



# JAVA

Programowanie Obiektowe - przypomnienie



# KLASA I OBIEKTY

Klasa w programowaniu obiektowym jest szablonem, który definiuje właściwości (nazywane również polami lub atrybutami) i zachowania (metody) obiektów, które z niej powstaną. Klasy służą do tworzenia struktur danych, które mogą zawierać zarówno stan (dane) jak i akcje (metody), które mogą być wykonane na tych danych.



# KLASA I OBIEKTY

Obiekt jest instancją klasy.  
W Javie, obiekty są tworzone przy użyciu słowa kluczowego **new** i konstruktora klasy.



# KLASA I OBIEKTY

```
public class Samochod {  
    private String marka;  
    private String model;  
    private int rokProdukcji;  
  
    public Samochod(String marka, String model, int rokProdukcji) {  
        this.marka = marka;  
        this.model = model;  
        this.rokProdukcji = rokProdukcji;  
    }  
  
    public void wyswietl() {  
        System.out.println("Marka: " + marka + ", Model: " + model +  
            ", Rok Produkcji: " + rokProdukcji);  
    }  
}
```



# KLASA I OBIEKTY

```
public class Main {  
    public static void main(String[] args) {  
  
        Samochod mojSamochod = new Samochod("Toyota",  
                                              "Corolla", 2020);  
  
        mojSamochod.wyswietl();  
    }  
}
```



# MODYFIKATORY DOSTĘPU

Modyfikatory dostępu określają poziom dostępności klas, zmiennych, metod i konstruktorów w obrębie programu. Są one kluczowym elementem koncepcji enkapsulacji, pozwalając na kontrolowanie, które części programu mogą korzystać z określonych danych i metod.



# MODYFIKATORY DOSTĘPU

Elementy mogą  
być

## public

dostępne z każdego innego miejsca w programie, niezależnie od pakietu, w którym się znajduje

## protected

dostępne są w ramach tego samego pakietu, oraz dodatkowo w klasach dziedziczących

## default

dostępne są w ramach tego samego pakietu

## private

dostępne tylko w obrębie klasy, w której zostały zdefiniowane.

# SŁOWO KLUCZOWE `static`

`static` może być stosowane zarówno do pól (zmiennych), metod, jak i bloków inicjalizacyjnych w klasie, nadając im specjalny charakter statyczny. Elementy statyczne należą do samej klasy, a nie do jej instancji, co oznacza, że są wspólne dla wszystkich instancji tej klasy.





# SŁOWO KLUCZOWE static

```
public class Samochod {  
    public static int liczbaSamochodow = 0;  
    //...  
  
    public Samochod(String marka, String model,  
                     int rokProdukcji) {  
        liczbaSamochodow++;  
        //...  
    }  
  
    public static void main(String[] args) {  
  
        Samochod mojSamochod = new Samochod("Toyota", "Corolla", 2020);  
        Samochod mojSamochod2 = new Samochod("Toyota", "CHR", 2020);  
        System.out.println(Samochod.liczbaSamochodow);  
  
    }  
}
```



# GETTERY i SETTERY

Są to metody w Javie, które zapewniają kontrolowany dostęp do pól klasy.

**getter**

Umożliwiają one  
odczytywanie

**setter**

Umożliwiają one  
modyfikowanie

# GETTERY i SETTERY

Są to metody w Javie, które zapewniają kontrolowany dostęp do pól klasy.

**getter**

Umożliwiają one  
odczytywanie

```
public String getMarka() {  
    return marka;  
}  
  
public String getModel() {  
    return model;  
}  
  
public int getRokProdukcji() {  
    return rokProdukcji;  
}
```

**setter**

Umożliwiają one  
modyfikowanie

```
public void setMarka(String marka) {  
    this.marka = marka;  
}  
  
public void setModel(String model) {  
    this.model = model;  
}  
  
public void setRokProdukcji(int rokProdukcji) {  
    this.rokProdukcji = rokProdukcji;  
}
```

# Metoda toString()

Metoda **toString()** używa się aby uzyskać reprezentację tekstową obiektu, co jest szczególnie użyteczne podczas debugowania, logowania, czy prezentowania danych użytkownikowi.

```
@Override
public String toString() {
    return "Samochod{" +
        "marka='" + marka + '\\'' +
        ", model='" + model + '\\'' +
        ", rokProdukcji=" + rokProdukcji +
        '}';
}
```

# Słowo kluczowe final

1. **Zmienne finalne:** Zmienna oznaczona jako final może być przypisana tylko raz. Niekoniecznie musi być to zmienna stała w sensie, że jej wartość jest znana w czasie kompilacji, ale raz przypisana, nie może być zmieniona. Zmienna finalna musi zostać zainicjalizowana w czasie deklaracji lub w konstruktorze w przypadku pól obiektu.
2. **Metody finalne:** Metoda oznaczona jako final nie może być nadpisana przez żadną z klas pochodnych.
3. **Klasy finalne:** Klasa oznaczona jako final nie może mieć klas pochodnych, czyli nie można od niej dziedziczyć.

# Słowo kluczowe final

```
import java.util.ArrayList;
import java.util.List;

public class RepozytoriumSamochodow {

    private final List<Samochod> samochody;

    public RepozytoriumSamochodow() {
        this.samochody = new ArrayList<>();
    }

    public void dodajSamochod(Samochod samochod) {
        samochody.add(samochod);
    }

    public boolean usunSamochod(Samochod samochod)
    {
        return samochody.remove(samochod);
    }

    public List<Samochod> getSamochody() {
        return new ArrayList<>(samochody);
    }
}
```

# Interfejsy

pozwalają na definiowanie zestawu metod bez ich implementacji. Klasa implementująca interfejs musi dostarczyć implementację wszystkich metod zadeklarowanych w interfejsie.



# Interfejsy

```
public interface IRepozytoriumSamochodow {  
    void dodajSamochod(Samochod samochod);  
    boolean usunSamochod(Samochod samochod);  
    List<Samochod> getSamochody();  
}
```





# Interfejsy

```
public class RepozytoriumSamochodow implements IRepozytoriumSamochodow{  
    private final List<Samochod> samochody;  
  
    public RepozytoriumSamochodow() {  
        this.samochody = new ArrayList<>();  
    }  
    @Override  
    public void dodajSamochod(Samochod samochod) {  
        samochody.add(samochod);  
    }  
    @Override  
    public boolean usunSamochod(Samochod samochod) {  
        return samochody.remove(samochod);  
    }  
    @Override  
    public List<Samochod> getSamochody() {  
        return new ArrayList<>(samochody);  
    }  
}
```



# Dziedziczenie

pozwała na tworzenie nowej klasy na podstawie już istniejącej. Klasa potomna dziedziczy, czyli przejmuje pola i metody klasy nadrzędnej (bazowej), co umożliwia ponowne użycie kodu, oraz wprowadza hierarchię klas. Dzięki dziedziczeniu, możemy tworzyć bardziej złożone obiekty, które korzystają z ogólnych cech i zachowań klas bazowych, jednocześnie rozszerzając lub modyfikując je o specyficzne właściwości.



# Dziedziczenie

```
public abstract class Pojazd {
    private String marka;
    private String model;
    private int rokProdukcji;

    public Pojazd(String marka, String model, int rokProdukcji) {
        this.marka = marka;
        this.model = model;
        this.rokProdukcji = rokProdukcji;
    }

    public abstract void wyswietl();
    public String getMarka() {
        return marka;
    }

    public String getModel() {
        return model;
    }

    public int getRokProdukcji() {
        return rokProdukcji;
    }

    public void setMarka(String marka) {
        this.marka = marka;
    }

    public void setModel(String model) {
        this.model = model;
    }

    public void setRokProdukcji(int rokProdukcji) {
        this.rokProdukcji = rokProdukcji;
    }
}
```

# Dziedziczenie

```
public class SamochodPotomny extends Pojazd {

    public SamochodPotomny(String marka, String model, int rokProdukcji) {
        super(marka, model, rokProdukcji);
    }

    // Metoda klasy
    @Override
    public void wyswietl() {
        System.out.println("Marka: " + super.getMarka() + ", Model: " + super.getModel()+
            ", Rok Produkcji: " + super.getRokProdukcji() );
    }

    @Override
    public String toString() {
        return "SamochodPotomny{" +
            "marka='" + super.getMarka() + '\'' +
            ", model='" + super.getModel() + '\'' +
            ", rokProdukcji=" + super.getRokProdukcji() +
            '}';
    }
}
```

# Dziedziczenie

```
public class Bus extends Pojazd {
    private int miejscaSiedzace;
    public Bus(String marka, String model, int rokProdukcji, int miejscaSiedzace) {
        super(marka, model, rokProdukcji);
        this.miejscaSiedzace = miejscaSiedzace;
    }

    // Metoda klasy
    @Override
    public void wyswietl() {
        System.out.println("Marka: " + super.getMarka() + ", Model: " + super.getModel()+
            ", Rok Produkcji: " + super.getRokProdukcji() +" Miejsca siedzace: "+ this.miejscaSiedzace );
    }

    @Override
    public String toString() {
        return "Bus{" +
            "marka='" + super.getMarka() + '\'' +
            ", model='" + super.getModel() + '\'' +
            ", rokProdukcji=" + super.getRokProdukcji() +
            ", miejscaSiedzace=" + this.miejscaSiedzace +
            '}';
    }
}
```

# Polimorfizm

Polimorfizm w Javie pozwala metodzie w klasie potomnej na nadpisanie metody w klasie bazowej. Metoda w klasie potomnej musi mieć tę samą nazwę, listę parametrów i typ zwracany co metoda w klasie bazowej. W czasie wykonania, JVM decyduje, która wersja metody ma zostać wywołana, na podstawie rzeczywistego typu obiektu, co pozwala na wykonanie różnych wersji metody w zależności od typu obiektu.



# Klasa abstrakcyjna

Klasa abstrakcyjna jest klasą, która nie może być bezpośrednio zinstancjonowana. Zamiast tego, służy jako klasa bazowa dla innych klas, które dziedziczą po niej i implementują jej abstrakcyjne metody. Klasy abstrakcyjne mogą zawierać pola, konstruktory metody abstrakcyjnych oraz metody z pełną implementacją.



# Przykład użycia interfejsów oraz dziedziczenia – repozytorium pojazdów:

```
public interface IRepozytoriumPojazdow {  
    void dodajPojazd(Pojazd pojazd);  
  
    boolean usunPojazd(Pojazd pojazd);  
  
    List<Pojazd> getPojazdy();  
}
```

```
public class RepozytoriumPojazdow implements IRepozytoriumPojazdow {  
    private final List<Pojazd> samochody;  
  
    public RepozytoriumPojazdow() {  
        this.samochody = new ArrayList<>();  
    }  
    @Override  
    public void dodajPojazd(Pojazd samochod) {  
        samochody.add(samochod);  
    }  
    @Override  
    public boolean usunPojazd(Pojazd samochod) {  
        return samochody.remove(samochod);  
    }  
    @Override  
    public List<Pojazd> getPojazdy() {  
        return new ArrayList<>(samochody);  
    }  
}
```



# Przykład użycia interfejsów oraz dziedziczenia – repozytorium pojazdów:

```
public class Main {  
    public static void main(String[] args) {  
        IRepozytoriumPojazdow repozytoriumPojazdow = new RepozytoriumPojazdow();  
        Pojazd sp = new SamochodPotomny("Audi", "A4", 2019);  
        SamochodPotomny sp2 = new SamochodPotomny("Audi", "A3", 2017);  
        Bus bus = new Bus("Ford", "Transit", 2017, 9);  
        repozytoriumPojazdow.dodajPojazd(sp);  
        repozytoriumPojazdow.dodajPojazd(sp2);  
        repozytoriumPojazdow.dodajPojazd(bus);  
        for (Pojazd p : repozytoriumPojazdow.getPojazdy()){  
            p.wyswietl();  
        }  
    }  
}
```

# Dziękuję za uwagę!

CREDITS: This presentation template was created by Slidesgo, including icons by Flaticon, and infographics & images by Freepik.

**Please, keep this slide for  
attribution.**

