

MSc/ICY Software Workshop Packages, Inheritance

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

Wednesday 14 November 2018

1 / 20 ©Manfred Kerber

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.

3 / 20 ©Manfred Kerber

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.)

4 / 20 ©Manfred Kerber

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
    }
}
```

6 / 20 ©Manfred Kerber

Conditional Operator

It allows to replace

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

by the more concise

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

2 / 20 ©Manfred Kerber

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{`
...
}

3 / 20 ©Manfred Kerber

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.

5 / 20 ©Manfred Kerber

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
    }
}
```

7 / 20 ©Manfred Kerber

```
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
    }
}
```

20 ©Manfred Kerber

```
classDiagram
    class Employee {
        <<abstract>>
    }
    class Invoice
    class Holiday
    class Payable
    class HourlyEmployee
    class MonthlyEmployee

    Employee <|-- Invoice
    Employee <|-- HourlyEmployee
    Employee <|-- MonthlyEmployee
    Employee ..> Payable
    Employee ..> Holiday
```

The diagram illustrates the relationships between several classes and interfaces. At the top, there are two interfaces: **Interface Payable** and **Interface Holiday**. Below them is the **Abstract Class Employee**. Two dashed arrows point from **Abstract Class Employee** to **Interface Payable** and **Interface Holiday**, indicating that **Employee** implements both interfaces. Below **Abstract Class Employee**, there are three classes: **Class Invoice**, **Class HourlyEmployee**, and **Class MonthlyEmployee**. Solid lines connect **Class Invoice**, **Class HourlyEmployee**, and **Class MonthlyEmployee** to **Abstract Class Employee**, indicating inheritance. **Class Invoice** is positioned to the left of **Abstract Class Employee**, while **Class HourlyEmployee** and **Class MonthlyEmployee** are positioned below it.

12 / 20 ©Manfred Kerber

for some subclass.

14 / 20 ©Manfred Kerber

```
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
    }
}
}
```

- 11 / 20 ©Manfred Kerber

13 / 20 ©Manfred Kerber

15 / 20 ©Manfred Kerber

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();
    }
}
```

16 / 20 ©Manfred Kerber

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`).

18 / 20 ©Manfred Kerber

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).]

20 / 20 ©Manfred Kerber

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.

17 / 20 ©Manfred Kerber

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)

19 / 20 ©Manfred Kerber