

MSc/ICY Software Workshop

Packages, Inheritance

Manfred Kerber www.cs.bham.ac.uk/~mmk

10 November 2016

Object-Oriented Programming (Revisited)

Distinguish

- **Classes**, e.g., `Employee`, `Invoice`
- **Objects**, e.g., `employeejohn`, `employeeMary`
created by a **Constructor**, e.g.
`public Employee (String firstName, ...`
- **Methods**, e.g. `getFirstName()`, `toString()`
- **overriding** vs **overloading** vs **polymorphism**.

Note, although **overriding** and **overwriting** sound similar they are different. With **overriding**, the old method is still there. If you, however, **overwrite** the old value of a variable, it is gone. With **overriding** always the most specific method (in its environment) is taken.

It is good practice to optionally write **@Override**. (Compiler checks whether the method actually does override.)

- packages as collection of Java classes that belong together.
- “Packages are Java libraries of classes. `import` statements make classes from a package available to your program.”
[Absolute Java, 4th Edition by Walter Savitch, 2010, p. 90]
- Packages determine the access of variables and methods. We have seen up to now two access modifiers `public` and `private`. There are two more `protected` and the default, which is package access. The difference can best be seen by an example.

Packages – An Example

From [Absolute Java, 4th Edition by Walter Savitch, 2010, p.481]

Inside the same package:

```
package somePackage;
public class A {
    public      static int v1 = 1;
    protected static int v2 = 2;
               static int v3 = 3; // package access
    private    static int v4 = 4;
}
```

Packages – An Example

From [Absolute Java, 4th Edition by Walter Savitch, 2010, p.481]

Inside the same package:

```
package somePackage;
public class A {
    public      static int v1 = 1;
    protected static int v2 = 2;
                static int v3 = 3; // package access
    private    static int v4 = 4;
}

package somePackage;
public class B {
    public void BPrint() {
        System.out.println(A.v1); //access
        System.out.println(A.v2); //access
        System.out.println(A.v3); //access
        System.out.println(A.v4); //no access, compiler error
    }
}
```

Packages – An Example (Cont'd)

From [Absolute Java, 4th Edition by Walter Savitch, 2010, p.481]

Inside the same package and subclass

```
package somePackage;
public class A {
    public      static int v1 = 1;
    protected static int v2 = 2;
                static int v3 = 3; // package access
    private static int v4 = 4;
}

package somePackage;
public class C extends A {
    public void CPrint() {
        System.out.println(v1); //access
        System.out.println(v2); //access
        System.out.println(v3); //access
        System.out.println(v4); //no access, compiler error
    }
}
```

Packages – An Example (Cont'd)

From [Absolute Java, 4th Edition by Walter Savitch, 2010, p.481]

Outside the same package but subclass

```
package somePackage;
public class A {
    public      static int v1 = 1;
    protected static int v2 = 2;
                static int v3 = 3; // package access
    private    static int v4 = 4;
}

import somePackage.A;
public class D extends A {
    public void DPrint() {
        System.out.println(A.v1); //access
        System.out.println(A.v2); //access
        System.out.println(A.v3); //no access, compiler error
        System.out.println(A.v4); //no access, compiler error
    }
}
```

Packages – An Example (Cont'd)

From [Absolute Java, 4th Edition by Walter Savitch, 2010, p.481]

Outside the same package and no subclass

```
package somePackage;

public class A {
    public      static int v1 = 1;
    protected static int v2 = 2;
                static int v3 = 3; // package access
    private    static int v4 = 4;
}

import somePackage.A;
public class E {
    public void EPrint() {
        System.out.println(A.v1); //access
        System.out.println(A.v2); //no access, compiler error
        System.out.println(A.v3); //no access, compiler error
        System.out.println(A.v4); //no access, compiler error
    }
}
```


No Cyclic Class Structure

We cannot have a class A1

```
package myTest;
```

```
public class A1 extends A2 {  
  
}
```

and a class A2

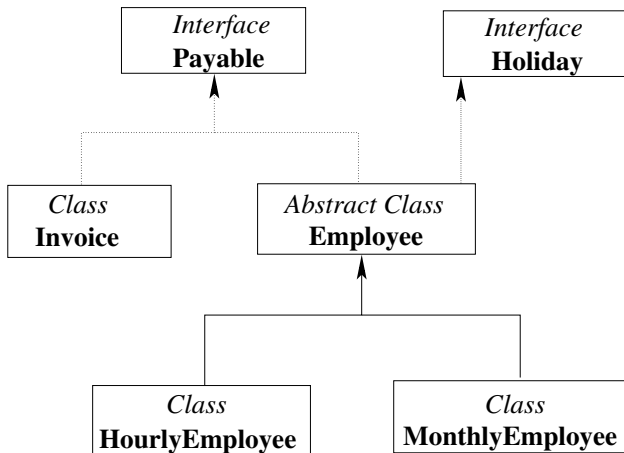
```
package myTest;
```

```
public class A2 extends A1 {  
  
}
```

- Classes which have subclasses, but there are no direct objects of that class. E.g., abstract class `Employee` with subclasses: `MonthlyEmployee` and `HourlyEmployee`.

Abstract Classes (Cont'd)

public class MonthlyEmployee extends Employee and
public class HourlyEmployee extends Employee



Abstract Methods

Just as Interfaces provide only the header of a method without an implementation we may have in an abstract class also **abstract methods** for which only the header but no implementation is given in the abstract class. In this case, it is necessary to override the abstract method in each subclass with a concrete method.

e.g., `public abstract int getPaymentAmount();`

final

Just as variables that are declared `final`, also methods can be declared `final`. It means that they **CANNOT** be overridden in a subclass.

final

Just as variables that are declared `final`, also methods can be declared `final`. It means that they **CANNOT** be overridden in a subclass.

E.g., you may want to `disallow` in the `BankAccount` class that the `withdraw` method is `overridden` from

```
public void withdraw(int amount) {  
    if (balance >= amount){  
        balance = balance - amount;  
    }  
}
```

to something like

```
public void withdraw(int amount) {  
}
```

for some subclass.

Polymorphism

Distinguish in inheritance whether you **create** a new method with **different arguments** or **override an existing one** (with the same number and types of arguments).

In superclass:

```
public String toString() {  
    return String.format("%s %s, NI: %s ",  
        getFirstName(), getLastName(), getnI());  
}
```

Override in subclass

```
public String toString() {  
    return String.format("%s, salary: %d",  
        super.toString(),  
        getPaymentAmount());  
}
```

Polymorphism (Cont'd)

In superclass:

```
public String toString() {  
    return String.format("%s %s, NI: %s ",  
        getFirstName(), getLastName(), getnI());  
}
```

Is NOT overridden in subclass by

```
public String toString(boolean verbose) {  
    if (verbose) {  
        return String.format("%s\nSalary: %d",  
            super.toString(),  
            getPaymentAmount());  
    } else {  
        return this.toString();  
    }  
}
```


We said that with `super` it is possible to access public methods (and public variables) in the superclass. Note that the usage is restricted and it is NOT possible to use e.g.

`super.super.methodName()`; since this would contradict the idea of class structuring.

[Horstmann, Big Java, p.319]:

“A class invariant is a statement about an object that is true after every constructor and that is preserved by every mutator (provided that the caller respects all preconditions).” (mutator = setter)

An example is that the amount in a `BankAccount` is always bigger than or equal to 0 (or bigger than or equal to the negative `overDraftLimit` in a `BankAccountWithOverdraft`).