Klasy

Klasy to wzorzec do tworzenia obiektów. Klasy mogą zawierać: **zmienne – pola składowe**, **metody**, **inne klasy (klasa wewnętrzna)**

Klasa ma przynajmniej jeden konstruktor domyślny **Klasa()** z pustą listą argumentów i pustym ciałem.

Konstruktor to metoda klasy uruchamiana w momencie tworzenia obiektu. To on jest odpowiedzialny za zainicjalizowanie obiektu klasy. Musi mieć taką samą nazwę jak nazwa klasy. Może być przeciążany. Nawet jeśli nie zdefiniujemy własnego konstruktora to i tak klasa **dostaje domyślny,** bezparametrowy konstruktor z automatu. Ma on puste ciało. Konstruktor uruchamia się pisząc **new.**

UWAGA: Jeśli napiszemy własny konstruktor, to nie mamy już konstruktora domyślnego. Musimy go wtedy już sami napisać.

Konstruktor można również wywołać używając słowa this(). **this** – obiekt na rzecz którego wywołujemy np. metodę

obiekt – to instancja klasy, referencja do bieżącego obiektu, to wypełnienie konkretnymi wartościami wzorca klasy.

metody to zachowania klas.

przesłanianie metod to przedefiniowanie na nowo w klasie podrzędnej metody, którą klasa odziedziczyła z klasy nadrzędnej (bazowej)

przeciążanie metod to definiowanie metody o tej samej nazwie i wyniku, ale o innej liście parametrów.

Klasy wewnętrzna

Standardowe klasy wewnętrzne

Klasa wewnętrzna ma dostęp do wszystkich atrybutów czy metod klasy zewnętrznej, w której została zdefiniowana.

https://github.com/idzikpro/JavaCore/blob/master/src/main/java/pl/idzikpro/classes/OuterClass.java

```
public class OuterClass {
   public class InnerClass{

   }
   public InnerClass getInner(){
      return new InnerClass();
   }
}
```

Jak zainicjalizować klasę wewnętrzną?

https://github.com/idzikpro/JavaCore/blob/master/src/main/java/pl/idzikpro/classes/InnerClassMain.java

```
public static void main(String[] args) {
   OuterClass outerClass = new OuterClass();

   OuterClass.InnerClass inner1 = outerClass.getInner();
   OuterClass.InnerClass inner2 = outerClass.new InnerClass();
}
```

Statyczne klasy wewnętrzne

Są to klasy wewnętrzne poprzedzone modyfikatorem static.

```
public class OuterClass {
   public class InnerClass{

   }
   public static class StaticInnerClass{

   }

   public StaticInnerClass getStaticInner(){
      return new StaticInnerClass();
   }
}
```

A jak zainicjalizować ?

```
OuterClass outerClass = new OuterClass();

OuterClass.StaticInnerClass staticInner1=outerClass.getStaticInner();
OuterClass.StaticInnerClass staticInner2=new OuterClass.StaticInnerClass();
```

Lokalne klasy wewnętrzne

Możemy je definiować wewnątrz bloku (metody, instrukcji warunkowej). Nie poprzedzają jej żadne modyfikatory dostępu. Te klasy są dostępne tylko w tym bloku, w którym zostały zadeklarowane.

Dzięki klasom wewnętrznym jest poprawiona hemetyzacja klasy.

Anonimowe klasy wewnętrzne.

Klasy anonimowe to klasy definiowane w kodzie, które mają dokładnie jedną instancję. Klasa anonimowa nie tworzy instancji interfejsu. Kompilator tworzy nową klasę, która implementuje dany interfejs. Klasa stworzona przez kompilator tak naprawdę ma także nazwę i można ją np. wyświetlić.

Wewnątrz definicji wszystkich klas wewnętrznych można używać zmiennych lokalnych z otaczającego je kodu.

Przykład klasy anonimowej

```
Runnable runnable2=new Runnable() {
   @Override
   public void run() {
      IntStream.rangeClosed(0,100)
            .forEach(i->System.out.println(i+" running "+MyThread.currentThread().getName()));
   }
};
```

Klasy statyczne

To takie klasy, których instancji nie można utworzyć. Nie można także po niej dziedziczyć.

Kod statyczny, blok static itp.

https://github.com/idzikpro/JavaCore/blob/master/src/main/java/pl/idzikpro/classes/StaticBlocks.java

```
public class StaticBlocks {
  {
    System.out.println("block");
  }

  static {
    System.out.println("static block");
  }

  public StaticBlocks() {
    System.out.println("constructor");
  }

  public static void main(String[] args) {
    StaticBlocks staticBlocks=new StaticBlocks();
  }
}
```

Co pierwsze będzie wypisane?

- static block
- konstruktor klasy, z której dziedziczymy (nie ma na powyższym screenie)
- block
- inicjalizacja zmiennych statycznych klasy (nie ma na powyższym screenie)
- constructor

OOP – programowanie obiektowe

Paradygmat programowania, w którym programy definiuje się za pomocą obiektów, które komunikują się ze sobą w celu wykonania zadania. Przypomina to bardziej rzeczywistość w której żyjemy. W innym podejściu proceduralnym dane i funkcje nie były ze sobą związane.

Podstawie założenia OOP: abstrakcja, hermetyzacja, polimorfizm, dziedziczenie.

Abstrakcja

Abstrakcja to uproszczenie rozwiązywanego problemu w taki sposób, by objąć kluczowe cechy danego obiektu, niezależnie od implementacji.

Dziedziczenie

Dziedziczenie to mechanizm, dzięki któremu klasa może posiadać cechy innej klasy - definiowanie bardziej szczegółowych typów na podstawie tych bardziej ogólnych

metoda super() – odwołuje się do metod z klasy bazowej.

Może być np. **super.nazwaMetody()** albo **super()**

Zalety dziedziczenia

- Mniej kodu do przepisania
- Polimorfizm (patrz dalej)

Wady dziedziczenia

- Można dziedziczyć tylko z jednej klasy.
- Zmiana pewnych idei w klasie bazowej prowadzi do wymuszonych zmian we wszystkich klasach dziedziczących bo są one przecież ściśle powiązane. Sposobem na poradzenie sobie z tym problemem może być **kompozycja**.

Polimorfizm

Polimorfizm to mechanizm pozwalający programiście używać obiektów lub metod na kilka różnych sposobów. Polimorfizm to inaczej **wielopostaciowość** czyli traktowanie obiektów różnych typów w taki sam sposób, np. obiekty typu Pies jak typ Animal.

Wszelkie zmiany typów (jawne i niejawne), statyczne i dynamiczne należy również traktować jako mechanizmy polimorficzne.

Przykład polimorfizmu

https://github.com/idzikpro/JavaCore/blob/master/src/main/java/pl/idzikpro/classes/Animal.java

```
Animal[] animals={new Cat(), new Dog()};
for (Animal animal:animals
    ) {
   if (animal instanceof Cat){
      Cat catAnimal=((Cat) animal);
      catAnimal.saySth();
   }
   if (animal instanceof Dog){
      Dog dogAnimal=((Dog) animal);
      dogAnimal.saySth();
   }
}
```

Jeśli chcemy potraktować obiekt klasy Animal jako np. Cat, to musimy go najpierw zrzutować. Jest to konsekwencja statycznego typowania w Java. To są przecież obiekty klasy Animal.

Polimorfizm a klasy abstrakcyjne

https://github.com/idzikpro/JavaCore/blob/master/src/main/java/pl/idzikpro/classes/AnimalMain.java

```
public class Animal {
   public void giveVoice() {
      System.out.println("Animal voice");
   }
}
```

```
public class Cat extends Animal {

   @Override
   public void giveVoice() {
      System.out.println("Cat voice");
   }
}
```

```
Animal cat=new Cat();
cat.giveVoice();
```

cat.giveVoice() wywoła metodę z klasy **Cat** bo istnieje metoda przesłonięta. Gdyby w Cat jej nie było, to wywołałby metodę z Animal. Jednak gdyby była metoda tylko w Cat to byłby błąd bo obiekt cat jest obiektem klasy Animal (statyczne typowanie).

Jak to poprawić (**ale bez interfejsów**)?

Napisać w klasie Animal metodę abstract dajGlos(), to i wtedy klasa musi być też abstract. Nie możemy teraz stworzyć obiektu klasy Animal. Teraz widać, że nie robimy sztucznej metody. Tzn. nie robimy ciała metody i wtedy wszystko gra bez rzutowania czyli obiekt cat dalej będzie robił Miau

```
public abstract class Animal {
   public abstract void giveVoice();
}
```

Dalej przykład (**tym razem z interfejsami**)

https://github.com/idzikpro/JavaCore/blob/master/src/main/java/pl/idzikpro/classes/Voice.java

```
public interface Voice {
   public void saySth();
}
```

```
public class Cat implements Voice {

   @Override
   public void saySth() {
      System.out.println("Miau");
   }
}
```

```
List<Voice> voiceList = Arrays.asList(
      new Cat(),
      new Dog()
);

for (Voice animal : voiceList
) {
   animal.saySth();
}
```

Tutaj też obiekt cat “będzie robił miau”.

Hermetyzacja

Hermetyzacja to ukrywanie implementacji kodu (tzw. enkapsulacja) - użytkownik nie ma dostępu do wszystkich składowych klasy, jedynie do udostępnionych przez nas składowych i metod.

Klasy powinny ukrywać swoją wewnętrzną strukturę. Dzięki temu będzie można zmienić ciało metody bez informowania o tym innych klas z nich korzystających. Ukrywanie implementacji osiągamy przez odpowiednie modyfikator dostępu - **private**. Najlepiej będzie oznaczyć pola składowe jako **private** i zrealizowac dostęp do nich przez tzw **gettery** i **settery**.

Modyfikatory dostępu

Modyfikatory dostępu decydują o tym jak zmienna, metoda czy klasa jest widoczna i gdzie.

	Class	**Package**	**SubClass**	**World**
Public	Y	Y	Y	Y
Protected	Y	Y	Y	N
no-modifer	Y	Y	N	N
Private	Y	N	N	N

Przesłonięcie metody

Adnotacja **@override** przed nazwą metody informuje o tym, że metoda jest przesłonięta. Przesłonięcie metody to implementacja na nowo metody odziedziczonej po klasie bazowej (parametry i typ ten sam)

Przesłanianie metod – zasady

1) Modyfikator dostępu nie może być bardziej restrykcyjny
2) Argumenty są takie same
3) Zwracany typ jest ten sam albo jest podtypem typu zwracanego
4) Zbiór wyjątków metody przesłaniającej musi być podzbiorem wyjątków deklarowanych przez metodę przesłanianą

Słowo kluczowe abstract

Zadeklarowanie klasy jako **abstract** skutkuje tym, że nie możemy utworzyć obiektu takiej klasy. Takie klasy tworzy się, aby je później rozszerzyć przez inną klasę. **Metody abstrakcyjne nie posiadają implementacji.** Mają tylko modyfikator abstract. Klasa abstrakcyjna nie musi zawierać metod abstrakcyjnych, ale jeśli jakaś jest abstrakcyjna to automatycznie cała klasa staje się abstrakcyjna i musi być zadeklarowana jako abstract. **W klasie abstrakcyjnej może deklarować metody z ciałami. Metody abstrakcyjnej nie można przesłaniać**.