Encapsulation and abstraction

Adam Krechowicz

Packages

- Allows to group classes
- Prevents class names conflicts
- To indicate that class belong to package we use keyword *package*
- To use class from package we use keyword *import*

Packages

Packages have form of reversed internet addresses. It represents the place on disc drive.

```java
package com.wyklad.packages;

public class Package{

}
```

Class Package is inside package com.wyklad.packages
Directory structure com/wyklad/packages/Package.class

Importing

Single class

```
import com.wyklad.packages.Package;

public class Importowanie{
  Package p = new Package();
}
```

All classes from package

```
import com.wyklad.packages.*;

public class Importowanie{
  Package p = new Package();
}
```

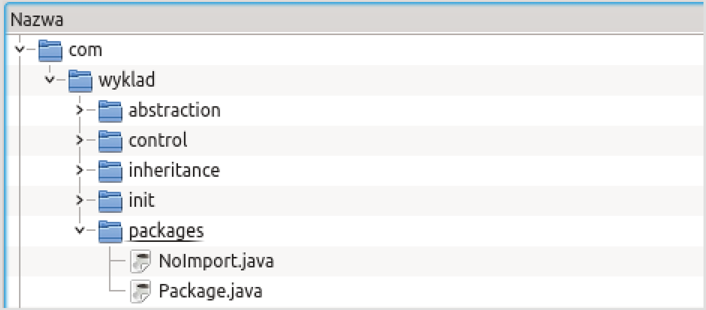
Importing

Without import keyword

```java
public class Importowanie{
  com.wyklad.packages.Package p = new com.wyklad.packages.Package();
}
```

Package organisation

Hard disc structure

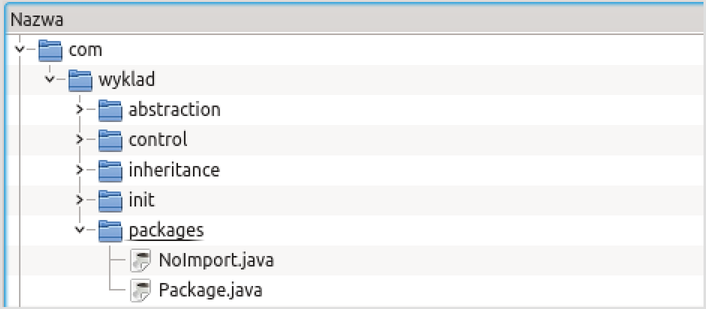


Compilation

- from main directory : javac com/wyklad/packages/Package.java
- from directory com/wyklad/packages : javac Package.java

Package organisation

Hard disc structure



Running

- only from main directory!
- java com.wyklad.packages.Package

Multiple location

Jeden.java

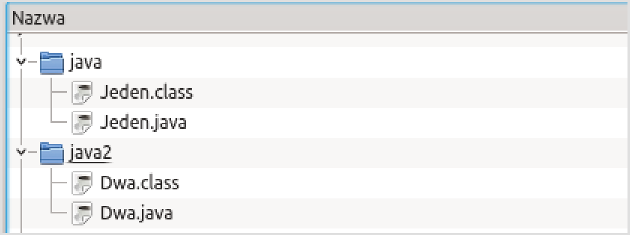
```java
public class Jeden{
        public void metoda(){
                System.out.println("metoda jeden");
        }
}
```

Dwa.java

```java
public class Dwa{
        public static void main(String[] args){
                Jeden j = new Jeden();
                j.metoda();
        }
}
```

Multiple locations

Hard disc structure



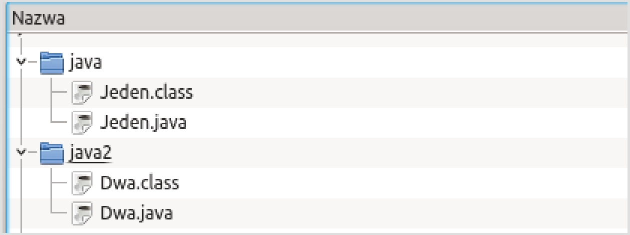
```
Nazwa
java
    Jeden.class
    Jeden.java
java2
    Dwa.class
    Dwa.java
```

Compilation

- cd java2
- javac Dwa.java
- error: cannot find symbol Jeden j = new Jeden();

Multiple locations

Hard disc structure

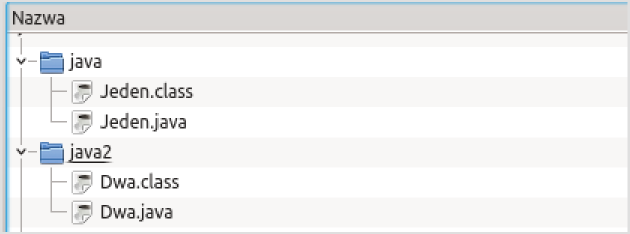

Nazwa
- java
  - Jeden.class
  - Jeden.java
- java2
  - Dwa.class
  - Dwa.java

Compilation

- cd java2
- javac -cp ../java Dwa.java
- javac -classpath ../java Dwa.java

Multiple locations

Hard disc structure

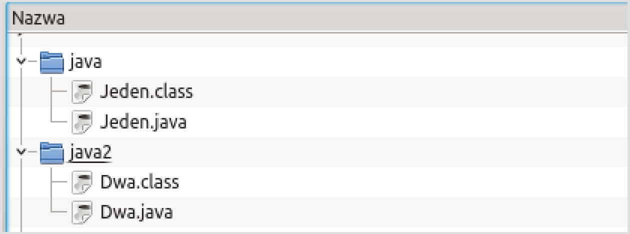

Nazwa
java
Jeden.class
Jeden.java
java2
Dwa.class
Dwa.java

Running

- cd java2
- java Dwa
- Exception in thread "main"java.lang.NoClassDefFoundError: Jeden

Multiple locations

Hard disc structure

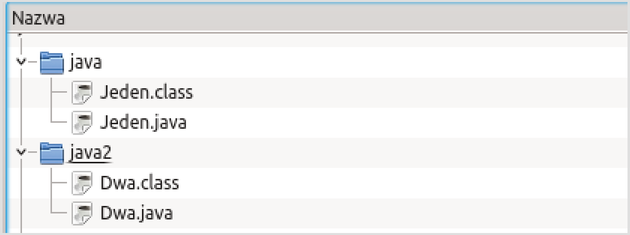

Nazwa
- java
  - Jeden.class
  - Jeden.java
- java2
  - Dwa.class
  - Dwa.java

Running

- cd java2
- java -cp ../java Dwa
- Error: Could not find or load main class Dwa

Multiple locations

Hard disc structure


Nazwa

- java
  - Jeden.class
  - Jeden.java
- java2
  - Dwa.class
  - Dwa.java

Running

- cd java2
- java -cp .:../java Dwa
- java -classpath .:../java Dwa

Visibility control

- It allows to hide some elements
- Provides class encapsulation
- Hides implementation
- Everyone see only what it have to see
- It refers to:
    - Methods
    - Fields
    - Classes

Types of visibility

Types of visibility

There are four visibility accessors:

- *public*
- *private*
- *protected*
- package (without keyword)

Public elements

```
class BasePublic{
    public void method1(){
        System.out.println("Other public method");
    }
}

public class Public extends BasePublic {

    public void method(){
        System.out.println("Public method");
        method1();
    }

    public static void main(String[] args){
        BasePublic bp = new BasePublic();
        bp.method1();
    }

}
```

Public elements

- Visible from other classes
- Visible in derived classes
- Visible from the same package
- Visibile from other packages

Private elements

```java
class AnotherPrivate{
    private void method1(){
        System.out.println("Other private method");
    }
}

public class Private{
    public static void main(String[] args){
        AnotherPrivate ap = new AnotherPrivate();
        //ap.method1();
    }
}
```

Private elements

```java
class AnotherPrivate{
    private void method1(){
        System.out.println("Other private method");
    }
}

public class Private extends AnotherPrivate {
    private void method(){
        System.out.println("Private method");
        //method1();
    }
}
```

Private elements

```java
class AnotherPrivate{
    private void method1(){
        System.out.println("Other private method");
    }
}

public class Private extends AnotherPrivate {
    private void method(){
        System.out.println("Private method");
        //method1();
    }

    public static void main(String[] args){
        Private p = new Private();
        p.method();
        AnotherPrivate ap = new AnotherPrivate();
        //ap.method1();
    }
}
```

Private elements

- Visible from the same class
- Invisible in derived class
- Invisible from the same package
- Invisible from different packages
- In case of classes refers only to inner classes

Protected elements

```java
class BaseProtected{
    protected void method1(){
        System.out.println("Protected method");
    }
}

public class Protected extends BaseProtected {

    protected void method(){
        System.out.println("Protected method");
        method1();
    }

    public static void main(String[] args){
        BaseProtected bp = new BaseProtected();
        bp.method1();
    }
}
```

Protected elements

- Visible in own class
- Visible in derived classes
- Visible from the same package
- Invisible from different packages
- In case of classes refers only to inner classes

Package elements

```java
public  class  Friendly{
    void  method1(){
        System.out.println("Other friendly  method");
    }
}

class  AnotherFriendly extends Friendly {

    void  method(){
        System.out.println("Friendly method");
        method1();
    }
}
```

Package element

- Visible in own class
- Invisible from derived class
- Visible from the same package
- Invisible from other packages

Rules for grant visibilities

- Element should always have as minimal visibility as possible
- If possible use private
- Fields should be private
- Access to fields should be performed by access methods (getField() and setField())
- Constants are exceptions which are typical public
- In source file can be only one public class (with the same name as filename)

Changing visibility

```java
public class Changing {
    public void publicMethod(){}
    protected void protectedMethod(){}
    private void privateMethod(){}
    void packageMethod(){}
}

class ExtendedChanging extends Changing{
    //@Override
    //public void protectedMethod(){}
    @Override
    protected void packageMethod(){}
    //@Override
    //private void publicMethod(){}
    //@Override
    //public void privateMethod(){}
}
```

Abstract classes

```
abstract class Abstract{
    abstract void method();
}
```

- method without implementation should be declared as abstract
- if there is at least one abstract method in the class the class itself also needs to be declared abstract
- class without abstract method can also be abstract

Abstract classes

```
abstract class Abstract{
    abstract void method();
}
public class AbstractTest{

    public static void main(String[] args){
        //Abstract a = new Abstract();
    }

}
```

Abstract classes

```java
abstract class Abstract{
    abstract void method();
}
public class AbstractTest extends Abstract {

    @Override
    void method() {
        System.out.println("Implementacja");
    }

    public static void main(String[] args){
        AbstractTest at = new AbstractTest();
        Abstract a = new AbstractTest();
    }

}
```

Abstract classes

```java
abstract class Abstract{
    abstract void method();
}

abstract class AnotherAbstract extends Abstract{

}
```

Interfaces

```java
interface SomeInterface{
    public void method();
}
public class Interfaces implements SomeInterface {

    @Override
    public void method() {
        System.out.println("Implementacja");
    }

    //SomeInterface si = new SomeInterface();
    Interfaces i = new Interfaces();
    SomeInterface si = new Interfaces();
}
```

Interfaces

```
interface SomeInterface{
    public void method();
}

interface AnotherInterface extends SomeInterface{
    public void method1();
}

abstract class YetAnotherClass implements SomeInterface{

}
```

Interfaces and Abstract classes

- Class may inherit only from one (abstract) class
- Class may implement many interfaces
- Abstract class may contains part of implementation
- Interace typically do not contain implementation

Serialization

```
import java.io.Serializable;

public  class  SerializableClass implements Serializable {

}
```

Cloning

```
public class Cloning implements Cloneable {
    int wartosc;
    String nazwa;
    public Cloning(int wartosc, String nazwa){
        this.wartosc = wartosc;
        this.nazwa = nazwa;
    }
    public String toString(){
        return nazwa + " " + wartosc;
    }
    public static void main(String[] args){
        try {
            Cloning c1 = new Cloning(123, "hello");
            Cloning c2 = (Cloning)c1.clone();
            System.out.println(c1);
            System.out.println(c2);
        } catch (CloneNotSupportedException ex) {
            Logger.getLogger(Cloning.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```

Comparison

```java
public class ComparableClass implements Comparable<ComparableClass> {
    int pole;
    public ComparableClass(int pole){
        this.pole = pole;
    }
    public int compareTo(ComparableClass o) {
        return this.pole - o.pole;
    }
    public String toString(){
        return ""+pole;
    }
    public static void main(String[] args){
        Random r = new Random();
        ComparableClass[] cc = new ComparableClass[5];
        for (int i = 0; i < 5; i++)
            cc[i] = new ComparableClass(r.nextInt() % 10);
        Arrays.sort(cc);
        System.out.println(Arrays.toString(cc));
    }
```

Comparison

```
class ComparatorClass implements Comparator<ComparableClass>{
    public int compare(ComparableClass o1, ComparableClass o2) {
        return o2.pole − o1.pole;
    }
}
public class ComparableClass{
    int pole;
    public ComparableClass(int pole){
        this .pole = pole;
    }
    public static void main(String[] args){
        Random r = new Random();
        ComparableClass[] cc = new ComparableClass[5];
        for (int i = 0; i < 5; i++)
            cc[i] = new ComparableClass(r.nextInt() % 10);
        Arrays.sort(cc, new ComparatorClass());
        System.out.println(Arrays.toString(cc));
    }

}
```

Composition

```java
class  Nadrzedna{
    void metoda(){
        System.out.println("Metoda");
    }
}

class  Inheritance extends Nadrzedna{
    void metoda1(){
        metoda();
    }
}

public  class  Composition {
    private  Nadrzedna p = new Nadrzedna();
    void metoda2(){
        p.metoda();
    }
}
```

Inheritance and composition

- Composition may solve the problem of multi inheritance
- Inheritance allows class hierarchy, casting and polymorphism
- By using composition objects can be created in a lazy way (only if needed)

Delegation

```java
interface NaszInterfejs{
    public void metoda();
}
class Implementacja1 implements NaszInterfejs{
    @Override
    public void metoda() {
        System.out.println("Pierwsza implmenetacja");
    }
}
class Implementacja2 implements NaszInterfejs{
    @Override
    public void metoda() {
        System.out.println("Druga implmenetacja");
    }
}
```

Delegation

```java
public class Delegation implements NaszInterfejs {
    NaszInterfejs ni = new Implementacja1();

    @Override
    public void metoda() {
        ni.metoda();
    }

    public static void main(String[] args){
        Delegation d = new Delegation();
        d.metoda();
    }
}
```

Delegation

```java
public class Delegation implements NaszInterfejs {
    NaszInterfejs ni = new Implementacja1();
    public void zmiana(){
        ni = new Implementacja2();
    }
    @Override
    public void metoda() {
        ni.metoda();
    }
    public static void main(String[] args){
        Delegation d = new Delegation();
        d.metoda();
        d.zmiana();
        d.metoda();
    }
}
```

Anonymous objects

```java
public class Anonymous {

    public void method(){
        System.out.println("Hello world");
    }

    public static void main(String[] args){
        new Anonymous().method();
    }

}
```

Anonymous classes

```java
interface Interfejs{
    public void metoda();
}

public class Anonymous {

    public static void main(String[] args){

        Interfejs i = new Interfejs() {
            @Override
            public void metoda() {
                System.out.println("Implementacja");
            }
        };
    }
}
```

THE END!

Additional reading:

- https://docs.oracle.com/javase/tutorial/java/package/packages.html
- https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.htm
- https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html
- https://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html
- Thinking in Java chapters by Bruce Eckel:
  - Access control
  - Interfaces