Praktische Softwaretechnologie

Lecture 4.

Károly Bósa
(Karoly.Bosa@jku.at)

Research Institute for Symbolic Computation
(RISC)

Visibility

For Fields and Methods: there are 4 different visibilities

modifier	Visible
private	Only in the same class
(default)	In each class of the same package
protected	In the same package and the inherited/sub classes
public	From every class

Classes can be only public or default

The real rules for the visibility is more compound, read more if it is necessary.

Principles of Visibility

- **Package**: a well arranged unit of cooperating classes
- **Fields** are almost always private
- **Classes** are public, if they part of the facade/interface of the package
- The **methods** are public, if they are part of the documented interface of the class.
- The **methods** are private, if they are contains the details of some implementations/algorithms.
- The modifier protected is used only within the classes which were designed for inheritance

The Class java.lang.Object

Karoly.Bosa@jku.at

If there is not explicitly given parent class ➔ java.lang.Object

➔Each class is derived/inherited from the class Object

The Object defines some methods:

- equals(Object o)
- hashCode()
- toString()
- …

These can/should be overwritten by the inherited classes.

The Class java.lang.Object

Karoly.Bosa@jku.at

If there is not explicitly given parent class ➔ java.lang.Object

➔Each objects is derived/inherited from the class Object

The Object defines some methods:

- equals(Object o)
- hashCode() $\longrightarrow$

```
public int hashCode() {
    return 31*x + y;
}
```

- toString()
- …

These can/should be overwritten by the inherited classes.

instanceof

For testing a dynamic type in runtime:

p instanceof WeightedPoint ➔ It is true if p is an instance object of class WeightedPoint or of one of its subclasses

instanceof

For testing a dynamic type in runtime:

p instanceof WeightedPoint ➔ It is true if the type of value of p is an instance object of WeightedPoint or of one of its subclasses

Typical application:

```
class Point {
    ...
    public boolean equals(Object o) {
        if (o instanceof Point) {
            Point other = (Point)o;
            return this.x==other.x && this.y == other.y;
        } else {
            return false;
        }
    }
}
```

The Modifier final

The values of data fields must not be modified after their initialization:

```
private final int x;
```

It is useful to indicate that the value cannot change!

Static constants (values are always knows at compile time):

```
public static final int MONTHS_PER_YEAR = 12;
```

In case of methods, this modifier forbids the overriding:

```
public final void m(…) {… }
```

In case of classes, the modifier does not allow to create the sub classes:

```
public final class A {…}
```

Failure Handling in C

Karoly.Bosa@jku.at

The error handling in C happens either via return value:

```
c = getc();
if (c == EOF) {
  … Failure Handling
}
```

Or via the global variable errno:

```
if ((fd = fopen(“file.txt, “r”)) == -1) {
  if (errno == EACCESS) “{
     … Failure Handling
  }
  …
}
```

➔Error prone, diverts the attention from the important issues

➔Failure handling is often neglected

Exceptions

The error handling in Java: Exceptions

Exceptions are objects instantiated from one of the sub classes of java.lang.Exception

Exceptions are “thrown”:

```
    throw new java.io.IOException();
```

Exceptions are “caught::

```
    try {
        …Here an Exception can be thrown…
    } catch (java.io.IOException e) {
        …Here java.io.IOException e can be treated…
    }
```

➔ A “throw” can be within the called methods

Interrupted Termination

“Official” terminology: each statement (and each block) can end/complete as follows:

- Performing until the last statement/command
  ➔ normal end

- Performing until: break continue, return or throw
  ➔ interuppted termination

Interrupted Termination always has a particular reason, for instance in case of methods:

- a method is interrupted by break or throw
- a method completes normally with return value

Interrupted Termination 2

Karoly.Bosa@jku.at

The statement *throw e* always causes a interrupted termination

When a statement located in a block, an if –then-else or a loop is interrupted (and the blocks/methods where it happened do not contain a corresponding try-catch block), them:

➔ Outer blocks will be interrupted as well because of the same reason.
➔ The called methods will be interrupted as well because of the same reason.

Consequently: An Exception is able to jump back from more than one embedded blocks or method calls and finally it will cause an interrupted termination of the whole program.

It can be stopped only by applying a try-catch structure

try-catch **Semantic**

Karoly.Bosa@jku.at

```
TC = try {
          ...P1...
     } catch (e Exc) {
          ... P2 ...
     }
```

- P1 terminates normally ➔ TC terminates normally
- P1 is interrupted by an Exception e of Type Exc ➔ P2 is called and TC terminates as P2
- P1 is interrupted by another reason ➔ TC is interrupted because of the same reason

try-catch-finally **Semantic**

```
TCF = try {
            ...P1...
      } catch (e Exc) {
            ... P2 ...
      } finally {
            ... P3 ...
      }
```

Like the try-catch, but:

- In all circumstances P3 will be performed at last.
- If P3 terminates normally ➔ TCF terminates normally
- If P3 is interrupted ➔ TCF is interrupted because of the same reason

Throws Declaration

Karoly.Bosa@jku.at

Exceptions can be thrown via methods, but they have to be declared:

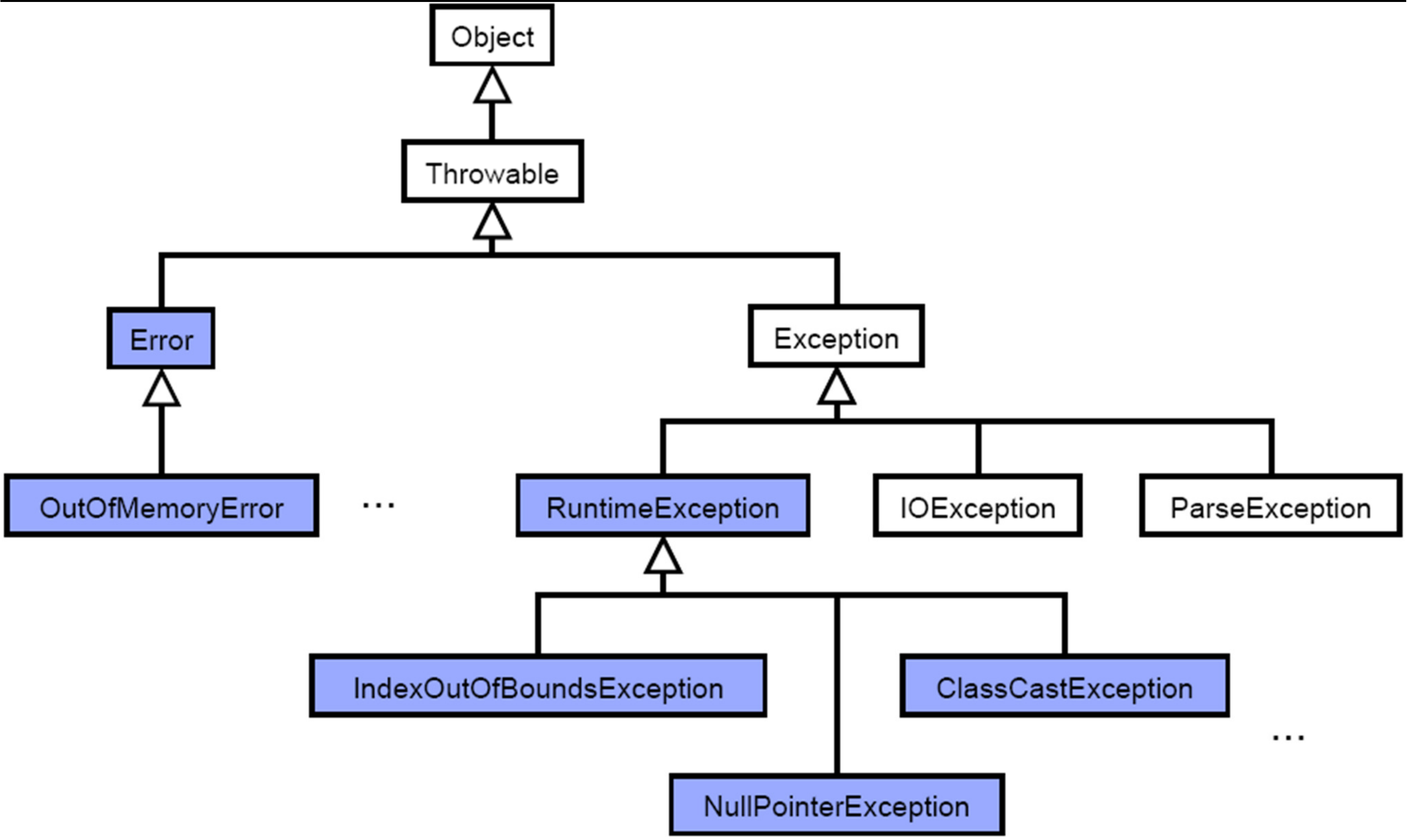
```
public void writeData() throws java.io.IOException {
    …
}
```

It is required for all Exceptions, which were thrown, but not caught (in the method)

Exception: There are some Exceptions which are “unchecked” and they do not need to be declared.

The Throwable Hierarchy

Karoly.Bosa@jku.at



Recall the HelloWorld2 Example

```
import java.io.*;

public class HelloWorld2 {

  public static void main (String[] args) throws IOException {
     PrintWriter out  = new PrintWriter(new FileWriter(“hello.txt”));
     out.println(“Hello World!”);
     out.close();
  }
}
```

Own Exceptions

Karoly.Bosa@jku.at

There are 2 predefined constructors usually:

```
public class EmptyStackException extends RuntimeException {
   public EmptyStackException() {
      super();
   }

   public EmptyStackException(String detail) {
      super(detail);
   }
}
```

null

Karoly.Bosa@jku.at

The value null is comprised by all reference types:

```
Point p = null;
int[] a = null;
String s = null;
```

Accessing to Data Fields and Methods via null is not possible
➔ NullPointerException in runtime

Casts

Cast = Type Conversion

- Between primitive types with different runtime semantic
  - `(double)1 == 1.0`
  - `(int)1.3 == 1`
  - `(int)0x100000001L == 1`

Casts 2

Cast = Type Conversion

- Between primitive types with different runtime semantic
- Between reference types

```
Point p = new WeightedPoint();
...
double w = p.weight;     // Compiler error
WeightedPoint wp = p;    // Compiler error
```

Casts 2

Cast = Type Conversion

- Between primitive types with different runtime semantic
- Between reference types

```
Point p = new WeightedPoint();
...
double w = ((WeightedPoint)p).weight;
WeightedPoint wp = (WeightedPoint)p;
```

Casts 2

Cast = Type Conversion

- Between primitive types with different runtime semantic
- Between reference types

```
Point p = new WeightedPoint();
...
double w = ((WeightedPoint)p).weight;
WeightedPoint wp = (WeightedPoint)p;
```

  - It changes only static types, but not dynamic types or values
  - In case of any error it throws ClassCastException

Interfaces

```
public interface Stack {
   void push(String o);
   String pop();
   void clear();
   boolean isEmpty();
}
```

Interfaces

Karoly.Bosa@jku.at

```
public interface Stack {
   void push(String o);
   String pop();
   void clear();
   boolean isEmpty();
}

public class BoundedArrayStack implements Stack {
   public void push(String o) {...}
   public String pop() {...}
   public void clear() {...}
   public boolean isEmpty() {...}
   ...
}
```

The class implements the declared methods of the interface

Interfaces

```
public interface Stack {
   void push(String o);
   String pop();
   void clear();
   boolean isEmpty();
}

public class BoundedArrayStack implements Stack {
   public void push(String o) {...}
   public String pop() {...}
   public void clear() {...}
   public boolean isEmpty() {...}
   ...
}

public class LinkedListStack implements Stack {
   public void push(String o) {...}
   ...
}
```

Multiple Inheritance with Interfaces

```
public interface Bounded {
   public int getMaxSize();
}
```

Multiple Inheritance with Interfaces

```
public interface Bounded {
   public int getMaxSize();
}

public class BoundedArrayStack implements Stack, Bounded {
   public void push(String o) {...}
   public String pop() {...}
   public void clear() {...}
   public boolean isEmpty() {...}
   public int get getMaxSize() {...}
   ...
}
```

Multiple Inheritance with Interfaces

Karoly.Bosa@jku.at

```
public interface Bounded {
   public int getMaxSize();
}

public class BoundedArrayStack implements Stack, Bounded {
   public void push(String o) {...}
   public String pop() {...}
   public void clear() {...}
   public boolean isEmpty() {...}
   public int get getMaxSize() {...}
   ...
}

public interface BoundedStack extends Bounded, Stack {
}
```

Multiple Inheritance with Interfaces

Karoly.Bosa@jku.at

```
public interface Bounded {
   public int getMaxSize();
}

public class BoundedArrayStack implements Stack, Bounded {
   public void push(String o) {...}
   public String pop() {...}
   public void clear() {...}
   public boolean isEmpty() {...}
   public int get getMaxSize() {...}
   ...
}

public interface BoundedStack extends Bounded, Stack {
}
```

In C++: there exists multiple inheritance
➔ complicated rules

Elements in Interfaces

- Method declarations:
  - Always (implicit) public
  - Never static
- No constructors
- Fields
  - Always public static final, thus constants
- Interfaces can be extended only with other interfaces

Usage of Interfaces

```
public static void fillStack(Stack s1) {
   s1.push("test");
   ...
}

public static void transferStack(Stack from, Stack to) {
   while (!from.isEmpty()) {
      to.push(from.pop())
   }
}

public static void main(String[] args) {
   Stack s1 = new BoundedArrayStack(10);
   Stack s2 = new LinkedListStack();
   fillStack(s1);
   System.out.println(s1);
   transferStack(s1,s2);
   System.out.println(s2);
}
```

Abstract Classes

A class is abstract, if it is declared as follows:

```
public abstract class AbstractStack {
    …
}
```

Abstract Classes

A class is abstract, if it is declared as follows:

```
public abstract class AbstractStack {
    …
}
```

An abstract class

- can contain *abstract methods* without implementation
  `public abstract void push(String s)`
  ➔ they will be overwritten by (non-abstract) implemented sub classes

Abstract Classes

A class is abstract, if it is declared as follows:

```
public abstract class AbstractStack {
    …
}
```

An abstract class

- can contain *abstract methods* without implementation

  `public abstract void push(String s)`

  ➔ they will be overwritten by (non-abstract) implemented sub classes

- Cannot be instantiated

  `new AbstractStack` ➔Compiler error!

Typical Application of Abstract Classes

Karoly.Bosa@jku.at

```
public abstract class AbstractStack implements Stack {

   public abstract String pop();
   public abstract void push(String s);
   public abstract String isEmpty();

   /** Remove all elements from the stack.
    * This default implementiation repeatedly calls pop() */
   public void clear() {
      while(!isEmpty()) {
         pop();
      }
   }
}
```

➔BoundedStack extends AbstractStack will inherit clear().
➔However it can be overwritten with a more efficient implementation.

Enumeration Types

Karoly.Bosa@jku.at

Types with a prescribed, finite quantity of discrete values

```
public class Apple {
   public static final int FUJI = 0;
   public static final int BOSKOP = 1;
   public static final int GRANNY_SMITH = 2;
}

public class Pear {
   public static final int WILLIAMS_CHRIST = 0;
   public static final int ABATE_FETEL = 1;
   public static final int MOSTBIRNE = 2;
}

... int obstsalat = 2*FUJI/BOSKOP + 1*WILLIAMS_CHRIST
```

Enumeration Types

Karoly.Bosa@jku.at

Types with a prescribed, finite quantity of discrete values

```
public class  Apple  {
   public static final int FUJI = 0;
   public static final int BOSKOP = 1;
   public static final int GRANNY_SMITH = 2;
}

public class  Pear  {
   public static final int WILLIAMS_CHRIST = 0;
   public static final int ABATE_FETEL = 1;
   public static final int MOSTBIRNE = 2;
}

... int obstsalat = 2*FUJI/BOSKOP + 1*WILLIAMS_CHRIST
```

➔There is no type conversions

➔ There is not revision for the (legal) values

Enumeration Types 2

Karoly.Bosa@jku.at

```
public enum Day {
        SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
        THURSDAY, FRIDAY, SATURDAY
}
```

Enumeration Types 2

Karoly.Bosa@jku.at

```
public class EnumTest {
    Day day;
    public EnumTest(Day day) { this.day = day;}
    public void tellItLikeItIs() {
       switch (day) {
              case MONDAY: System.out.println("Mondays are bad."); break;
              case FRIDAY: System.out.println("Fridays are better."); break;
              case SATURDAY:
              case SUNDAY: System.out.println("Weekends are best."); break;
              default: System.out.println("Midweek days are so-so."); break;
       }
    }
    public static void main(String[] args) {
              EnumTest firstDay = new EnumTest(Day.MONDAY);
              firstDay.tellItLikeItIs();
              EnumTest thirdDay = new EnumTest(Day.WEDNESDAY);
              thirdDay.tellItLikeItIs();
              EnumTest fifthDay = new EnumTest(Day.FRIDAY);
              fifthDay.tellItLikeItIs();
              EnumTest sixthDay = new EnumTest(Day.SATURDAY);
              sixthDay.tellItLikeItIs();
              EnumTest seventhDay = new EnumTest(Day.SUNDAY);
              seventhDay.tellItLikeItIs();
    }
```

Garbage Collection

Karoly.Bosa@jku.at

In C:

```
    int[] p = malloc(1024 * sizeof(int));
    ...

    free(p);
```

In Java:

```
    int[] p = new int[1024];
    ...
```

Release an object, if there is not any reference pointing to it any more. (a null value should be given to the variable).

➔ It can avoid lots of problem related to illegal references
➔ Release of network sockets, database connections, etc. are still manual

Recommended to Read

Reading and completing the course material from the online Java Tutorial:

http://download.oracle.com/javase/tutorial/java/index.html

- Interfaces and Inheritance

Exercise 5 (Extension for Exercise 4)

Karoly.Bosa@jku.at

Extend the implementation "stack" (from Exercise 4) with a sub class (inherited from class Stack) called DebugStack, in which:

- the constructor of DebugStack waits for a name String which is stored in a field
- when the constructor is called, it prints out a message: "DebugStack *XYZ* is initialized"
  (where "XYZ" is the value of the name string) and
- DebugStack rewrites the methods "push" and "pop" such that they print out debug messages, too:
  - "String *blabla* is inserted into the DebugStack *XYZ*"
  - "String *blabla* is removed from the DebugStack *XYZ*"

Then modify your test program (from Exercise 4) such that it applies two DebugStack Objects.

Deadline: 09.04.2014

Exercise 6

Deadline: 09.04.2014

Karoly.Bosa@jku.at

1. Improve your stack implementation:

- Arrange the different Stack classes into a *stack* package and their test programs into a *test* package

- Create an interface *Stack* with
  - push(String s), pop(), isEmpty(), toString() and
  - clear() removes all values from the stack
  - exch() exchange the first and the second elements in the stack.
  - peek() returns with the first/top element of the stack, but does not remove it.

- Create abstract class *AbstractStack* implements interface *Stack*
  - Leave push, pop, toString and isEmpty as abstract methods
  - Implement clear, exch and peek with calling abstract methods

Exercise 6 (continuation)

Deadline: 09.04.2014

Karoly.Bosa@jku.at

2. Rename the initial stack class (which is called Stack as well and which was implemented first in Exercise 4.) to BoundedStack, then:

- Extends AbstractStack in BoundedStack
- Inherits the new methods from AbstractStack
- Define exception classes (in the *stack* package) and throw them in case of empty/full stack in the corresponding methods
- In the test program, try out and check the new methods (use the interface Stack as an expected argument type)
- In the test program, create errors (e.g.: pop from an empty stack, or push into a full stack) and handle the exceptions

Exercise 6 (continuation 2) Deadline: 09.04.2014

Karoly.Bosa@jku.at

3. Reimplement DebugStack with delegation instead of inheritance (expect a class which implements the interface Stack):

- Constructor: DebugStack(String name, Stack delegate)
- Store the reference for the delegate in a private field
- All operations are performed on the delegate, but with the debugging-versions of the methods
- Take care! DebugStack does not have any super class anymore (no inheritance), rather you should implement new methods with delegation (its methods work with the data field *delegate* given as the second argument of the constructor).

Test the newly implemented BoundedStack and DebugStack classes in separate test programs.