MSc/ICY Software Workshop
Packages, Inheritance

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

Tuesday 12 November 2019

Overview

1. Pocket calculator computations, base types, simple strings, variables, static methods, JavaDoc
   Wed/Thu/Fri: 1st Lab Lecture (login, editor, javac, javadoc)
2. Classes, objects, methods, JUnit tests
   Wed/Thu/Fri: 2nd Lab Lecture (Eclipse)
3. Conditionals, 'for' Loops, arrays, ArrayList
4. Exceptions, I/O (Input/Output)
5. Functions, interfaces
6. Sub-classes, inheritance, abstract classes
7. **Inheritance (Cont'd), packages**
8. Revision
9. Graphics
10. Graphical User Interfaces
11. Graphical User Interfaces (Cont'd)

Changes possible

Conditional Operator

It allows to replace

```
if (cond) {
   var = value1;
} else {
   var = value2;
}
```

by the more concise

```
var = cond ? value1 : value2;
```


Checked vs Unchecked Exceptions

- Unchecked Exceptions may or may not be caught by the program.
  They deal typically with problems that are under control of the programmer (e.g., an `ArrayIndexOutOfBoundsException`)
- Checked Exceptions must be caught by the program. These deal typically with problems that are NOT under control of the programmer (e.g. whether a file exists or is accessible, `FileNotFoundException` or `AccessDeniedException`).
  The Java compiler enforces a catch statement for a checked exception.

Checked vs Unchecked Exceptions

- Unchecked Exceptions may or may not be caught by the program.
  They deal typically with problems that are under control of the programmer (e.g., an `ArrayIndexOutOfBoundsException`)
- Checked Exceptions must be caught by the program. These deal typically with problems that are NOT under control of the programmer (e.g. whether a file exists or is accessible, `FileNotFoundException` or `AccessDeniedException`).
  The Java compiler enforces a catch statement for a checked exception.
  UNLESS the task is passed on to the calling method by explicitly writing something like

```
public MyType myMeth() throws FileNotFoundException{
...
}
```


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

- 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
    }
}
```

The same rules apply for methods.

No Cyclic Class Structure

We cannot have a class A1

```
package myTest;

public class A1 extends A2 {

}
```

and a class A2

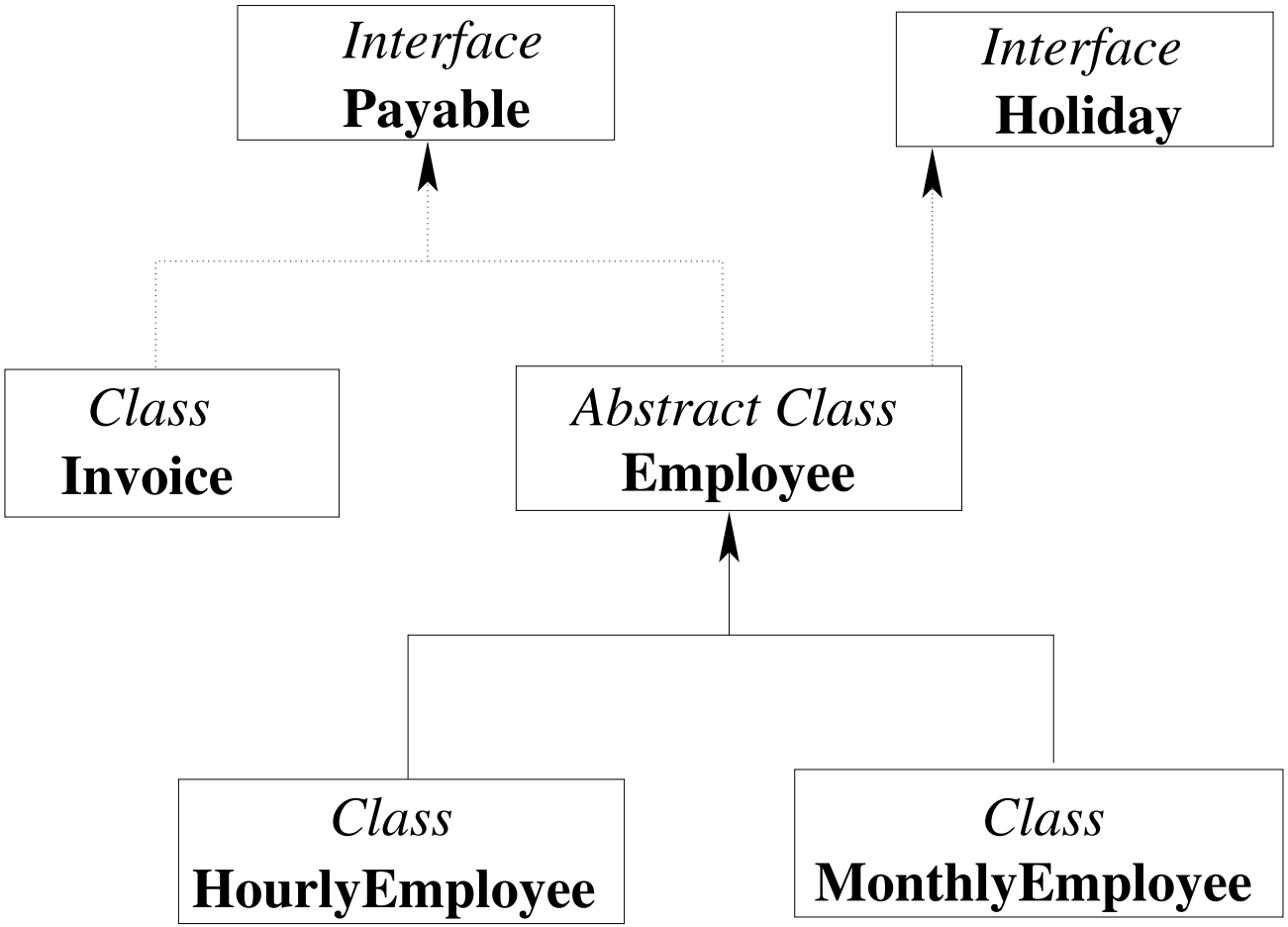
```
package myTest;

public class A2 extends A1 {

}
```


Abstract Classes

- 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();
    }
}
```


super

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 and encapsulation.

Class invariants

[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`).

Bugs and Debugging

"A software bug is an error, flaw, failure or fault in a computer program or system that causes it to produce an incorrect or unexpected result, or to behave in unintended ways. The process of fixing bugs is termed "debugging" and often uses formal techniques or tools."
(from `https://en.wikipedia.org/wiki/Software_bug`)

Debugging and Eclipse

Eclipse has a debug mode (the bug button left of the run button, or under the Run Tab choose Debug).
Caution: THE DEBUGGER DOES NOT MYSTERIOUSLY DEBUG YOUR PROGRAM. IT DOES:

- allow you to set breakpoints in the program (points at which you want to check the values of some of the variables),
- to step from breakpoint to breakpoint and inspect the value of the variables in order to see whether they are what you expect them to be. [As opposed to running the program until it terminates (either normally or abruptly).]

