Object Oriented Programming

Java

Part 10: Packages, exceptions and assertions

Packages – revision (1)

- Packages are a mechanism to group information:
  - Packages may contain other packages, classes, interfaces and objects.
  - The package defines a **namespace**, so its members need to have unique identifiers (for instance, in a package it cannot exist two classes with the same name).
  - The identifier of a package may consist in a **simple name** or in a **qualified name**. The qualified name corresponds to the simple name prefixed with the name of the package where it resides, if it exists. It is common to use `::` to split the simple names.

Packages – revision (2)

- An **import** adds the content of the imported package to the namespace of the importing package, so that the members of the imported package need not to be used by their qualified name.
- Importing is not transitive.
  - If package B imports package A, and package C imports package B, in package C the members of A are not imported.
  - If package C also wants to import the members of A, two imports are needed, one to import package A and other to import package B.

- The set of methods of a package is referred as **API** (*Application Programmer Interface*).

Packages (1)

- Packages are useful for several reasons:
  - Group related interfaces and classes in the same package (which can then be stored in a jar file, together with the MANIFEST.MF describing the package).
  - Create a namespace that help avoid naming conflict between types defined inside and outside the package (possible use of popular names, e.g., List).
  - Provide a protected domain for application development (code within a package can cooperate using access to members of the classes and interfaces of the package that are unavailable to external code).

Packages (2)

- A class is inserted in a package with

```
package IdPacote;
```

- A `package` declaration must appear first in a source file, before any class or interface declaration (and before any `import`).
- Only one `package` declaration can appear in a source file.
- The package name is implicitly prefixed to each type name contained within the package.

Packages (3)

- If a type is not declared as being part of an explicit package, it is placed in an **unammed package** (ease the implementation of small programs).

Packages (4)

- Type import are preformed as:

```
import IdPacote[.IdSubPacote]*.(*|IdTipo);
```

- **Type import on demand**:

```
import java.util.*;
```

The * imports all public types in the corresponding package.

- **Single type import**:

```
import java.util.Set;
```

Packages (5)

- The `import` statement should be used after the declaration of the `package`, but before the declaration of the type.
- The package `java.lang` is automatically imported by Java (subpackage `lang` of the package `java`).
  - The separator "." in Java corresponds to the "/" in Unix and "\" in Windows.

Packages (6)

- Code in a package, which needs types defined outside that package, has two options:
  - Use the qualified name of the type.
  - Import part or all the package.

```
package xpto;
public class Xpto {
     java.util.Set<String> strings;
     //...
}
```

```
package xpto;
import java.util.Set;
public class Xpto {
     Set<String> strings;
     //...
}
```

Packages (7)

- The `import` statement simply tells the compiler how it can determine the fully qualified name for a type that is used in the program if it cannot find that type locally.
- The compiler will search for the type in the following order:
  - The current type including inherited types.
  - A nested type of the current type.
  - Explicitly named imported types (single type import).
  - Other types declared in the same package.
  - Implicitly named imported types (import on demand).
- If after all these steps the type is still not found it is an error.

Packages (8)

- There are only two options of visibility for classes and interfaces (non-nested ones) in a package: package and **public**.
  - A public class or interface is available on code outside the package.
  - By default a class or interface is accessible only in code within the same package.
    - The types are hidden out of the package.
    - The types are hidden for subpackages.

Packages (9)

- By default, a member of a class is visible within the corresponding package, and only inside this.
- Members of a class not declared **private** in a package are visible throughout the package.
- All members of an interface are implicitly **public**.

Packages (10)

- A method can only be redefined in a subclass if it is accessible (from the superclass).
- When a method is invoked, the runtime system must consider the accessibility of the method to decide which implementation to use ...

Packages (11)

```
package p1;

public abstract class AbstractSuperClass {
     private void pri() {print("AbstractSuperClass.pri()");}
     void pac() {print("AbstractSuperClass.pac()");}
     protected void pro() {print("AbstractSuperClass.pro()");}
     public void pub() {print("AbstractSuperClass.pub()");}
     public final void print() {
          pri();
          pac();
          pro();
          pub();
     }
}
```

Packages (12)

```
package p2;
import p1.AbstractSuperClass;

public class SubClass1 extends AbstractSuperClass {
     public void pri() {print(“SubClass1.pri()”);}
     public void pac() {print(“SubClass1.pac()”);}
     public void pro() {print(“SubClass1.pro()”);}
     public void pub() {print(“SubClass1.pub()”);}
}
```

Invoking `new SubClass1().print();`

prints in the terminal

```
AbstractSuperClass.pri()
AbstractSuperClass.pac()
SubClasse1.pro()
SubClasse1.pub()
```

Packages (13)

```
package p1;
import p2.SubClassa1;

public class SubClass2 extends SubClass1 {
    public void pri() {print("SubClass2.pri()");}
    public void pac() {print("SubClass2.pac()");}
    public void pro() {print("SubClass2.pro()");}
    public void pub() {print("SubClass2.pub()");}
}
```

Invoking `new SubClass2().print();`

prints in the terminal

```
AbstractSuperClass.pri()
SubClasse2.pac()
SubClasse2.pro()
SubClasse2.pub()
```

Packages (14)

```
package p3;
import p1.SubClass2;

public class SubClass3 extends SubClass2 {
     public void pri() {print("SubClass3.pri()");}
     public void pac() {print("SubClass3.pac()");}
     public void pro() {print("SubClass3.pro()");}
     public void pub() {print("SubClass3.pub()");}
}
```

Invoking `new SubClass3().print();`

prints in the terminal

```
AbstractSuperClass.pri()
SubClass3.pac()
SubClass3.pro()
SubClass3.pub()
```

Exceptions – revision (1)

- Frequently, applications are subject to many kind of errors:
  - Mathematic errors (for instance, divide by 0 arithmetic).
  - Invalid data format (for instance, integer with invalid characters).
  - Attempt to access a null reference.
  - Open an unexisting file.
  - …

Exceptions – revision (2)

- An **exception** is a signal that is thrown when an unexpected error is encountered.
  - The signal is synchronous if it occurs directly as a consequence of a particular user instruction.
  - Otherwise, it is asynchronous.
- Exceptions can be handled in different ways:
  - Terminating abruptly execution, with warning messages and printing useful information (unacceptable in critical systems).
  - Being managed in specific places, denominated **handlers**.

Handling exceptions (1)

- In Java, exceptions are objects of type **`Exception`**, which extends **`Throwable`**.
  - Exceptions are primarily **checked exceptions**, meaning that the compiler checks the exceptions a method declared to throw.
    - Checked exceptions represent conditions that, although exceptional, can reasonably be expected to occur.
  - The standard runtime exceptions and errors extend one of the classes **`RuntimeException`** and **`Error`**, making them **unchecked exceptions**.
    - Unchecked runtime exceptions represent conditions that generally reflect errors in the program's logic and cannot be reasonably recovered from at run time (e.g. **`NullPointerException`**, **`ArrayIndexOutOfBound`**).

Handling exceptions (2)

- Exceptions are catch by enclosing code in **`try`** blocks with either at least one **`catch`** clause or the **`finally`** clause.
- The basic syntax is:

```
try {
     statements:
     if (test) throw new MyException(String);
} catch(IdExceptiono1 e) {
     statements
} catch (IdException2 e) {
     statements
//... As many catches as needed
} finally {
     statements
}
```

Handling exceptions (3)

- A `try` clause must have at least one `catch`, or a `finally`.
- Any number of `catch` clauses, including zero, can be associated with a particular `try` as long as each clause catches a different type of exception.
- **The body of the `try` statement is executed until either an exception is thrown or the body finishes successfully.**

Handling exceptions (4)

- **If an exception is thrown, each `catch` clause is examined in turn, from first to last, to see whether the type of the exception object is assignable to the type declared in the `catch` .**
  - **When an assignable `catch` is found, its block is executed.** No other `catch` clause is executed.
  - **If no appropriate `catch` is found, the exception percolated out of the `try` statement into any outer `try` that might have a `catch` clause to handle it.**
    - If this clause is never found the program ends abruptly.

Handling exceptions (5)

- **It is not possible to put a superclass `catch` clause before a `catch` of one of its subclasses.**
  - The first clause would always catch the exception, and the second clause would never be reached.
  - This is a compile-time error.
- **If a `try` has the `finally` clause, its code is executed after the code in the `try` is completed** (even if an unexpected exception occurs, or a `return` or `break` is executed).

Handling exceptions (6)

- **Only one exception is handled in a `try` clause. If a `catch` or `finally` clause throw other exception, the `catch` clauses of the `try` are not reexamined.**
  - The `catch` and `finally` clauses are out of the protection of the respective `try` clause.
  - Such exceptions are passed from method to method in the program stack and they can (or cannot) be handled in a `try` clause of one of those methods.

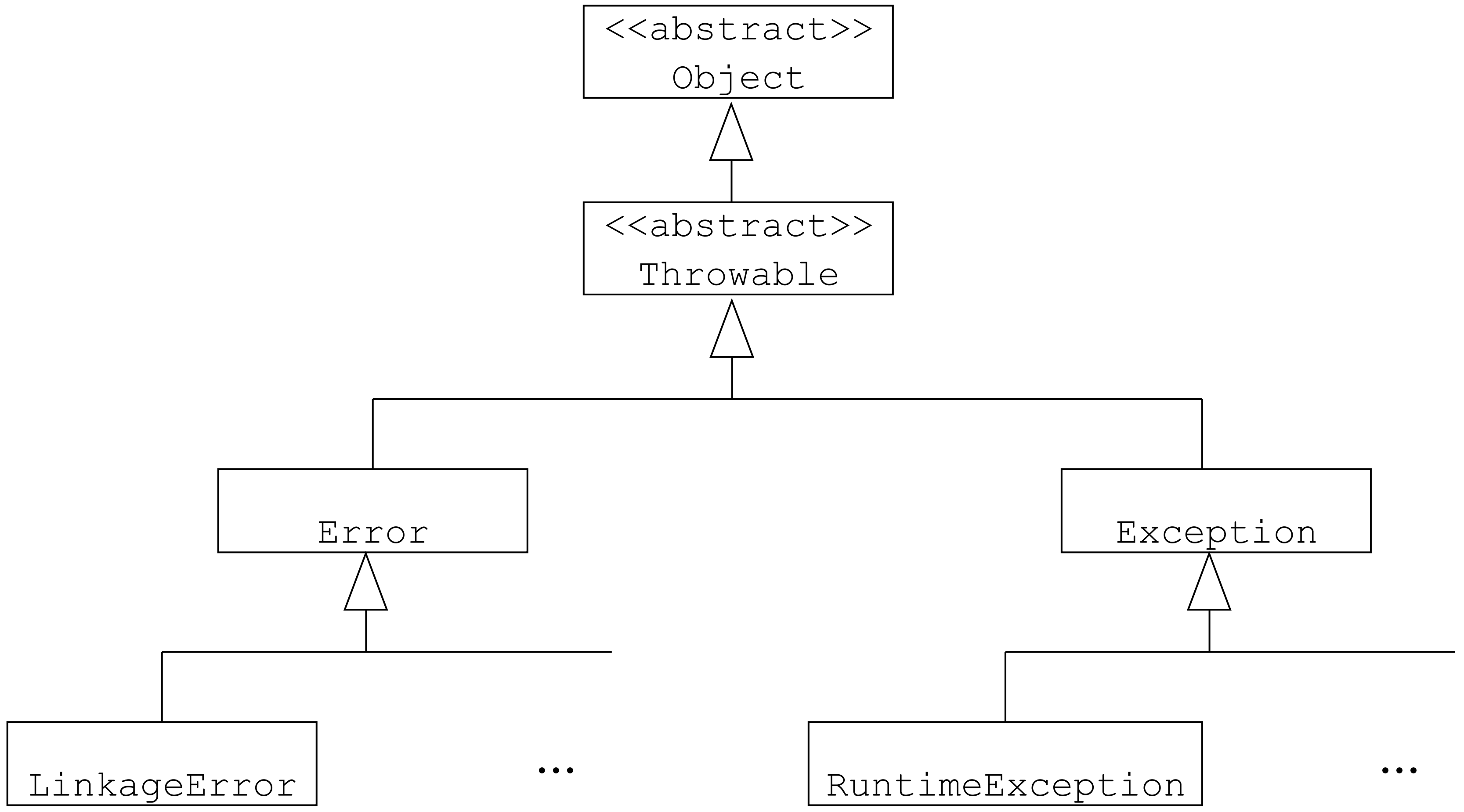
Handling exceptions (7)

- Usually, in the exception handlers:
  1. The error message is saved in a log.
  2. The object state is recovered.
  3. The method from where the exception was thrown is invoked.

Handling exceptions (8)

- If the program throws an exception, and in the code there is no `catch` clause to handle it, the JVM:
  1. Aborts the execution of the application.
  2. Prints in the `System.err` the exception thrown and the program stack.

Exception type hierarchy



`Throwable` class (1)

- Superclass of all errors and exceptions.
- Only objects of this class, or its subclasses, can be thrown in a `throw` statement and used as arguments of a `catch` clause.

`Throwable` class (2)

- Constructor:

`Throwable()`

Builds a new `Throwable` without detailed message.

`Throwable(String message)`

Builds a new `Throwable` with this detailed message.

`Throwable(String message, Throwable cause)`

Builds a new `Throwable` with this detailed message and cause.

`Throwable(Throwable cause)`

Builds a new `Throwable` with this cause and detailed message (`cause==null ? null : cause.toString()`).

`Throwable` class (3)

- Some methods:

`Throwable initCause(Throwable cause)`

Initialize the cause of this `Throwable` with the cause received as parameter. This method can be invoked only once, usually is invoked directly in the constructor, or immediately after its call. If this `Throwable` was built with `Throwable(Throwable)` or `Throwable(String,Throwable)` this method cannot be invoked.

`Throwable getCause()`

Returns the cause of this `Throwable`, or `null`.

`String getMessage()`

Returns the associated detailed message.

`void printStackTrace()`

Prints in the `System.err` the program stack.

`Error` class

- Used to throw exceptions on failure of the JVM.
- Usually are not handled by the programmer.

`Exception` class (1)

- J2SE provides 55 subclasses:
  - `ClassNotFoundException`
  - `IOException` (contains 21 subclasses)
  - ...
- Programmer exceptions are subclasses of `Exception`.

`Exception` class (2)

- Constructors:

`Exception()`
    Builds a new `Exception` without detailed message.
`Exception(String message)`
    Builds a new `Exception` with this detailed message.
`Exception(String message, Throwable cause)`
    Builds a new `Exception` with this detailed message and cause.
`Exception(Throwable cause)`
    Builds a new `Exception` with this cause and detailed message (`cause==null ? null : cause.toString()`).

- The `Exception` class does not provide new methods.

Example (1)

- Example of an user exception (division by zero):

```
public class DivisionByZero extends Exception {
     public DivisionByZero() {
          super(“Division by zero”);
     }
     public DivisionByZero(String message) {
          super(message);
     }
}
```

Example (2)

```
public class Divide {
     protected int op1;
     public Divide() { op1=20; }
     public int divide(int op2) {
          try {
               if (op2==0) throw new DivisonByZero();
               return op1/op2;
          } catch (DivisionByZero e) { // to avoid
               System.out.println(e);
               return -1;
          }
     }
}
```

Example (3)

```
public static void main (String args[]){
     if (args.length!=1) {
          System.out.println("Only one number!");
          System.out.exit(0);
     }
     Divide d = new Divide();
     int result = d.divide(Integer.parseInt(args[0],10));
     if (result==-1) System.exit(1);
     else {
          System.out.println(d.val()+"/"+args[0]+"="+result);
          System.exit(0);
     }
}
```

Exception chaining (1)

- Usually it is of interest the exception to be handled by the object that invoked the method (usually the recovery depends on the object that calls it).
- The method, where the exception may be thrown, should indicate in the header **`throws` IdException**
- In the example of division by zero, the method divide should be updated to:

```
public int divide(int op2) throws DivisionByZero{
     if (op2==0) throw new DivisionByZero();
     return op1/op2;
}
```

Exception chaining (2)

```
public static void main(String args[]) {
     if (args.length!=1) {
          System.out.println("Only one number!");
          System.out.exit(0);
     }
     int result;
     Divide d = new Divide();
     try {
          result = d.divide(Integer.parseInt(args[0],10));
          System.out.println(d.op1()+"/"+args[0]+"="+result);
          System.exit(0);
     } catch (DivisionByZero e) {
          System.out.println(e);
          System.exit(1);
     }
}
```

Assertions – definition

- An **assertion** is used to verify an invariant.
- An **invariant** is a condition which should be always true.

Assertions (1)

Syntax:

```
assert expr1 [: expr2]
```

- expr1 is a `boolean` or `Boolean` expression.
  - When an `assert` is found in the code:
    - The expr1 is evaluated:
      - If expr1 is `true`, the assertion passes.
      - If expr1 is `false`, the assertion fails, and an `AssertionError` is built and thrown.

Assertions (2)

- expr2 is an optional expression that is passed to the `AssertionError` to describe the problem found.
  - If expr2 is a `Throwable` it would be the value returned by the `getCause` of the corresponding `AssertionError`.
  - Otherwise, expr2 is converted into a `String` and it would be the detailed message of the corresponding `AssertionError`.

Assertions (3)

- Assertions should not be used to test malfunctions that sometimes happen (`IOException`, `NullPointerException`, ...).
- **The assertions should be used only to test conditions that should never fail in a correct program.** For instance:
  - Test whether the current state of an object is correct.
  - Verify the code flow.

Assertions (4)

- Test whether the current state of an object is correct: when removing an object from a set of objects if the object was found and removed the set is supposed to be one less element.

```
public boolean remove(Object object) {
    assert nbElements >= 0;
    if (object==null)
        throw NullPointerException(“remove: object null”);
    int aux = nbElements;
    boolean found = false;
    try {
        //remove object from the set (if it exists)
        return found;
     } finally {
        assert ( (found==false && nbElements==aux) ||
                 (found==true  && nbElemnets==aux-1));
     }
}
```

Assertions (5)

- Verify the code flow: in this case, one wants to replace the `old` value from the `values` array by the `new` value, assuring that the `old` value is always present in `values` array.

```
private void replace(int old, int new) {
     for (int i=0; i<values; i++) {
          if (values[i] == old) {
               values[i] = new;
               return;
          }
     }
     assert false : “replace: value “+old+” not found”;
}
```

Assertions (6)

- By default, the assertions are not evaluated.
  - It is possible to turn on and off assertions (for packages and classes).
  - When the assertions are off they are not evaluated, so assertion expression should not contain side effects.

```
assert ++i < max;
```

```
i++;
assert i < max;
```

Assertions (7)

- To turn on and off assertions:
  - Run an application in the shell with the following options: (in NetBeans Project properties > Run > VM Options):
    - -ea[:IdPackage][.IdSubPackage]*[.IdClass]
      Turn on the assertions to the given packages/classes. If no package or class is given assertions are turned on for all classes.
    - -da[:IdPackage][.IdSubPackage]*[.IdClass]
      Turn off the assertions to the given packages/classes. If no package or class is given assertions are turned off for all classes.
  - In the case of packages IdPackage... Can be used to apply to the package and all its subpackages.

Assertions (8)

- Attempting to completely remove the assertions from the source code:

```
private static final assertionsOn = false;

if (assertionsOn)
      assert i < max;
```

Assertions (9)

- Making assertions mandatory (deliberate use of side effects in the assertion):

```
static {
    boolean assertionsOn = false;
    assert assertionsOn = true;
    if (!assertionsOn)
        throw new IllegalStateException(“Assertions needed!”);
}
```