Programowanie w języku Java

Wykład 8: Klasy wewnętrzne

Klasy zagnieżdżone

```
class OuterClass {
 ...
 static class StaticNestedClass {
   ...
 }
 class InnerClass {
  ...
 }
}
```

Po co klasy zagnieżdżone?

- Grupowanie klas „lokalnych"
  - Gdy klasa jest wykorzystywana tylko w jednej klasie
- Ukrywanie implementacji
  - Klasy wewnętrzne mogą być prywatne
- Większa czytelność kodu

Statyczne klasy zagnieżdżone

- Nie ma dostępu do elementów klasy zewnętrznej
- Odwołanie: OuterClass.StaticNestedClass

np.

```
OuterClass.StaticNestedClass nestedObject = new
    OuterClass.StaticNestedClass();
```

Klasy wewnętrzne

- Nie mogą zawierać elementów statycznych
- Istnieją tylko w obiektach klasy zewnętrznej
- Mają dostęp do wszystkich elementów klasy zewnętrznej (również prywatnych)
- Tworzenie obiektów:

```
OuterClass.InnerClass innerObject =
   outerObject.new InnerClass();
```

Przykład

```
public class Parcel1 {
 class Contents {
  private int i = 11;
  public int value() { return i; }
 }
 class Destination {
  private String label;
  Destination(String whereTo) {
   label = whereTo;
  }
  String readLabel() { return label; }
 }
public void ship(String dest) {
  Contents c = new Contents();
  Destination d = new Destination(dest);
  System.out.println(d.readLabel());
 }
 public static void main(String[] args) {
  Parcel1 p = new Parcel1();
  p.ship("Tasmania");
 }
}
```

Odwołania do klas wewnętrznych

```
public class Parcel3 {
  class Contents {
    private int i = 11;
    public int value() { return i; }
  }
  class Destination {
    private String label;
    Destination(String whereTo) { label = whereTo; }
    String readLabel() { return label; }
  }
  public static void main(String[] args) {
    Parcel3 p = new Parcel3();
    Parcel3.Contents c = p.new Contents();
    Parcel3.Destination d = p.new  Destination("Tasmania");
  }
}
```

Dziedziczenie po klasie wewnętrznej

```
class WithInner {
  class Inner {}
}

public class InheritInner extends WithInner.Inner {
  //! InheritInner() {}
  InheritInner(WithInner wi) {
    wi.super();
  }
  public static void main(String[] args) {
    WithInner wi = new WithInner();
    InheritInner ii = new InheritInner(wi);
  }
}
```

Anonimowe klasy wewnętrzne

```
public class Parcel7 {
  public Contents contents() {
    return new Contents() {
      private int i = 11;
      public int value() { return i; }
    };
  }
  public static void main(String[] args) {
    Parcel7 p = new Parcel7();
    Contents c = p.contents();
  }
}
```

Klasy lokalne

- Deklaracja wewnątrz metody
- Rodzaj klasy wewnętrznej
- Dostęp do zmiennych lokalnych i parametrów metody (ale tylko final !!!)
  - Zapobiega zmianom wartości np. podczas tworzenia lokalnego obiektu
- Nie ma kwalifikatorów dostępu

Przykład

```
public class Parcel4 {
  public Destination dest(String s) {
    class PDestination implements Destination {
      private String label;
      private PDestination(String whereTo) {
        label = whereTo;
      }
      public String readLabel() { return label; }
    }
    return new PDestination(s);
  }
  public static void main(String[] args) {
    Parcel4 p = new Parcel4();
    Destination d = p.dest("Tanzania");
  }
}
```

Zagnieżdżone interfejsy

```
class A {
  interface B {
    void f();
  }
  public class BImp implements B {
    public void f() {}
  }
  private class BImp2 implements B {
    public void f() {}
  }
  public interface C {
    void f();
  }
  class CImp implements C {
    public void f() {}
  }
  private class CImp2 implements C {
    public void f() {}
  }
  private interface D {
    void f();
  }
  private class DImp implements D {
    public void f() {}
  }
  public class DImp2 implements D {
    public void f() {}
  }
  public D getD() { return new DImp2(); }
  private D dRef;
  public void receiveD(D d) { dRef = d;     dRef.f();
 }
}
```

```
interface E {
  interface G {
    void f();
  }
  // Redundant "public":
  public interface H {
    void f();
  }
  void g();
  // Cannot be private within an interface:
  //! private interface I {}
}
```

Implementacja zagnieżdżonych interfejsów

```
public class NestingInterfaces {
  public class BImp implements A.B {
    public void f() {}
  }
  class CImp implements A.C {
    public void f() {}
  }
  // Cannot implement a private interface
except
  // within that interface's defining class:
  //! class DImp implements A.D {
  //!  public void f() {}
  //! }
  class EImp implements E {
    public void g() {}
  }
  class EGImp implements E.G {
    public void f() {}
  }
  class EImp2 implements E {
    public void g() {}
    class EG implements E.G {
      public void f() {}
    }
  }
```

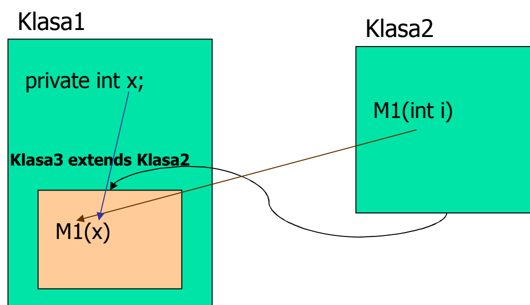
```
  public static void main(String[] args) {
    A a = new A();
    // Can't access A.D:
    //! A.D ad = a.getD();
    // Doesn't return anything but A.D:
    //! A.DImp2 di2 = a.getD();
    // Cannot access a member of the interface:
    //! a.getD().f();
    // Only another A can do anything with getD():
    A a2 = new A();
    a2.receiveD(a.getD());
  }
} ///:~
```


Przesłanianie klas wewnętrznych

```
class Egg {
  private Yolk y;
  protected class Yolk {
    public Yolk() { System.out.println("Egg.Yolk()"); }
  }
  public Egg() {
    System.out.println("New Egg()");
    y = new Yolk();
  }
}

public class BigEgg extends Egg {
  public class Yolk {
    public Yolk() { System.out.println("BigEgg.Yolk()"); }
  }
  public static void main(String[] args) {
    new BigEgg();
  }
}
```

```
New Egg()
Egg.Yolk()
```

Zastosowanie klas wewnętrznych



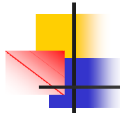
Wielodziedziczenie implementacji??

Wywołania zwrotne

```
interface Incrementable {
  void increment();
}

class Callee1 implements Incrementable {
  private int i = 0;
  public void increment() {
    i++;
    System.out.println(i);
  }
}

class MyIncrement {
  void increment() {
    System.out.println("Other operation");
  }
  static void f(MyIncrement mi) { mi.increment(); }
}

class Callee2 extends MyIncrement {
  private int i = 0;
  private void incr() {
    i++;
    System.out.println(i);
  }
  private class Closure implements Incrementable {
    public void increment() { incr(); }
  }
  Incrementable getCallbackReference() {
    return new Closure();
  }
}
```

```
class Caller {
  private Incrementable callbackReference;
  Caller(Incrementable cbh) { callbackReference = cbh; }
  void go() { callbackReference.increment(); }
}

public class Callbacks {
  public static void main(String[] args) {
    Callee1 c1 = new Callee1();
    Callee2 c2 = new Callee2();
    MyIncrement.f(c2);
    Caller caller1 = new Caller(c1);
    Caller caller2 = new Caller(c2.getCallbackReference());
    caller1.go();
    caller1.go();
    caller2.go();
    caller2.go();
  }
}
```

```
Other operation
1
2
1
2
```



Koniec