Faculty of Electronics and Information Technology
WARSAW UNIVERSITY OF TECHNOLOGY

Programowanie Aplikacyjne (PAP)

Java - programowanie obiektowe

Julian Myrcha
Institute of Computer Science
26.02.2025

inheritance -dziedziczenie klas

overriding -nadpisywanie metod, wszystkie metody wirtualne

polymorphism -polimorfizm realizowany przez metody wirtualne

abstraction -abstrakcja klasy abstrakcyjne, interfejsy

- Deklaracja klasy publicznej w `SimpleClass` w pliku SimpleClass.java

```
1  package pap;
2  public class SimpleClass {
3      public void fun1() {System.out.println("SimpleClass.fun1");}
4      public void fun2() {System.out.println("SimpleClass.fun2");}
5      public static void main(String[] args) {
6          SimpleClass base = new SimpleClass();
7          base.fun1();             // SampleClass.fun1
8          base.fun2();             // SampleClass.fun2
9          SimpleClass derived = new DerivedClass();
10         derived.fun1();           // SampleClass.fun1
11         derived.fun2();           // DerivedClass.fun2
12     }
13 }
14 class DerivedClass extends SimpleClass {
15     public void fun2() {System.out.println("DerivedClass.fun2");}
16 }
```

- Deklaracja klasy publicznej w `SimpleClass` w pliku SimpleClass.java
- Wszystkie pozostałe klasy muszą być prywatne(`DerivedClass`)

```
1  package pap;
2  public class SimpleClass {
3      public void fun1() {System.out.println("SimpleClass.fun1");}
4      public void fun2() {System.out.println("SimpleClass.fun2");}
5      public static void main(String[] args) {
6          SimpleClass base = new SimpleClass();
7          base.fun1();             // SampleClass.fun1
8          base.fun2();             // SampleClass.fun2
9          SimpleClass derived = new DerivedClass();
10         derived.fun1();           // SampleClass.fun1
11         derived.fun2();           // DerivedClass.fun2
12     }
13 }
14 class DerivedClass extends SimpleClass {
15     public void fun2() {System.out.println("DerivedClass.fun2");}
16 }
```

- Deklaracja klasy publicznej w `SimpleClass` w pliku SimpleClass.java
- Wszystkie pozostałe klasy muszą być prywatne(`DerivedClass`)
- Jednokrotne dziedziczenie za pomocą słowa kluczowego `extends`

```
1  package pap;
2  public class SimpleClass {
3      public void fun1() {System.out.println("SimpleClass.fun1");}
4      public void fun2() {System.out.println("SimpleClass.fun2");}
5      public static void main(String[] args) {
6          SimpleClass base = new SimpleClass();
7          base.fun1();             // SampleClass.fun1
8          base.fun2();             // SampleClass.fun2
9          SimpleClass derived = new DerivedClass();
10         derived.fun1();           // SampleClass.fun1
11         derived.fun2();           // DerivedClass.fun2
12     }
13 }
14 class DerivedClass extends SimpleClass {
15     public void fun2() {System.out.println("DerivedClass.fun2");}
16 }
```

- Deklaracja klasy publicznej w `SimpleClass` w pliku SimpleClass.java
- Wszystkie pozostałe klasy muszą być prywatne(`DerivedClass`)
- Jednokrotne dziedziczenie za pomocą słowa kluczowego `extends`
- Identyczna lista argumentów

```
1  package pap;
2  public class SimpleClass {
3      public void fun1() {System.out.println("SimpleClass.fun1");}
4      public void fun2() {System.out.println("SimpleClass.fun2");}
5      public static void main(String[] args) {
6          SimpleClass base = new SimpleClass();
7          base.fun1();            // SampleClass.fun1
8          base.fun2();            // SampleClass.fun2
9          SimpleClass derived = new DerivedClass();
10         derived.fun1();          // SampleClass.fun1
11         derived.fun2();          // DerivedClass.fun2
12     }
13 }
14 class DerivedClass extends SimpleClass {
15     public void fun2() {System.out.println("DerivedClass.fun2");}
16 }
```

- Deklaracja klasy publicznej w `SimpleClass` w pliku SimpleClass.java
- Wszystkie pozostałe klasy muszą być prywatne(`DerivedClass`)
- Jednokrotne dziedziczenie za pomocą słowa kluczowego `extends`
- Identyczna lista argumentów
- typ zwracany identyczny lub pochodny od typu metody nadpisywanej

- Deklaracja klasy publicznej w `SimpleClass` w pliku SimpleClass.java
- Wszystkie pozostałe klasy muszą być prywatne(`DerivedClass`)
- Jednokrotne dziedziczenie za pomocą słowa kluczowego `extends`
- Identyczna lista argumentów
- typ zwracany identyczny lub pochodny od typu metody nadpisywanej
- zakres dostępu nie może być bardziej zawężony niż w klasie bazowej

- Deklaracja klasy publicznej w `SimpleClass` w pliku SimpleClass.java
- Wszystkie pozostałe klasy muszą być prywatne(`DerivedClass`)
- Jednokrotne dziedziczenie za pomocą słowa kluczowego `extends`
- Identyczna lista argumentów
- typ zwracany identyczny lub pochodny od typu metody nadpisywanej
- zakres dostępu nie może być bardziej zawężony niż w klasie bazowej
- metoda `final` nie może być nadpisana

- Deklaracja klasy publicznej w `SimpleClass` w pliku SimpleClass.java
- Wszystkie pozostałe klasy muszą być prywatne(`DerivedClass`)
- Jednokrotne dziedziczenie za pomocą słowa kluczowego `extends`
- Identyczna lista argumentów
- typ zwracany identyczny lub pochodny od typu metody nadpisywanej
- zakres dostępu nie może być bardziej zawężony niż w klasie bazowej
- metoda `final` nie może być nadpisana
- metoda `static` nie może być nadpisana, ale może być zadaklarowana powtórnie

- Deklaracja klasy publicznej w `SimpleClass` w pliku SimpleClass.java
- Wszystkie pozostałe klasy muszą być prywatne(`DerivedClass`)
- Jednokrotne dziedziczenie za pomocą słowa kluczowego `extends`
- Identyczna lista argumentów
- typ zwracany identyczny lub pochodny od typu metody nadpisywanej
- zakres dostępu nie może być bardziej zawężony niż w klasie bazowej
- metoda `final` nie może być nadpisana
- metoda `static` nie może być nadpisana, ale może być zadaklarowana powtórnie
- w tym samym pakiecie nie możemy nadpisać metody która jest `private` lub `final`

- Deklaracja klasy publicznej w `SimpleClass` w pliku SimpleClass.java
- Wszystkie pozostałe klasy muszą być prywatne(`DerivedClass`)
- Jednokrotne dziedziczenie za pomocą słowa kluczowego `extends`
- Identyczna lista argumentów
- typ zwracany identyczny lub pochodny od typu metody nadpisywanej
- zakres dostępu nie może być bardziej zawężony niż w klasie bazowej
- metoda `final` nie może być nadpisana
- metoda `static` nie może być nadpisana, ale może być zadaklarowana powtórnie
- w tym samym pakiecie nie możemy nadpisać metody która jest `private` lub `final`
- w różnych pakietach możemy nadpisać metody `public` lub `protected`

- Deklaracja klasy publicznej w `SimpleClass` w pliku SimpleClass.java
- Wszystkie pozostałe klasy muszą być prywatne(`DerivedClass`)
- Jednokrotne dziedziczenie za pomocą słowa kluczowego `extends`
- Identyczna lista argumentów
- typ zwracany identyczny lub pochodny od typu metody nadpisywanej
- zakres dostępu nie może być bardziej zawężony niż w klasie bazowej
- metoda `final` nie może być nadpisana
- metoda `static` nie może być nadpisana, ale może być zadaklarowana powtórnie
- w tym samym pakiecie nie możemy nadpisać metody która jest `private` lub `final`
- w różnych pakietach możemy nadpisać metody `public` lub `protected`
- Konstruktorów nie możemy nadpisać

- każda deklaracja pola lub metody ma deklarator dostępu (domyślnie `private`)

```
1  package pap;
2  public class SimpleClass {
3    String s;                      // nie ma public, prywatne, inicjalizowane na pusty string
4    public SimpleClass(String s) {  // konstruktor
5        this.s = s;                 // this potrzebne, bo samo s to parametr
6    }
7    public String fun1(String s) {   return "Simple.fun1: " + this.s;  }
8    public String fun2(String s) {   return "Simple.fun2: " + s ;      }
9    public static void main(String[] args) {
10         // SimpleClass simple = new SimpleClass();        // nie mamy kostruktora domyslnego
11         SimpleClass simple = new SimpleClass("balbinka");
12         System.out.println(simple.fun1("main"));          // Simple.fun1: balbinka
13         System.out.println(simple.fun2("main"));          // Simple.fun2: main
14   }
15 }
```

- każda deklaracja pola lub metody ma deklarator dostępu (domyślnie `private`)
- dostęp do pól i metod obiektu za pomocą słowa kluczowego `this` z kropką

```
1  package pap;
2  public class SimpleClass {
3    String s;                      // nie ma public, prywatne, inicjalizowane na pusty string
4    public SimpleClass(String s) {  // konstruktor
5        this.s = s;                 // this potrzebne, bo samo s to parametr
6    }
7    public String fun1(String s) {   return "Simple.fun1: " + this.s;  }
8    public String fun2(String s) {   return "Simple.fun2: " + s ;      }
9    public static void main(String[] args) {
10         // SimpleClass simple = new SimpleClass();        // nie mamy kostruktora domyslnego
11         SimpleClass simple = new SimpleClass("balbinka");
12         System.out.println(simple.fun1("main"));          // Simple.fun1: balbinka
13         System.out.println(simple.fun2("main"));          // Simple.fun2: main
14   }
15 }
```

- każda deklaracja pola lub metody ma deklarator dostępu (domyślnie `private`)
- dostęp do pól i metod obiektu za pomocą słowa kluczowego `this` z kropką
- w ciele metody możemy pominąć `this` z kropką jeżeli nie mamy konfliktu z nazwami zmiennych lokalnych lub parametrów

```
1  package pap;
2  public class SimpleClass {
3    String s;                      // nie ma public, prywatne, inicjalizowane na pusty string
4    public SimpleClass(String s) {  // konstruktor
5        this.s = s;                // this potrzebne, bo samo s to parametr
6    }
7    public String fun1(String s) {   return "Simple.fun1: " + this.s;  }
8    public String fun2(String s) {   return "Simple.fun2: " + s ;      }
9    public static void main(String[] args) {
10         // SimpleClass simple = new SimpleClass();        // nie mamy kostruktora domyslnego
11         SimpleClass simple = new SimpleClass("balbinka");
12         System.out.println(simple.fun1("main"));          // Simple.fun1: balbinka
13         System.out.println(simple.fun2("main"));          // Simple.fun2: main
14   }
15 }
```

- aby klasa potomna miała dostęp do pola lub metody musi być co najmniej `protected`

```
1  package pap;
2  public class Simple {
3    protected String s;
4    public Simple(String s)      { this.s = s;  }
5    public String fun1(String s) { return "Simple "+this.s + " "+s; }
6    public static void main(String[] args) {
7    Simple simple = new Derived("balbinka", "gucio");
8    System.out.println(simple.fun1("main")); // Derived gucio main balbinka
9    }
10 }
11 class Derived extends Simple {
12   String s;
13   public Derived(String s1, String s2) {
14     super(s1);
15     this.s = s2;
16   }
17   public String fun1(String s){return "Derived "+this.s+" "+s+" "+super.s;}
18 }
```

- aby klasa potomna miała dostęp do pola lub metody musi być co najmniej `protected`
- jeżeli są konflikty nazw z aktualną klasą używamy słowa `super` z kropką

```
1  package pap;
2  public class Simple {
3    protected String s;
4    public Simple(String s)      { this.s = s;  }
5    public String fun1(String s) { return "Simple "+this.s + " "+s; }
6    public static void main(String[] args) {
7    Simple simple = new Derived("balbinka", "gucio");
8    System.out.println(simple.fun1("main")); // Derived gucio main balbinka
9    }
10 }
11 class Derived extends Simple {
12   String s;
13   public Derived(String s1, String s2) {
14     super(s1);
15     this.s = s2;
16   }
17   public String fun1(String s){return "Derived "+this.s+" "+s+" "+super.s;}
18 }
```

- aby klasa potomna miała dostęp do pola lub metody musi być co najmniej `protected`
- jeżeli są konflikty nazw z aktualną klasą używamy słowa `super` z kropką
- tego samego słowa używamy gdy wołamy konstruktor klasy bazowej

```
1  package pap;
2  public class Simple {
3    protected String s;
4    public Simple(String s)      { this.s = s;  }
5    public String fun1(String s) { return "Simple "+this.s + " "+s; }
6    public static void main(String[] args) {
7    Simple simple = new Derived("balbinka", "gucio");
8    System.out.println(simple.fun1("main")); // Derived gucio main balbinka
9    }
10 }
11 class Derived extends Simple {
12   String s;
13   public Derived(String s1, String s2) {
14     super(s1);
15     this.s = s2;
16   }
17   public String fun1(String s){return "Derived "+this.s+" "+s+" "+super.s;}
18 }
```

- pole statyczne jest wspólne dla wszystkich instancji klasy

```
1  package pap;
2  class World {
3    static int id = 123;
4    static {
5      System.out.println(log("in static"));            // ins tattic: 123
6    }
7    public static String log(String s) { return String.format("%s: %d", s, id);}
8    public String msg() {return log("in instance");}
9  }
10 public class Hello {
11   public static void main(String[] args) throws Exception {
12     System.out.println(World.log("called in main")); // called in main: 123
13     System.out.println((new World()).msg());         // in instance: 123
14   }
15 }
```

- pole statyczne jest wspólne dla wszystkich instancji klasy
- metoda statyczna może być zawołana na rzacz obiektu, ale go nie widzi

```
1  package pap;
2  class World {
3    static int id = 123;
4    static {
5      System.out.println(log("in static"));            // ins tattic: 123
6    }
7    public static String log(String s) { return String.format("%s: %d", s, id);}
8    public String msg() {return log("in instance");}
9  }
10 public class Hello {
11   public static void main(String[] args) throws Exception {
12     System.out.println(World.log("called in main")); // called in main: 123
13     System.out.println((new World()).msg());         // in instance: 123
14   }
15 }
```

- pole statyczne jest wspólne dla wszystkich instancji klasy
- metoda statyczna może być zawołana na rzacz obiektu, ale go nie widzi
- metoda statyczna widzi pola statyczne

```
1  package pap;
2  class World {
3    static int id = 123;
4    static {
5      System.out.println(log("in static"));            // ins tattic: 123
6    }
7    public static String log(String s) { return String.format("%s: %d", s, id);}
8    public String msg() {return log("in instance");}
9  }
10 public class Hello {
11   public static void main(String[] args) throws Exception {
12     System.out.println(World.log("called in main")); // called in main: 123
13     System.out.println((new World()).msg());         // in instance: 123
14   }
15 }
```

- pole statyczne jest wspólne dla wszystkich instancji klasy
- metoda statyczna może być zawołana na rzacz obiektu, ale go nie widzi
- metoda statyczna widzi pola statyczne
- statyczne bloki (może być ich wiele) inicjalizacyjne wykonują się przy pierwszym wykorzystaniu obiektu

```
1  package pap;
2  class World {
3    static int id = 123;
4    static {
5      System.out.println(log("in static"));            // ins tattic: 123
6    }
7    public static String log(String s) { return String.format("%s: %d", s, id);}
8    public String msg() {return log("in instance");}
9  }
10 public class Hello {
11   public static void main(String[] args) throws Exception {
12     System.out.println(World.log("called in main")); // called in main: 123
13     System.out.println((new World()).msg());         // in instance: 123
14   }
15 }
```

- wykorzystanie metody statycznej do implementacji wzorca singleton

```
1  package pap;
2  final class Singleton {
3    private static final Singleton INSTANCE = new Singleton();
4    private Singleton() {
5      System.out.println("created!");
6    }
7    public static Singleton getInstance() {
8        return INSTANCE;
9    }
10   public static String staticMsg() {return "static msg";}
11   public String instanceMsg() {return "instance msg";}
12 }
13 public class SingletonTest {
14   public static void main(String[] args) {
15     System.out.println(Singleton.staticMsg());  // to juz powoduje utworzenie obiektu !
16     Singleton v1 = Singleton.getInstance();
17     Singleton v2 = Singleton.getInstance();
18     System.out.println(v1 == v2);               // true
19     System.out.println(v1.instanceMsg());       // instance msg
20   }
21 }
```

- wersja bezpieczna dla programowania wielowątkowego

```
1  package pap;
2  final class Singleton {
3    private static volatile Singleton instance = null;
4    private Singleton() {}
5    public static Singleton getInstance() {
6        if (instance == null) {
7            synchronized(Singleton.class) {
8                if (instance == null)
9                    instance = new Singleton();
10           }
11       }
12       return instance;
13   }
14   public String instanceMsg() {return "instance msg";}
15 }
16 public class SingletonTest {
17   public static void main(String[] args) {
18     Singleton v1 = Singleton.getInstance();
19     System.out.println(v1.instanceMsg());       // instance msg
20   }
21 }
```

- klasa abstrakcyjna może, ale nie musi posiadać metod abstrakcyjnych
  - ale jak ma co najmniej jedną taką metodę, musi być abstrakcyjna
- jak jest abstrakcyjna, to nie można utworzyć instancji
- można z niej dziedziczyć i tam zrobić implementację

```
1  public abstract class Sample {
2    protected String s;
3    public Sample(String s) {
4      this.s = s;
5    }
6    public abstract String fun1(String s) ;
7    public static void main(String[] args) {
8      Sample obj = new Derived("balbinka");
9      System.out.println(obj.fun1("main"));
10     // Derived balbinka main balbinka
11   }
12 }
13 class Derived extends Sample {
14   public Derived(String s1) { super(s1);}
15   public String fun1(String s){
16     return "Derived "+this.s+" "+s+" "+super.s;
17   }
18 }
```

- klasa abstrakcyjna może, ale nie musi posiadać metod abstrakcyjnych
  - ale jak ma co najmniej jedną taką metodę, musi być abstrakcyjna
- jak jest abstrakcyjna, to nie można utworzyć instancji
- można z niej dziedziczyć i tam zrobić implementację

```java
public abstract class Sample {
  protected String s;
  public Sample(String s) {
    this.s = s;
  }
  public abstract String fun1(String s) ;
  public static void main(String[] args) {
    Sample obj = new Derived("balbinka");
    System.out.println(obj.fun1("main"));
    // Derived balbinka main balbinka
  }
}
class Derived extends Sample {
  public Derived(String s1) { super(s1);}
  public String fun1(String s){
    return "Derived "+this.s+" "+s+" "+super.s;
  }
}
```

- klasa abstrakcyjna może, ale nie musi posiadać metod abstrakcyjnych
  - ale jak ma co najmniej jedną taką metodę, musi być abstrakcyjna
- jak jest abstrakcyjna, to nie można utworzyć instancji
- można z niej dziedziczyć i tam zrobić implementację

```java
public abstract class Sample {
  protected String s;
  public Sample(String s) {
    this.s = s;
  }
  public abstract String fun1(String s) ;
  public static void main(String[] args) {
    Sample obj = new Sample("balbinka"); // err
    System.out.println(obj.fun1("main"));
    // Derived balbinka main balbinka
  }
}
class Derived extends Sample {
  public Derived(String s1) { super(s1);}
  public String fun1(String s){
    return "Derived "+this.s+" "+s+" "+super.s;
  }
}
```

- klasa abstrakcyjna może, ale nie musi posiadać metod abstrakcyjnych
  - ale jak ma co najmniej jedną taką metodę, musi być abstrakcyjna
- jak jest abstrakcyjna, to nie można utworzyć instancji
- można z niej dziedziczyć i tam zrobić implementację

```
1  public abstract class Sample {
2    protected String s;
3    public Sample(String s) {
4      this.s = s;
5    }
6    public abstract String fun1(String s) ;
7    public static void main(String[] args) {
8      Sample obj = new Derived("balbinka");
9      System.out.println(obj.fun1("main"));
10     // Derived balbinka main balbinka
11   }
12 }
13 class Derived extends Sample {
14   public Derived(String s1) { super(s1);}
15   public String fun1(String s){
16     return "Derived "+this.s+" "+s+" "+super.s;
17   }
18 }
```

All the members (methods and fields) of an interface are public.
All the methods in an interface are public and abstract (except static and default).
All the fields of an interface are public, static and, final by default.

- Interfejs nie jest klasą, ale zestawem wymagań, czyli zestawem funkcji, które klasa musi implementować:
- Interfejs nie zawiera implementacji (*), podobnie jak klasy abstrakcyjne:
- Ale w Javie mamy dziedziczenie po dokładnie jednej klasie bazowej (domyślnie `object`)
  - obiekt możemy przypisać do dowolnego jego interfejsu

```
1 public interface IPrintable {
2     void print(String msg);
3 }
```

All the members (methods and fields) of an interface are public.
All the methods in an interface are public and abstract (except static and default).
All the fields of an interface are public, static and, final by default.

- Interfejs nie jest klasą, ale zestawem wymagań, czyli zestawem funkcji, które klasa musi implementować:
- Interfejs nie zawiera implementacji (*), podobnie jak klasy abstrakcyjne:
- Ale w Javie mamy dziedziczenie po dokładnie jednej klasie bazowej (domyślnie `object`)
  - obiekt możemy przypisać do dowolnego jego interfejsu

```java
public interface IPrintable {
    void print(String msg);
}
```

```java
abstract class Printable {
    private String name;
    public Printable(String n){
        name = n;
    }
    public abstract void print(String msg);
    public String getName(){
        return name;
    }
}
```

All the members (methods and fields) of an interface are public.
All the methods in an interface are public and abstract (except static and default).
All the fields of an interface are public, static and, final by default.

- Interfejs nie jest klasą, ale zestawem wymagań, czyli zestawem funkcji, które klasa musi implementować:
- Interfejs nie zawiera implementacji (*), podobnie jak klasy abstrakcyjne:
- Ale w Javie mamy dziedziczenie po dokładnie jednej klasie bazowej (domyślnie `object`)
  - obiekt możemy przypisać do dowolnego jego interfejsu

```java
public interface IPrintable {
    void print(String msg);
}
```

```java
abstract class Printable {
    private String name;
    public Printable(String n){
        name = n;
    }
    public abstract void print(String msg);
    public String getName(){
        return name;
    }
}
```

All the members (methods and fields) of an interface are public.
All the methods in an interface are public and abstract (except static and default).
All the fields of an interface are public, static and, final by default.

- Interfejs nie jest klasą, ale zestawem wymagań, czyli zestawem funkcji, które klasa musi implementować:
- Interfejs nie zawiera implementacji (*), podobnie jak klasy abstrakcyjne:
- Ale w Javie mamy dziedziczenie po dokładnie jednej klasie bazowej (domyślnie `object`)
  - obiekt możemy przypisać do dowolnego jego interfejsu

```
1  class PrintableExt extends Printable {
2    public PrintableExt(String type) {
3      super(type);
4    }
5    @Override
6    public void print(String msg) {
7      System.out.println(msg);
8    }
9  }
10
11 public class PrintableTest {
12   public static void main(String[] args) {
13     Printable p = new PrintableExt("DZM");
14     p.print("pap");
15   }
16 }
```

All the members (methods and fields) of an interface are public.
All the methods in an interface are public and abstract (except static and default).
All the fields of an interface are public, static and, final by default.

- Interfejs nie jest klasą, ale zestawem wymagań, czyli zestawem funkcji, które klasa musi implementować:
- Interfejs nie zawiera implementacji (*), podobnie jak klasy abstrakcyjne:
- Ale w Javie mamy dziedziczenie po dokładnie jednej klasie bazowej (domyślnie `object`)
  - obiekt możemy przypisać do dowolnego jego interfejsu

```
1  class Printable
2        implements IPrintable, IDescribable {
3    private String name;
4    public Printable(String type) {
5      name = type;
6    }
7    @Override
8    public void print(String msg) {
9      System.out.println(msg+ " " + name);
10   }
11   @Override
12   public void describe() {
13     System.out.println("describe: " + name);
14   }
15   public static void main(String[] args) {
16     IDescribable d = new PrintableImpl("DZM");
17     IPrintable p = (IPrintable)d;
18     p.print("pap");
19   }
20 }
```

- mogą być publiczne (wtedy są we własnym pliku) lub prywatne
- nie można utworzyć instancji interfejsu (bo to nie obiekt)
- klasa może implementować dowolną liczbę interfejsów
- wiele klas może implementować ten sam interfejs
- możemy to używać polimorficznie
- rzutowanie interfejsu na klasę powoduje błąd wykonania, gdy intefejs nie wskazuje obiektu danego typu

```
1  interface IWalk {
2      public String walk();
3  }
4  interface ISwim {
5      public String swim();
6  }
7  class Dog implements IWalk, ISwim {
8      public String walk() { return "Dog can walk";}
9      public String swim() { return "Dog can swim";}
10 }
11 public class Simple  {
12   public static void main(String[] args) {
13     Dog dog = new Dog();
14     System.out.println(dog.walk());
15     System.out.println(dog.swim());
16   }
17 }
```

- mogą być publiczne (wtedy są we własnym pliku) lub prywatne
- nie można utworzyć instancji interfejsu (bo to nie obiekt)
- klasa może implementować dowolną liczbę interfejsów
- wiele klas może implementować ten sam interfejs
- możemy to używać polimorficznie
- rzutowanie interfejsu na klasę powoduje błąd wykonania, gdy intefejs nie wskazuje obiektu danego typu

```
interface IWalk {
    public String walk();
}
interface ISwim {
    public String swim();
}
class Dog implements IWalk, ISwim {
    public String walk() { return "Dog can walk";}
    public String swim() { return "Dog can swim";}
}
public class Simple  {
  public static void main(String[] args) {
    Dog dog = new Dog();
    System.out.println(dog.walk());
    System.out.println(dog.swim());
  }
}
```

- mogą być publiczne (wtedy są we własnym pliku) lub prywatne
- nie można utworzyć instancji interfejsu (bo to nie obiekt)
- klasa może implementować dowolną liczbę interfejsów
- wiele klas może implementować ten sam interfejs
- możemy to używać polimorficznie
- rzutowanie interfejsu na klasę powoduje błąd wykonania, gdy intefejs nie wskazuje obiektu danego typu

```
1  interface IWalk {
2      public String walk();
3  }
4  interface ISwim {
5      public String swim();
6  }
7  class Dog implements IWalk, ISwim {
8      public String walk() { return "Dog can walk";}
9      public String swim() { return "Dog can swim";}
10 }
11 public class Simple  {
12   public static void main(String[] args) {
13     Dog dog = new Dog();
14     System.out.println(dog.walk());
15     System.out.println(dog.swim());
16   }
17 }
```

- mogą być publiczne (wtedy są we własnym pliku) lub prywatne
- nie można utworzyć instancji interfejsu (bo to nie obiekt)
- klasa może implementować dowolną liczbę interfejsów
- wiele klas może implementować ten sam interfejs
- możemy to używać polimorficznie
- rzutowanie interfejsu na klasę powoduje błąd wykonania, gdy intefejs nie wskazuje obiektu danego typu

```
1  class Dog implements IWalk, ISwim {
2      public String walk() { return "Dog can walk";}
3      public String swim() { return "Dog can swim";}
4  }
5  class Fish implements ISwim {
6      public String swim() { return "Fish can swim";}
7  }
8  public class Sample  {
9    public static void main(String[] args) {
10     ISwim[] animals = {new Dog(), new Fish()};
11     for(var animal:animals)
12         animal.swim();
13   }
14 }
```

- mogą być publiczne (wtedy są we własnym pliku) lub prywatne
- nie można utworzyć instancji interfejsu (bo to nie obiekt)
- klasa może implementować dowolną liczbę interfejsów
- wiele klas może implementować ten sam interfejs
- możemy to używać polimorficznie
- rzutowanie interfejsu na klasę powoduje błąd wykonania, gdy intefejs nie wskazuje obiektu danego typu

```
1  class Dog implements IWalk, ISwim {
2      public String walk() { return "Dog can walk";}
3      public String swim() { return "Dog can swim";}
4  }
5  class Fish implements ISwim {
6      public String swim() { return "Fish can swim";}
7  }
8  public class Sample  {
9    public static void main(String[] args) {
10     ISwim[] animals = {new Dog(), new Fish()};
11     for(var animal:animals)
12         animal.swim();
13   }
14 }
```

- mogą być publiczne (wtedy są we własnym pliku) lub prywatne
- nie można utworzyć instancji interfejsu (bo to nie obiekt)
- klasa może implementować dowolną liczbę interfejsów
- wiele klas może implementować ten sam interfejs
- możemy to używać polimorficznie
- rzutowanie interfejsu na klasę powoduje błąd wykonania, gdy intefejs nie wskazuje obiektu danego typu

```
1  class Dog implements IWalk, ISwim {
2      public String walk() { return "Dog can walk";}
3      public String swim() { return "Dog can swim";}
4  }
5  class Fish implements ISwim {
6      public String swim() { return "Fish can swim";}
7  }
8  public class Sample  {
9    public static void main(String[] args) {
10     ISwim[] animals = {new Dog(), new Fish()};
11     for(var animal:animals)
12         ((Dog)animal).walk();
13   }
14 }
```

- mogą być publiczne (wtedy są we własnym pliku) lub prywatne
- nie można utworzyć instancji interfejsu (bo to nie obiekt)
- klasa może implementować dowolną liczbę interfejsów
- wiele klas może implementować ten sam interfejs
- możemy to używać polimorficznie
- rzutowanie interfejsu na klasę powoduje błąd wykonania, gdy intefejs nie wskazuje obiektu danego typu

```
1  class Dog implements IWalk, ISwim {
2      public String walk() { return "Dog can walk";}
3      public String swim() { return "Dog can swim";}
4  }
5  class Fish implements ISwim {
6      public String swim() { return "Fish can swim";}
7  }
8  public class Sample  {
9    public static void main(String[] args) {
10     ISwim[] animals = {new Dog(), new Fish()};
11     for(var animal:animals)
12         ((IWalk)animal).walk();
13   }
14 }
```

- jest możliwość podania domyślnej implementacji metody w interfejsie
  - nie ma obowiązku implementacji nowej metody
  - ale z czasem taką implementację można dodać
- dodano to w celu uniknięcia modyfikacji istniejącego kodu przy zmianie interfejsu
- upodabnia to interfejsy do klas abstracyjnych
- ale klasy abstrakcyjne mają konstruktory, interfejsy ich nie mają

```
1  interface IAnimal {
2    default public String name(){ return "Animal";}
3  }
4  interface IWalk extends IAnimal {
5    public String walk();
6  }
7  interface ISwim extends IAnimal {
8    public String swim();
9  }
10 class Dog implements IWalk, ISwim {
11
12   public String walk() { return "Dog can walk";}
13   public String swim() { return "Dog can swim";}
14 }
```

- jest możliwość podania domyślnej implementacji metody w interfejsie
  - nie ma obowiązku implementacji nowej metody
  - ale z czasem taką implementację można dodać
- dodano to w celu uniknięcia modyfikacji istniejącego kodu przy zmianie interfejsu
- upodabnia to interfejsy do klas abstracyjnych
- ale klasy abstrakcyjne mają konstruktory, interfejsy ich nie mają

```
1  interface IAnimal {
2    default public String name(){ return "Animal";}
3  }
4  interface IWalk extends IAnimal {
5    public String walk();
6  }
7  interface ISwim extends IAnimal {
8    public String swim();
9  }
10 class Dog implements IWalk, ISwim {
11
12   public String walk() { return "Dog can walk";}
13   public String swim() { return "Dog can swim";}
14 }
```

- jest możliwość podania domyślnej implementacji metody w interfejsie
  - nie ma obowiązku implementacji nowej metody
  - ale z czasem taką implementację można dodać
- dodano to w celu uniknięcia modyfikacji istniejącego kodu przy zmianie interfejsu
- upodabnia to interfejsy do klas abstracyjnych
- ale klasy abstrakcyjne mają konstruktory, interfejsy ich nie mają

```
1  interface IAnimal {
2    default public String name(){ return "Animal";}
3  }
4  interface IWalk extends IAnimal {
5    public String walk();
6  }
7  interface ISwim extends IAnimal {
8    public String swim();
9  }
10 class Dog implements IWalk, ISwim {
11   public String name() { return "Dog";}
12   public String walk() { return "Dog can walk";}
13   public String swim() { return "Dog can swim";}
14 }
```

- jest możliwość podania domyślnej implementacji metody w interfejsie
  - nie ma obowiązku implementacji nowej metody
  - ale z czasem taką implementację można dodać
- dodano to w celu uniknięcia modyfikacji istniejącego kodu przy zmianie interfejsu
- upodabnia to interfejsy do klas abstracyjnych
- ale klasy abstrakcyjne mają konstruktory, interfejsy ich nie mają

```
1  interface IAnimal {
2    default public String name(){ return "Animal";}
3  }
4  interface IWalk extends IAnimal {
5    public String walk();
6  }
7  interface ISwim extends IAnimal {
8    public String swim();
9  }
10 class Dog implements IWalk, ISwim {
11   public String name() { return "Dog";}
12   public String walk() { return "Dog can walk";}
13   public String swim() { return "Dog can swim";}
14 }
```

- jest możliwość podania domyślnej implementacji metody w interfejsie
  - nie ma obowiązku implementacji nowej metody
  - ale z czasem taką implementację można dodać
- dodano to w celu uniknięcia modyfikacji istniejącego kodu przy zmianie interfejsu
- upodabnia to interfejsy do klas abstracyjnych
- ale klasy abstrakcyjne mają konstruktory, interfejsy ich nie mają

```
1  interface IAnimal {
2    default public String name(){ return "Animal";}
3  }
4  interface IWalk extends IAnimal {
5    public String walk();
6  }
7  interface ISwim extends IAnimal {
8    public String swim();
9  }
10 class Dog implements IWalk, ISwim {
11   public String name() { return "Dog";}
12   public String walk() { return "Dog can walk";}
13   public String swim() { return "Dog can swim";}
14 }
```

- jest możliwość podania domyślnej implementacji metody w interfejsie
  - nie ma obowiązku implementacji nowej metody
  - ale z czasem taką implementację można dodać
- dodano to w celu uniknięcia modyfikacji istniejącego kodu przy zmianie interfejsu
- upodabnia to interfejsy do klas abstracyjnych
- ale klasy abstrakcyjne mają konstruktory, interfejsy ich nie mają

```
1  interface IAnimal {
2    default public String name(){ return "Animal";}
3  }
4  interface IWalk extends IAnimal {
5    public String walk();
6  }
7  interface ISwim extends IAnimal {
8    public String swim();
9  }
10 class Dog implements IWalk, ISwim {
11   public String name() { return "Dog";}
12   public String walk() { return "Dog can walk";}
13   public String swim() { return "Dog can swim";}
14 }
```

- metody interfejsu są domyślnie `abstract` i `public`
- atrybuty interfejsu są domyślnie `public`, `static` i `final`
- Interfejsy mogą nie mieć metod
  - Służą wtedy do grupowania klas, mogą być przecinające się grupy
- Do sprawdzenia przynależności do typu wykorzystujemy operator `instanceof`

```
interface IAnimal {
  int VALUE = 100;
  default public String name(){ return "Animal";}
}
interface IWalk extends IAnimal {
  public String walk();
}
interface ISwim extends IAnimal {
  public String swim();
}
class Dog implements IWalk, ISwim {
  public String name() { return "Dog";}
  public String walk() { return "Dog can walk";}
  public String swim() { return "Dog can swim";}
}
```

- metody interfejsu są domyślnie `abstract` i `public`
- atrybuty interfejsu są domyślnie `public`, `static` i `final`
- Interfejsy mogą nie mieć metod
  - Służą wtedy do grupowania klas, mogą być przecinające się grupy
- Do sprawdzenia przynależności do typu wykorzystujemy operator `instanceof`

```
1  interface IAnimal {
2    int VALUE = 100;
3    default public String name(){ return "Animal";}
4  }
5  interface IWalk extends IAnimal {
6    public String walk();
7  }
8  interface ISwim extends IAnimal {
9    public String swim();
10 }
11 class Dog implements IWalk, ISwim {
12   public String name() { return "Dog";}
13   public String walk() { return "Dog can walk";}
14   public String swim() { return "Dog can swim";}
15 }
```

- metody interfejsu są domyślnie `abstract` i `public`
- atrybuty interfejsu są domyślnie `public`, `static` i `final`
- Interfejsy mogą nie mieć metod
  - Służą wtedy do grupowania klas, mogą być przecinające się grupy
- Do sprawdzenia przynależności do typu wykorzystujemy operator `instanceof`

```
1  interface IAnimal {
2    int VALUE = 100;
3    default public String name(){ return "Animal";}
4  }
5  interface IWalk extends IAnimal {
6    public String walk();
7  }
8  interface ISwim extends IAnimal {
9    public String swim();
10 }
11 class Dog implements IWalk, ISwim {
12   public String name() { return "Dog";}
13   public String walk() { return "Dog can walk";}
14   public String swim() { return "Dog can swim";}
15 }
```

- metody interfejsu są domyślnie `abstract` i `public`
- atrybuty interfejsu są domyślnie `public`, `static` i `final`
- Interfejsy mogą nie mieć metod
  - Służą wtedy do grupowania klas, mogą być przecinające się grupy
- Do sprawdzenia przynależności do typu wykorzystujemy operator `instanceof`

```
interface IAnimal {
  int VALUE = 100;
  default public String name(){ return "Animal";}
}
interface IWalk extends IAnimal {
  public String walk();
}
interface ISwim extends IAnimal {
  public String swim();
}
class Dog implements IWalk, ISwim {
  public String name() { return "Dog";}
  public String walk() { return "Dog can walk";}
  public String swim() { return "Dog can swim";}
}
```

- metody interfejsu są domyślnie `abstract` i `public`
- atrybuty interfejsu są domyślnie `public`, `static` i `final`
- Interfejsy mogą nie mieć metod
  - Służą wtedy do grupowania klas, mogą być przecinające się grupy
- Do sprawdzenia przynależności do typu wykorzystujemy operator `instanceof`

```
interface IWalk {
    public String walk();
}
interface ISwim {
    public String swim();
}
public class Sample  {
  public static void main(String[] args) {
    ISwim[] animals = {new Dog(), new Fish()};
    for(var animal:animals)
        if(animal instanceof IWalk)
            ((IWalk)animal).walk();
  }
}
```

- Obiekt klasy wewnętrznej ma pełen dostęp do pól prywatnych obiektu klasy zewnętrznej
  - po nazwie
  - po pełnej nazwie
- Tworząc obiekt klasy wewnętrznej wewnątrz metody klasy zewnętrznej referencja automatycznie ustawiana na `this`
- Tworząc ten obiekt na zewnątrz ustawiamy ręcznie pisząc `obiekt zewnętrzny.new` zamiast `new`
- Klasę wewnętrzną możemy zadeklarować nie tylko na poziomie obiektu, ale także wewnątrz metody (LKW)

```
1  interface ICompute{int compute(int a,int b);}
2  public class LocaleNesting {
3    int result;
4    class Computer implements ICompute {
5      @Override
6      public int compute(int a, int b) {
7        return result = a+b;
8      }
9    }
10   public ICompute computer() {
11     return new Computer();
12   }
13   public static void main(String[] args) {
14     LocaleNesting ob = new LocaleNesting();
15     Computer cmp = ob.new Computer();
16     System.out.println(cmp.compute(10,20));
17   }
18 }
```

- Obiekt klasy wewnętrznej ma pełen dostęp do pól prywatnych obiektu klasy zewnętrznej
  - po nazwie
  - po pełnej nazwie
- Tworząc obiekt klasy wewnętrznej wewnątrz metody klasy zewnętrznej referencja automatycznie ustawiana na `this`
- Tworząc ten obiekt na zewnątrz ustawiamy ręcznie pisząc `obiekt zewnętrzny.new` zamiast `new`
- Klasę wewnętrzną możemy zadeklarować nie tylko na poziomie obiektu, ale także wewnątrz metody (LKW)

```
1  interface ICompute{int compute(int a,int b);}
2  public class LocaleNesting {
3    int result;
4    class Computer implements ICompute {
5      @Override
6      public int compute(int a, int b) {
7        return result = a+b;
8      }
9    }
10   public ICompute computer() {
11     return new Computer();
12   }
13   public static void main(String[] args) {
14     LocaleNesting ob = new LocaleNesting();
15     Computer cmp = ob.new Computer();
16     System.out.println(cmp.compute(10,20));
17   }
18 }
```

- Obiekt klasy wewnętrznej ma pełen dostęp do pól prywatnych obiektu klasy zewnętrznej
  - po nazwie
  - po pełnej nazwie
- Tworząc obiekt klasy wewnętrznej wewnątrz metody klasy zewnętrznej referencja automatycznie ustawiana na `this`
- Tworząc ten obiekt na zewnątrz ustawiamy ręcznie pisząc `obiekt zewnętrzny.new` zamiast `new`
- Klasę wewnętrzną możemy zadeklarować nie tylko na poziomie obiektu, ale także wewnątrz metody (LKW)

```
1  interface ICompute{int compute(int a,int b);}
2  public class LocaleNesting {
3    int result;
4    class Computer implements ICompute {
5      @Override
6      public int compute(int a, int b) {
7        return LocaleNesting.this.result=a+b;
8      }
9    }
10   public ICompute computer() {
11     return new Computer();
12   }
13   public static void main(String[] args) {
14     LocaleNesting ob = new LocaleNesting();
15     Computer cmp = ob.new Computer();
16     System.out.println(cmp.compute(10,20));
17   }
18 }
```

- Obiekt klasy wewnętrznej ma pełen dostęp do pól prywatnych obiektu klasy zewnętrznej
  - po nazwie
  - po pełnej nazwie
- Tworząc obiekt klasy wewnętrznej wewnątrz metody klasy zewnętrznej referencja automatycznie ustawiana na `this`
- Tworząc ten obiekt na zewnątrz ustawiamy ręcznie pisząc `obiekt zewnętrzny.new` zamiast `new`
- Klasę wewnętrzną możemy zadeklarować nie tylko na poziomie obiektu, ale także wewnątrz metody (LKW)

```
1  interface ICompute{int compute(int a,int b);}
2  public class LocaleNesting {
3    int result;
4    class Computer implements ICompute {
5      @Override
6      public int compute(int a, int b) {
7        return result = a+b;
8      }
9    }
10   public ICompute computer() {
11     return new Computer();
12   }
13   public static void main(String[] args) {
14     LocaleNesting ob = new LocaleNesting();
15     Computer cmp = ob.computer();
16     System.out.println(cmp.compute(10,20));
17   }
18 }
```

- Obiekt klasy wewnętrznej ma pełen dostęp do pól prywatnych obiektu klasy zewnętrznej
  - po nazwie
  - po pełnej nazwie
- Tworząc obiekt klasy wewnętrznej wewnątrz metody klasy zewnętrznej referencja automatycznie ustawiana na `this`
- Tworząc ten obiekt na zewnątrz ustawiamy ręcznie pisząc `obiekt zewnętrzny.new` zamiast `new`
- Klasę wewnętrzną możemy zadeklarować nie tylko na poziomie obiektu, ale także wewnątrz metody (LKW)

```
1  interface ICompute{int compute(int a,int b);}
2  public class LocaleNesting {
3    int result;
4    class Computer implements ICompute {
5      @Override
6      public int compute(int a, int b) {
7        return result = a+b;
8      }
9    }
10   public ICompute computer() {
11     return new Computer();
12   }
13   public static void main(String[] args) {
14     LocaleNesting ob = new LocaleNesting();
15     Computer cmp = ob.new Computer();
16     System.out.println(cmp.compute(10,20));
17   }
18 }
```

- Obiekt klasy wewnętrznej ma pełen dostęp do pól prywatnych obiektu klasy zewnętrznej
  - po nazwie
  - po pełnej nazwie
- Tworząc obiekt klasy wewnętrznej wewnątrz metody klasy zewnętrznej referencja automatycznie ustawiana na `this`
- Tworząc ten obiekt na zewnątrz ustawiamy ręcznie pisząc `obiekt zewnętrzny.new` zamiast `new`
- Klasę wewnętrzną możemy zadeklarować nie tylko na poziomie obiektu, ale także wewnątrz metody (LKW)

```
1  interface ICompute{int compute(int a,int b);}
2  public class LocaleNesting {
3    int result;
4    public ICompute computer() {
5      class Computer implements ICompute {
6          @Override
7          public int compute(int a, int b) {
8            return result = a+b;
9          }
10     }
11     return new Computer();
12   }
13   public static void main(String[] args) {
14     LocaleNesting ob = new LocaleNesting();
15     ICompute cmp = ob.computer();
16     System.out.println(cmp.compute(10,20));
17   }
18 }
```

- Klasa wewnętrzna jest niedostępna poza metodą.
- Jeżeli chcemy zwrócić obiekt tego typu, to jedynym sposobem jest Interfejs lub klasa bazowa
- Gdy obiekt klasy wewnętrznnej tworzymy tylko raz to nie trzeba nawet nadawać jej nazwy
- Czyli alokator klasy bazowej lub interfejsu a potem definicja klasy anonimowej rzeczywiście tworzonego obiektu
- Lub Lambda
- Jako LKW ma dostęp do lokalnych, finalnych zmiennych metody
- także jeżeli kompilator sam sprawdzi że może dodać final ...
- ... ma dostęp do wszystkich pól obiektu zewnętrznego

```
1  interface ICompute { int compute(int a, int b); }
2  public class LocaleNesting {
3    public ICompute computer() {
4      class Computer implements ICompute {
5        @Override
6        public int compute(int a, int b) {
7          return a + b;
8        }
9      }
10     return new Computer();
11   }
12   public static void main(String[] args) {
13     LocaleNesting ob = new LocaleNesting();
14     ICompute i = ob.computer();
15     System.out.println(i.compute(10, 20));
16   }
17 }
```

- Klasa wewnętrzna jest niedostępna poza metodą.
- Jeżeli chcemy zwrócić obiekt tego typu, to jedynym sposobem jest Interfejs lub klasa bazowa
- Gdy obiekt klasy wewnętrznnej tworzymy tylko raz to nie trzeba nawet nadawać jej nazwy
- Czyli alokator klasy bazowej lub interfejsu a potem definicja klasy anonimowej rzeczywiście tworzonego obiektu
- Lub Lambda
- Jako LKW ma dostęp do lokalnych, finalnych zmiennych metody
- także jeżeli kompilator sam sprawdzi że może dodać final ...
- ... ma dostęp do wszystkich pól obiektu zewnętrznego

```
1  interface ICompute { int compute(int a, int b); }
2  public class LocaleNesting {
3    public ICompute computer() {
4      class Computer implements ICompute {
5        @Override
6        public int compute(int a, int b) {
7          return a + b;
8        }
9      }
10     return new Computer();
11   }
12   public static void main(String[] args) {
13     LocaleNesting ob = new LocaleNesting();
14     ICompute i = ob.computer();
15     System.out.println(i.compute(10, 20));
16   }
17 }
```

- Klasa wewnętrzna jest niedostępna poza metodą.
- Jeżeli chcemy zwrócić obiekt tego typu, to jedynym sposobem jest Interfejs lub klasa bazowa
- Gdy obiekt klasy wewnętrznnej tworzymy tylko raz to nie trzeba nawet nadawać jej nazwy
- Czyli alokator klasy bazowej lub interfejsu a potem definicja klasy anonimowej rzeczywiście tworzonego obiektu
- Lub Lambda
- Jako LKW ma dostęp do lokalnych, finalnych zmiennych metody
- także jeżeli kompilator sam sprawdzi że może dodać final ...
- ... ma dostęp do wszystkich pól obiektu zewnętrznego

```
1  interface ICompute {   int compute(int a, int b); }
2  public class LocaleNesting {
3    public ICompute computer() {
4      return new ICompute() {
5        @Override
6        public int compute(int a, int b) {
7          return a+b;
8        }
9      };
10   }
11
12   public static void main(String[] args) {
13     LocaleNesting ob = new LocaleNesting();
14     ICompute i = ob.computer();
15     System.out.println(i.compute(10, 20));
16   }
17 }
```

- Klasa wewnętrzna jest niedostępna poza metodą.
- Jeżeli chcemy zwrócić obiekt tego typu, to jedynym sposobem jest Interfejs lub klasa bazowa
- Gdy obiekt klasy wewnętrznnej tworzymy tylko raz to nie trzeba nawet nadawać jej nazwy
- Czyli alokator klasy bazowej lub interfejsu a potem definicja klasy anonimowej rzeczywiście tworzonego obiektu
- Lub Lambda
- Jako LKW ma dostęp do lokalnych, finalnych zmiennych metody
- także jeżeli kompilator sam sprawdzi że może dodać final ...
- ... ma dostęp do wszystkich pól obiektu zewnętrznego

```
1  interface ICompute {   int compute(int a, int b); }
2  public class LocaleNesting {
3    public ICompute computer() {
4      return new ICompute() {
5        @Override
6        public int compute(int a, int b) {
7          return a+b;
8        }
9      };
10   }
11
12   public static void main(String[] args) {
13     LocaleNesting ob = new LocaleNesting();
14     ICompute i = ob.computer();
15     System.out.println(i.compute(10, 20));
16   }
17 }
```

- Klasa wewnętrzna jest niedostępna poza metodą.
- Jeżeli chcemy zwrócić obiekt tego typu, to jedynym sposobem jest Interfejs lub klasa bazowa
- Gdy obiekt klasy wewnętrznnej tworzymy tylko raz to nie trzeba nawet nadawać jej nazwy
- Czyli alokator klasy bazowej lub interfejsu a potem definicja klasy anonimowej rzeczywiście tworzonego obiektu
- Lub Lambda
- Jako LKW ma dostęp do lokalnych, finalnych zmiennych metody
- także jeżeli kompilator sam sprawdzi że może dodać final ...
- ... ma dostęp do wszystkich pól obiektu zewnętrznego

```
1  interface ICompute {   int compute(int a, int b); }
2  public class LocaleNesting {
3    public ICompute computer() {
4      return (a,b) ->  a+b;
5    }
6
7
8
9
10
11
12   public static void main(String[] args) {
13     LocaleNesting ob = new LocaleNesting();
14     ICompute i = ob.computer();
15     System.out.println(i.compute(10, 20));
16   }
17 }
```

- Klasa wewnętrzna jest niedostępna poza metodą.
- Jeżeli chcemy zwrócić obiekt tego typu, to jedynym sposobem jest Interfejs lub klasa bazowa
- Gdy obiekt klasy wewnętrznnej tworzymy tylko raz to nie trzeba nawet nadawać jej nazwy
- Czyli alokator klasy bazowej lub interfejsu a potem definicja klasy anonimowej rzeczywiście tworzonego obiektu
- Lub Lambda
- Jako LKW ma dostęp do lokalnych, finalnych zmiennych metody
- także jeżeli kompilator sam sprawdzi że może dodać final ...
- ... ma dostęp do wszystkich pól obiektu zewnętrznego

```
1  interface ICompute {   int compute(int a, int b); }
2  public class LocaleNesting {
3
4
5
6
7
8
9
10
11
12   public static void main(String[] args) {
13
14     ICompute i = (a,b) ->  a+b;
15     System.out.println(i.compute(10, 20));
16   }
17 }
```

- Klasa wewnętrzna jest niedostępna poza metodą.
- Jeżeli chcemy zwrócić obiekt tego typu, to jedynym sposobem jest Interfejs lub klasa bazowa
- Gdy obiekt klasy wewnętrznnej tworzymy tylko raz to nie trzeba nawet nadawać jej nazwy
- Czyli alokator klasy bazowej lub interfejsu a potem definicja klasy anonimowej rzeczywiście tworzonego obiektu
- Lub Lambda
- Jako LKW ma dostęp do lokalnych, finalnych zmiennych metody
- także jeżeli kompilator sam sprawdzi że może dodać final ...
- ... ma dostęp do wszystkich pól obiektu zewnętrznego

```
1  interface ICompute {   int compute(int a, int b); }
2  public class LocaleNesting {
3
4
5
6
7
8
9
10
11
12   public static void main(String[] args) {
13     final int z = 10;
14     ICompute i = (a,b) ->  a+b+z;
15     System.out.println(i.compute(10, 20));
16   }
17 }
```

- Klasa wewnętrzna jest niedostępna poza metodą.
- Jeżeli chcemy zwrócić obiekt tego typu, to jedynym sposobem jest Interfejs lub klasa bazowa
- Gdy obiekt klasy wewnętrznnej tworzymy tylko raz to nie trzeba nawet nadawać jej nazwy
- Czyli alokator klasy bazowej lub interfejsu a potem definicja klasy anonimowej rzeczywiście tworzonego obiektu
- Lub Lambda
- Jako LKW ma dostęp do lokalnych, finalnych zmiennych metody
- także jeżeli kompilator sam sprawdzi że może dodać final ...
- ... ma dostęp do wszystkich pól obiektu zewnętrznego

```
1  interface ICompute {   int compute(int a, int b); }
2  public class LocaleNesting {
3
4
5
6
7
8
9
10
11
12   public static void main(String[] args) {
13     int z = 10;    // final dodane przez kompilator
14     ICompute i = (a,b) ->  a+b+z;
15     System.out.println(i.compute(10, 20));
16   }
17 }
```

- Klasa wewnętrzna jest niedostępna poza metodą.
- Jeżeli chcemy zwrócić obiekt tego typu, to jedynym sposobem jest Interfejs lub klasa bazowa
- Gdy obiekt klasy wewnętrznnej tworzymy tylko raz to nie trzeba nawet nadawać jej nazwy
- Czyli alokator klasy bazowej lub interfejsu a potem definicja klasy anonimowej rzeczywiście tworzonego obiektu
- Lub Lambda
- Jako LKW ma dostęp do lokalnych, finalnych zmiennych metody
- także jeżeli kompilator sam sprawdzi że może dodać final ...
- ... ma dostęp do wszystkich pól obiektu zewnętrznego

- Tworzymy obiekt klasy anonimowej implementujący interfejs `ActionListener`
- Tworzymy obiekt typu `Timer`, przekazując mu referencję do obiektu
- Metoda `start` obiektu `t` uruchamia cykliczne wywołania metody `actionPerformed`
- Mamy na to czas, bo metoda `sleep` czeka 10 sekund
- Metoda `stop` obiektu `t` zatrzymuje cykliczne wywołania metody `actionPerformed`

```
1  public class TimerTest {
2    public static void main(String[] args)
3                              throws InterruptedException {
4      Timer t = new Timer(1000, e->{
5        System.out.println("godzina: " + new Date());
6        Toolkit.getDefaultToolkit().beep();
7      });
8      t.start();
9      Thread.sleep(10000);
10     t.stop();
11     System.out.println("Koniec");
12   }
13 }
```

- rozwiązanie z pobieraniem źródła z eventu nie zawsze wystarcza
- zamiast tego stosujemy klasy wewnętrzne
- i deklarację kontrolek na poziomie klasy głównej:
- albo lambda z kontrolkami w klasie głównej
- lub nawet jako zmienne w metodzie

```java
class ButtonListener implements ActionListener {
  @Override
  public void actionPerformed(ActionEvent e) {
    ((JButton)e.getSource()).setText("nacisnieto");
  }
}
public class Sample {
  void run() {
    SwingUtilities.invokeLater(() -> {
      JFrame frame = new JFrame("Pierwszy Przycisk");
      frame.setBounds(100, 100, 450, 300);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      var btn = new JButton("jeszcze nie nacisnieto");
      btn.addActionListener(new ButtonListener());
      frame.getContentPane().add(btn);
      frame.setVisible(true);
    });
  }
  public static void main(String[] args) {
    new Sample().run();
  }
}
```

- rozwiązanie z pobieraniem źródła z eventu nie zawsze wystarcza
- zamiast tego stosujemy `klasy wewnętrzne`
- i deklarację kontrolek na poziomie klasy głównej:
- albo lambda z kontrolkami w klasie głównej
- lub nawet jako zmienne w metodzie

```
1  public class SimpleSwingSample {
2    JButton closeButton;
3    class ButtonListener implements ActionListener {
4      @Override
5      public void actionPerformed(ActionEvent e) {
6        closeButton.setText("nacisnieto");
7      }
8    }
9    void run() {
10     SwingUtilities.invokeLater(() -> {
11       ...
12       closeButton = new JButton("jeszcze nie nacisnieto");
13       closeButton.addActionListener(new ButtonListener());
14       ...
15     });
16   }
17   public static void main(String[] args) {
18     new SimpleSwingSample().run();
19   }
20 }
```

- rozwiązanie z pobieraniem źródła z eventu nie zawsze wystarcza
- zamiast tego stosujemy klasy wewnętrzne
- i deklarację kontrolek na poziomie klasy głównej:
- albo lambda z kontrolkami w klasie głównej
- lub nawet jako zmienne w metodzie

```
1  public class SimpleSwingSample {
2    JButton closeButton;
3    class ButtonListener implements ActionListener {
4      @Override
5      public void actionPerformed(ActionEvent e) {
6        closeButton.setText("nacisnieto");
7      }
8    }
9    void run() {
10     SwingUtilities.invokeLater(() -> {
11       ...
12       closeButton = new JButton("jeszcze nie nacisnieto");
13       closeButton.addActionListener(new ButtonListener());
14       ...
15     });
16   }
17   public static void main(String[] args) {
18     new SimpleSwingSample().run();
19   }
20 }
```

- rozwiązanie z pobieraniem źródła z eventu nie zawsze wystarcza
- zamiast tego stosujemy `klasy wewnętrzne`
- i deklarację kontrolek na poziomie klasy głównej:
- albo lambda z kontrolkami w klasie głównej
- lub nawet jako zmienne w metodzie

```
1  public class Sample {
2    JButton btn;
3    void run() {
4      SwingUtilities.invokeLater(() -> {
5        JFrame frame = new JFrame("Pierwszy Przycisk");
6        frame.setBounds(100, 100, 450, 300);
7        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
8        this->btn = new JButton("jeszcze nie nacisnieto");
9        btn.addActionListener(
10         e->btn.setText("nacisnieto")
11       );
12
13       frame.getContentPane().add(btn);
14       frame.setVisible(true);
15     });
16   }
17   public static void main(String[] args) {
18     new Sample().run();
19   }
20 }
```

- rozwiązanie z pobieraniem źródła z eventu nie zawsze wystarcza
- zamiast tego stosujemy klasy wewnętrzne
- i deklarację kontrolek na poziomie klasy głównej:
- albo lambda z kontrolkami w klasie głównej
- lub nawet jako zmienne w metodzie

```
1  public class Sample {
2
3      void run() {
4      SwingUtilities.invokeLater(() -> {
5        JFrame frame = new JFrame("Pierwszy Przycisk");
6        frame.setBounds(100, 100, 450, 300);
7        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
8        JButton btn = new JButton("jeszcze nie nacisnieto");
9        btn.addActionListener(
10         e->btn.setText("nacisnieto")
11       );
12
13       frame.getContentPane().add(btn);
14       frame.setVisible(true);
15     });
16   }
17   public static void main(String[] args) {
18     new Sample().run();
19   }
20 }
```

- służą unikaniu konfliktu nazw

- służą unikaniu konfliktu nazw
- nazwa pakietu musi pasować do struktury katalogów, w której znajduje się bytecode

- służą unikaniu konfliktu nazw
- nazwa pakietu musi pasować do struktury katalogów, w której znajduje się bytecode
- zmienna środowiskowa `CLASSPATH` służy do znajdowania brakujących klas

wyświetlanie:

```
c:\>set CLASSPATH                                # Windows
/home/jmy>echo $CLASSPATH                        # Linux
```

ustawianie:

```
c:\ set CLASSPATH=c:\users\jmy\java\classes       # Windows
/home/jmy>export $CLASSPATH=/home/jmy>java/classes  # Linux
```

- Często mamy taką samą funkcjonalność dla kilku typów
- Domyślnym rozwiązaniem jest użycie klasy Object
  - Ale to może być zbyt szerokie bo wszystko pasuje
  - konieczne jest rzutowanie i tracimy kontrolę (błąd wykonania a nie bład kompilacji)
- Użycie typów generycznych zawęża dostępne typy
  - kompilator sprawdza to w trakcie kompilacji

```
1 import java.util.List;
2 import java.util.LinkedList;
3 ...
4 List list = new LinkedList();
5 list.add(Integer.valueOf(0));
6 list.add("balbinka");
7 Integer x=(Integer)list.iterator().next();
```

- Często mamy taką samą funkcjonalność dla kilku typów
- Domyślnym rozwiązaniem jest użycie klasy Object
  - Ale to może być zbyt szerokie bo wszystko pasuje
  - konieczne jest rzutowanie i tracimy kontrolę (błąd wykonania a nie blad kompilacji)
- Użycie typów generycznych zawęża dostępne typy
  - kompilator sprawdza to w trakcie kompilacji

```
1 import java.util.List;
2 import java.util.LinkedList;
3 ...
4 List list = new LinkedList();
5 list.add(Integer.valueOf(0));
6 list.add("balbinka");
7 Integer x=(Integer)list.iterator().next();
```

- Często mamy taką samą funkcjonalność dla kilku typów
- Domyślnym rozwiązaniem jest użycie klasy Object
  - Ale to może być zbyt szerokie bo wszystko pasuje
  - konieczne jest rzutowanie i tracimy kontrolę (błąd wykonania a nie bład kompilacji)
- Użycie typów generycznych zawęża dostępne typy
  - kompilator sprawdza to w trakcie kompilacji

```
1 import java.util.List;
2 import java.util.LinkedList;
3 ...
4 List list = new LinkedList();
5 list.add(Integer.valueOf(0));
6 list.add("balbinka");
7 Integer x=(Integer)list.iterator().next();
```

- Często mamy taką samą funkcjonalność dla kilku typów
- Domyślnym rozwiązaniem jest użycie klasy Object
  - Ale to może być zbyt szerokie bo wszystko pasuje
  - konieczne jest rzutowanie i tracimy kontrolę (błąd wykonania a nie blad kompilacji)
- Użycie typów generycznych zawęża dostępne typy
  - kompilator sprawdza to w trakcie kompilacji

```
1 import java.util.List;
2 import java.util.LinkedList;
3 ...
4 List list = new LinkedList();
5 list.add(Integer.valueOf(0));
6 list.add("balbinka");
7 Integer x=(Integer)list.iterator().next();
```

- Często mamy taką samą funkcjonalność dla kilku typów
- Domyślnym rozwiązaniem jest użycie klasy Object
  - Ale to może być zbyt szerokie bo wszystko pasuje
  - konieczne jest rzutowanie i tracimy kontrolę (błąd wykonania a nie blad kompilacji)
- Użycie typów generycznych zawęża dostępne typy
  - kompilator sprawdza to w trakcie kompilacji

```
1 import java.util.List;
2 import java.util.LinkedList;
3 ...
4 List<Integer> list=new LinkedList<Integer>();
5 list.add(Integer.valueOf(0));
6 //list.add("balbinka"); blad kompilacji
7 Integer x=list.iterator().next();
```

- Często mamy taką samą funkcjonalność dla kilku typów
- Domyślnym rozwiązaniem jest użycie klasy Object
  - Ale to może być zbyt szerokie bo wszystko pasuje
  - konieczne jest rzutowanie i tracimy kontrolę (błąd wykonania a nie blad kompilacji)
- Użycie typów generycznych zawęża dostępne typy
  - kompilator sprawdza to w trakcie kompilacji

```
1 import java.util.List;
2 import java.util.LinkedList;
3 ...
4 List<Integer> list=new LinkedList<Integer>();
5 list.add(Integer.valueOf(0));
6 //list.add("balbinka"); blad kompilacji
7 Integer x=list.iterator().next();
```

- W Javie nie są tworzone kopie dla każdego typu

```
1 public interface List <E> {
2     void add(E x);
3     Iterator<E> iterator();
4 }
5
6 public interface Iterator<E> {
7     E next();
8     boolean hasNext();
9 }
```

- W Javie nie są tworzone kopie dla każdego typu
- Jest to inne podejście niż template w C++

```
1 public interface List <E> {
2     void add(E x);
3     Iterator<E> iterator();
4 }
5
6 public interface Iterator<E> {
7     E next();
8     boolean hasNext();
9 }
```

- W Javie nie są tworzone kopie dla każdego typu
- Jest to inne podejście niż template w C++
- Deklaracja typu generic jest kompilowana raz na zawsze i przekształcana w pojedynczy plik klasy, tak jak zwykła deklaracja klasy lub interfejsu.

```
1 public interface List <E> {
2     void add(E x);
3     Iterator<E> iterator();
4 }
5
6 public interface Iterator<E> {
7     E next();
8     boolean hasNext();
9 }
```

- W Javie nie są tworzone kopie dla każdego typu
- Jest to inne podejście niż template w C++
- Deklaracja typu generic jest kompilowana raz na zawsze i przekształcana w pojedynczy plik klasy, tak jak zwykła deklaracja klasy lub interfejsu.
- Parametry typu są analogiczne do zwykłych parametrów metod

```
1 public interface List <E> {
2     void add(E x);
3     Iterator<E> iterator();
4 }
5
6 public interface Iterator<E> {
7     E next();
8     boolean hasNext();
9 }
```

- W Javie nie są tworzone kopie dla każdego typu
- Jest to inne podejście niż template w C++
- Deklaracja typu generic jest kompilowana raz na zawsze i przekształcana w pojedynczy plik klasy, tak jak zwykła deklaracja klasy lub interfejsu.
- Parametry typu są analogiczne do zwykłych parametrów metod
- Gdy wywoływana jest deklaracja ogólna, rzeczywiste argumenty typu są zastępowane formalnymi parametrami typu.

```
1 public interface List <E> {
2     void add(E x);
3     Iterator<E> iterator();
4 }
5
6 public interface Iterator<E> {
7     E next();
8     boolean hasNext();
9 }
```

- W Javie nie są tworzone kopie dla każdego typu
- Jest to inne podejście niż template w C++
- Deklaracja typu generic jest kompilowana raz na zawsze i przekształcana w pojedynczy plik klasy, tak jak zwykła deklaracja klasy lub interfejsu.
- Parametry typu są analogiczne do zwykłych parametrów metod
- Gdy wywoływana jest deklaracja ogólna, rzeczywiste argumenty typu są zastępowane formalnymi parametrami typu.
- Parametry formalne powinny być pojedynczymi dużymi literami

```
1 public interface List <E> {
2     void add(E x);
3     Iterator<E> iterator();
4 }
5
6 public interface Iterator<E> {
7     E next();
8     boolean hasNext();
9 }
```

- Przypisanie typu bardziej szczegółowego do mniej szczegółowego nie jest możliwe, ponieważ pozwoliłoby na wstawianie nie-stringów do listy stringów
- przypisanie do kolekcji nietypowanej możliwe i łatwo o błąd

```
1 List<String> ls = new ArrayList<String>();
2 List<Object> lo = ls;
3 // compilation error (List<String>
4 // nie dziedziczy z List<Object>)
```

- Przypisanie typu bardziej szczegółowego do mniej szczegółowego nie jest możliwe, ponieważ pozwoliłoby na wstawianie nie-stringów do listy stringów
- przypisanie do kolekcji nietypowanej możliwe i łatwo o błąd

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls;
// compilation error (List<String>
// nie dziedziczy z List<Object>)
```

```
List<Integer> list=new LinkedList<Integer>();
Collection col = myIntList;
col.add("ala");
for (int i : myIntList) {
  // java.lang.ClassCastException:
  // class java.lang.String cannot
  //be cast to class java.lang.Integer
  System.out.println(i);
}
```

- metoda ogólna obsłuży typy generyczne

```
1  static void printCollection(Collection c) {
2    Iterator i = c.iterator();
3    for (var k = 0; k < c.size(); k++) {
4        System.out.println(i.next());
5    }
6  }
7  ...
8  myIntList.add(Integer.valueOf(0));
9  Integer x = myIntList.iterator().next();
10 printCollection(myIntList) ;
```

- metoda ogólna obsłuży typy generyczne
- metoda specyficzna obsłuży jeden typ

```
static void printCollection(Collection<Integer> c) {
  for (Object e : c) {      // tu moze byc Integer lub Object
      System.out.println(e);
  }
}
...
myIntList.add(Integer.valueOf(0));
Integer x = myIntList.iterator().next();
printCollection(myIntList) ;
```

- metoda ogólna obsłuży typy generyczne
- metoda specyficzna obsłuży jeden typ
- metoda z symbolem wieloznacznym ? (wildcard) obsłuży wszystko

```
static void printCollection(Collection<?> c) {
  for (Object e : c) {      // tu moze byc Integer lub Object
      System.out.println(e);
  }
}
...
myIntList.add(Integer.valueOf(0));
Integer x = myIntList.iterator().next();
printCollection(myIntList) ;
```

- metoda ogólna obsłuży typy generyczne
- metoda specyficzna obsłuży jeden typ
- metoda z symbolem wieloznacznym ? (wildcard) obsłuży wszystko
- ponieważ ten typ jest nieokreślony nie da się zawołać metod wymagających określonego typu - np. add

- jeżeli mamy dziedziczenie to nie chcemy zupełnej ogólności

```
1  public abstract class Shape {
2      public abstract void draw(Canvas c);
3  }
4  public class Circle extends Shape {
5      private int x, y, radius;
6      public void draw(Canvas c) {
7          ...
8      }
9  }
10 public class Rectangle extends Shape {
11     private int x, y, width, height;
12     public void draw(Canvas c) {
13         ...
14     }
15 }
```

- jeżeli mamy dziedziczenie to nie chcemy zupełnej ogólności
- tu musimy rzutować

```
1  public class Canvas {
2    public void draw(Shape s) {
3      s.draw(this);
4    }
5    public void drawAll(List<?> shapes){
6      for (Object s: shapes) {
7          ((Shape)s).draw(this);
8      }
9    }
10 }
```

- jeżeli mamy dziedziczenie to nie chcemy zupełnej ogólności
- tu musimy rzutować
- tu nie zawołamy dla kolekcji `ListCircle`

```
1  public class Canvas {
2    public void draw(Shape s) {
3      s.draw(this);
4    }
5    public void drawAll(List<Shape> shapes){
6      for (Shape s: shapes) {
7          s.draw(this);
8      }
9    }
10 }
```

- jeżeli mamy dziedziczenie to nie chcemy zupełnej ogólności
- tu musimy rzutować
- tu nie zawołamy dla kolekcji `ListCircle`
- tu pasują wszystkie kolekcje elementów dziedziczących z Shape

```
1  public class Canvas {
2    public void draw(Shape s) {
3      s.draw(this);
4    }
5    public void
6    drawAll(List<? extends Shape> shapes){
7      for (Shape s: shapes) {
8          s.draw(this);
9      }
10   }
11 }
```

- jeżeli mamy dziedziczenie to nie chcemy zupełnej ogólności
- tu musimy rzutować
- tu nie zawołamy dla kolekcji `ListCircle`
- tu pasują wszystkie kolekcje elementów dziedziczących z Shape
- ale ponieważ rzeczywisty typ nieznany w casie kompilacji metody biorące parametr nie działają

```
1  public class Canvas {
2    public void draw(Shape s) {
3      s.draw(this);
4    }
5    public void
6    drawAll(List<? extends Shape> shapes){
7      shapes.add(0, new Circle());
8            // not allowed
9      for (Shape s: shapes) {
10         s.draw(this);
11     }
12   }
13 }
```

- metoda może być parametryzowana typem

```
static <T> void fromArrayToCollection(T[] a, Collection<T> c) {
    for (T o : a) {
        c.add(o); // Correct
    }
}
...
String[] sa = new String[100];
Collection<String> cs = new ArrayList<String>();

// T inferred to be String
fromArrayToCollection(sa, cs);
```

- metoda może być parametryzowana typem
- możemy mieć parametr i wildcards

```
class Collections {
    public static <T> void copy(List<T> dest, List<? extends T> src) {
    ...
}
lub
class Collections {
    public static <T, S extends T> void copy(List<T> dest, List<S> src) {
    ...
}
```

- metoda może być parametryzowana typem
- możemy mieć parametr i wildcards
- wildcards może być użyty do deklaracji pól i zmiennych lokalnych

```
class Canvas {
  public void draw(Shape s) {
    s.draw(this);
  }
  public static List<List<? extends Shape>>
  history = new ArrayList<List<? extends Shape>>();

  public void drawAll(List<? extends Shape> shapes) {
    history.add(shapes);
    for (Shape s: shapes) {
        s.draw(this);
    }
  }
}
```

- metoda może być parametryzowana typem
- możemy mieć parametr i wildcards
- wildcards może byż użyty do deklaracji pól i zmiennych lokalnych

```
1  public class App {
2    public static void main(String[] args) throws Exception {
3      List<Shape> shapes = new LinkedList<Shape>();
4      List<Circle> circles = new LinkedList<Circle>();
5      shapes.add(new Circle());
6      Canvas c = new Canvas();
7      c.drawAll(shapes);
8      c.drawAll(circles);
9      System.out.println(Canvas.history.size());  // 2
10   }
11 }
```