięki temu będziemy spełniać założenia **enkapsulacji** (czyli jednej z podstaw obiektowości).

# Settery

**Settery** to metody, których celem jest ustawienie jakiegoś atrybutu klasy.

Zazwyczaj pisze się osobne metody do każdego atrybutu klasy, który pozwalamy zmieniać (a nie zawsze będziemy pozwalać zmieniać wszystkie z naszych atrybutów).

```
class Book {
    private String name;
    public void setName(String name) {
        this.name = name;
    }
}
```

# Settery

**Settery** to metody, których celem jest ustawienie jakiegoś atrybutu klasy.

Zazwyczaj pisze się osobne metody do każdego atrybutu klasy, który pozwalamy zmieniać (a nie zawsze będziemy pozwalać zmieniać wszystkie z naszych atrybutów).

```
class Book {
    private String name;
    public void setName(String name) {
        this.name = name;
    }
}
```

Zwyczajowo settery zaczynają się od słowa **set**, a potem mają nazwę atrybutu, na który wpływają.

# Metoda set i chaining

Przy okazji omawiania możliwości klasy **`StringBuilder`** wprowadziliśmy pojęcie **`chaining`**.

Możemy tak zdefiniować nasz setter, aby umożliwiał taką konstrukcję.

Na początku może dziwić, że settery zwracają obiekt, na którym wykonały jakąś zmianę.

Zwracanie obiektu, na którym pracujemy, daje nam możliwość wywołania następnej funkcji **`set`** w łańcuchu.

**Czasami jednak powoduje to trudność w znalezieniu błędu w kodzie (w jednej linijce jest wywołane kilka metod).**

# Metoda set i chaining

```
class Book {
    private String name;
    public Book setName(String name) {
        this.name = name;
        return this;
    }
}
Book book1 = new Book().setName("Thinking in Java");
```

# Metoda set i chaining

```
class Book {
    private String name;
    public Book setName(String name) {
        this.name = name;
        return this;
    }
}
Book book1 = new Book().setName("Thinking in Java");
```

Dzięki zwracaniu **`this`** z setterów możemy po kolei wywoływać je na jednym obiekcie.

# Gettery

**Gettery** to metody, które zwracają jakiś atrybut z obiektu. Dzięki nim możemy przeczytać, jakie obiekt ma wartości, ale bez możliwości wpływu na nie.

Zazwyczaj piszemy osobne gettery do każdego atrybutu, który chcemy pokazać na zewnątrz (nie zawsze będziemy chcieli pokazywać wszystkie atrybuty).

```
class Book {
    private String name;
    public String getName() {
        return name;
    }
}
```

# Gettery

**Gettery** to metody, które zwracają jakiś atrybut z obiektu. Dzięki nim możemy przeczytać, jakie obiekt ma wartości, ale bez możliwości wpływu na nie.

Zazwyczaj piszemy osobne gettery do każdego atrybutu, który chcemy pokazać na zewnątrz (nie zawsze będziemy chcieli pokazywać wszystkie atrybuty).

```
class Book {
    private String name;
    public String getName() {
        return name;
    }
}
```

Zwyczajowo gettery zaczynają się od słowa **get**, a potem mają nazwę atrybutu, który zwracają.

# Gettery

**`Gettery`** to metody, które zwracają jakiś atrybut z obiektu. Dzięki nim możemy przeczytać, jakie obiekt ma wartości, ale bez możliwości wpływu na nie.

Zazwyczaj piszemy osobne gettery do każdego atrybutu, który chcemy pokazać na zewnątrz (nie zawsze będziemy chcieli pokazywać wszystkie atrybuty).

```
class Book {
    private String name;
    public String getName() {
        return name;
    }
}
```

Zwyczajowo gettery zaczynają się od słowa **get**, a potem mają nazwę atrybutu, który zwracają.

Jedynym celem funkcji **`get`** jest zwrócenie danego atrybutu.

# Metoda toString()

Stworzenie nowego obiektu klasy **Book**:

```
Book book1 = new Book();
System.out.println(book1);
```

Do metody **println()** przekazujemy referencję do obiektu, co powoduje automatyczne wywołanie metody **toString()** tego obiektu.

Obecnie nasz obiekt nie ma własnej implementacji tej metody.

Co w takim razie wyświetli opisywana przez nas metoda **toString()**?

Metoda wyświetli wynik analogiczny do poniższego:

```
pl.coderslab.object.Book@1db9742
```

# Metoda toString()

Stworzenie nowego obiektu klasy **Book**:

```
Book book1 = new Book();
System.out.println(book1);
```

Do metody **println()** przekazujemy referencję do obiektu, co powoduje automatyczne wywołanie metody **toString()** tego obiektu.

Obecnie nasz obiekt nie ma własnej implementacji tej metody.

Co w takim razie wyświetli opisywana przez nas metoda **toString()**?

Metoda wyświetli wynik analogiczny do poniższego:

```
pl.coderslab.object.Book@1db9742
```

**pl.coderslab.object.Book** – pełna nazwa naszej klasy wraz z nazwą pakietu

# Metoda toString()

Stworzenie nowego obiektu klasy **Book**:

```
Book book1 = new Book();
System.out.println(book1);
```

Do metody **println()** przekazujemy referencję do obiektu, co powoduje automatyczne wywołanie metody **toString()** tego obiektu.

Obecnie nasz obiekt nie ma własnej implementacji tej metody.

Co w takim razie wyświetli opisywana przez nas metoda **toString()**?

Metoda wyświetli wynik analogiczny do poniższego:

```
pl.coderslab.object.Book@1db9742
```

**pl.coderslab.object.Book** – pełna nazwa naszej klasy wraz z nazwą pakietu

**1db9742** – ten element może się różnić – jest to unikalny wyróżnik, tzw. hash

# Metoda toString()

Jak to się stało, że niejawnie została wywołana metoda, której nie mieliśmy w naszej klasie?

Dzieje się tak dlatego, że wszystkie klasy dziedziczą (rozszerzają) metody klasy **`Object`**.

O dziedziczeniu dowiemy się więcej już niebawem.

Klasa **`Object`** ma implementację tej metody, więcej w dokumentacji: https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#toString--

```
public String toString()
```

Słowo **`public`** oznacza, że metoda jest publiczna – omówimy to w następnym module.

Krótko o metodzie **`toString()`**:

- służy do przedstawiania zawartości obiektów,
- powinna zwracać napis reprezentujący zawartość obiektu,
- możemy sami definiować, jak będzie wyglądała ta metoda.

# Metoda toString()

Dodamy do naszej klasy **Book** własną metodę **`toString()`**, a następnie wywołamy ją i sprawdzimy różnicę.

```
class Book {
    public String toString() {
        return "This is my Book.";
    }
}
```

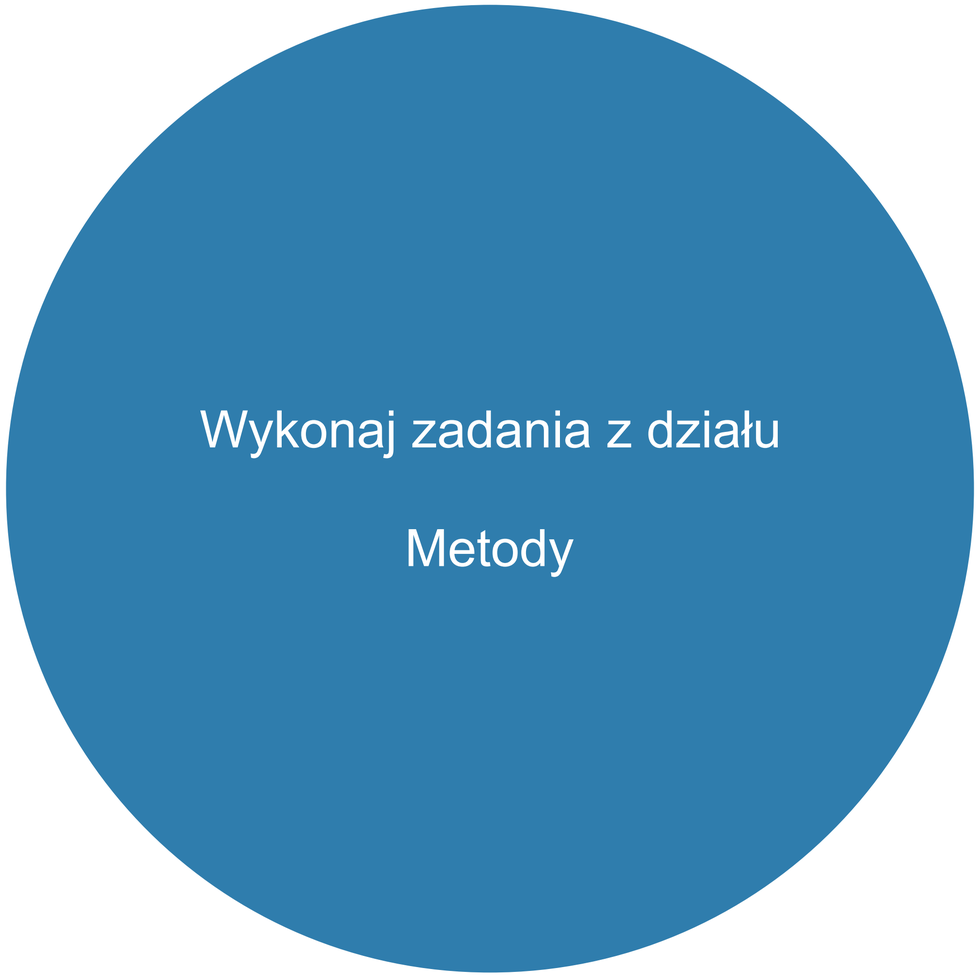
**Wywołanie:**

```
System.out.println(book1);
```

**Wynik:**

```
This is my Book.
```

# Zadania



# Konstruktor

**Konstruktor** to metoda, która jest wywoływana podczas tworzenia nowego obiektu.

Podstawowe cechy konstruktora:

- inicjuje obiekt,
- nie zwraca żadnej wartości,
- ma taką samą nazwę jak nazwa klasy,
- możemy mieć wiele konstruktorów.

- Dobrą praktyką jest przypisywanie wartości domyślnie w konstruktorze, a nie w ciele klasy.
- Jeżeli nie zdefiniujemy żadnego konstruktora, Java stworzy pusty konstruktor dla naszej klasy.
- Domyślny konstruktor nie przyjmuje żadnych argumentów.
- Po zdefiniowaniu konstruktora z argumentami, domyślny konstruktor nie będzie automatycznie dodawany.

# Konstruktor

## Schematyczna konstrukcja

```
[modyfikator dostępu] NazwaKlasy() {}
```

**`NazwaKlasy`** – nazwa konstruktora jest taka sama jak nazwa klasy;

**`()`** – argumenty przekazywane do konstruktora, np. **`(int a, int b)`**;

**`{}`** – zawartość metody konstruktora.

# Konstruktor

```
class Book {
    public Book() {
        System.out.println("Tworzenie obiektu klasy Book.");
    }
}
```

Konstruktor wywoła się podczas tworzenia każdego nowego obiektu klasy **Book**.

```
Book book1 = new Book();
```

# Konstruktor

```
class Book {
    public Book() {
        System.out.println("Tworzenie obiektu klasy Book.");
    }
}
```

Konstruktor wywoła się podczas tworzenia każdego nowego obiektu klasy **Book**.

```
Book book1 = new Book();
```

Zawartość metody konstruktora.

# Konstruktor z parametrami

Powszechną praktyką jest tworzenie konstruktorów, które otrzymują parametry, a następnie ustawiają atrybuty obiektu.

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

## Wywołanie konstruktora

```
Book book1 = new Book("Thinking in Java", 95.99, "Bruce Eckel");
```

# Prywatny konstruktor

- Prywatny konstruktor spowoduje sytuację, w której stworzenie klasy spoza metod tej klasy będzie niemożliwe.

- Jest to sytuacja poprawna, służąca do implementacji wzorca projektowego **Singleton**. O wzorcach projektowych będziemy się jeszcze uczyć w kolejnych modułach.

```
class Book {
    private Book(){}
}
```

# Referencje jako atrybuty klas

Oprócz typów standardowych Javy (np. **String**, **int**), atrybutami klas mogą być także np. referencje do obiektów naszych własnych klas czy tablice.

**Połączenia atrybutów różnych typów będą bardzo powszechne w naszych programach.**

**Tworzymy własną klasę Author**

```
public class Author {
}
```

```
class Book {
    public Author mainAuthor;
    public Author[] additionalAuthors;
}
```

# Referencje jako atrybuty klas

Oprócz typów standardowych Javy (np. **String**, **int**), atrybutami klas mogą być także np. referencje do obiektów naszych własnych klas czy tablice.

**Połączenia atrybutów różnych typów będą bardzo powszechne w naszych programach.**

**Tworzymy własną klasę Author**

```
public class Author {
}
```

```
class Book {
    public Author mainAuthor;
    public Author[] additionalAuthors;
}
```

Referencja do obiektu typu **`Author`**.

# Referencje jako atrybuty klas

Oprócz typów standardowych Javy (np. **String**, **int**), atrybutami klas mogą być także np. referencje do obiektów naszych własnych klas czy tablice.

**Połączenia atrybutów różnych typów będą bardzo powszechne w naszych programach.**

**Tworzymy własną klasę Author**

```
public class Author {
}
```

```
class Book {
    public Author mainAuthor;
    public Author[] additionalAuthors;
}
```

Tablica referencji do obiektów typu **`Author`**
(w przyszłości tablicę zastąpimy kolekcjami).

# Zadania