Course of Programming in Java

by Łukasz Stafiniak

Email: lukstafi@gmail.com, lukstafi@ii.uni.wroc.pl
Web: www.ii.uni.wroc.pl/~lukstafi

The Java Programming Language

Chapter 12: Exceptions and Assertions

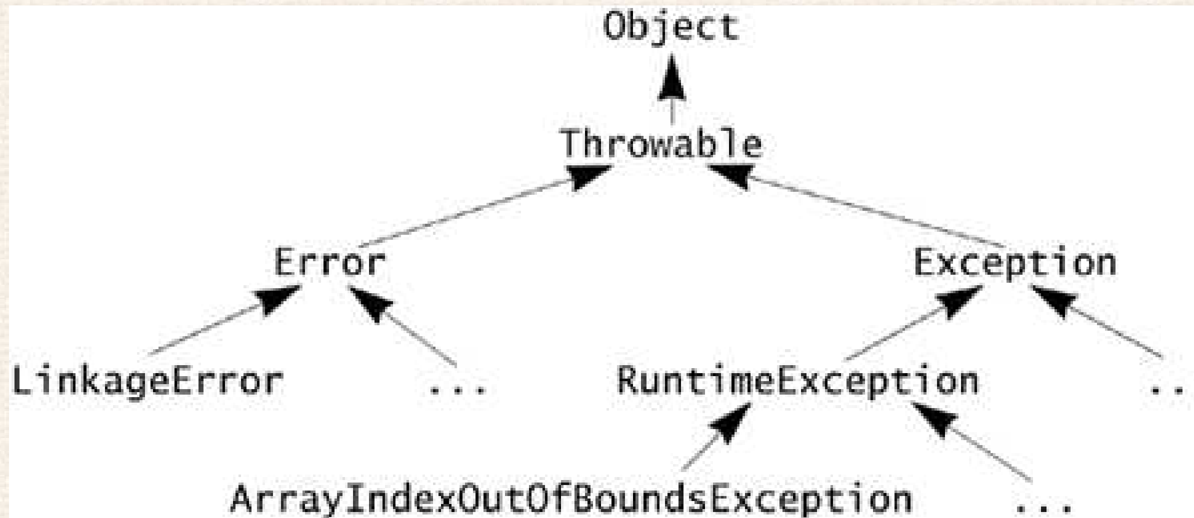
Chapter 18: Packages

Chapter 19: Documentation Comments

by Ken Arnold, James Gosling, David Holmes

Exceptions

- Exceptions are objects whose classes inherit from `Throwable`



- `Error` and `RuntimeException` exceptions are not checked by the compiler (*unchecked*), remaining exceptions need to be provided in the `throws` clause if they can arise but are not caught.

- For unchecked exceptions, you should be able to find some existing `RuntimeException`. For application exceptions, define checked ones.

```java
public class NoSuchAttributeException extends Exception {
    public final String attrName;

    public NoSuchAttributeException(String name) {
        super("No attribute named "" + name + "" found");
        attrName = name;
    }
}
```

...

```java
public void replaceValue(String name, Object newValue)
    throws NoSuchAttributeException
{
    Attr attr = find(name);             // look up the attr
    if (attr == null)                   // it isn't found
        throw new NoSuchAttributeException(name);
    attr.setValue(newValue);
}
```

- Execution is terminated immediately: no computations “to the right” of a computed `throw` expression are performed.
- Exceptions can be of a subclass of one of classes listed after `throws`.
- If you invoke a method that lists a checked exception in its `throws` clause, you can:
  - Catch the exception and handle it.
  - Catch the exception and map it into one of your exceptions by throwing an exception of a type declared in your own `throws` clause.
  - Declare the exception in your `throws` clause and let the exception pass through your method.
- En overriding or implementing method cannot declare more checked exceptions in the `throws` clause than the inherited method does.

```
Object value = new Integer(8);
try {
    attributedObj.replaceValue("Age", value);
} catch (NoSuchAttributeException e) {
    // shouldn't happen, but recover if it does
    Attr attr = new Attr(e.attrName, value);
    attributedObj.add(attr);
} catch (Exception e) {
    System.err.println ("Unexpected exn. in
replaceValue.");
    throw e;
}
```

- Multiple `catch` clauses are tried in order: always put a more specific exception in front of a more general one (a superclass).

```
public boolean searchFor(String file, String word)
    throws StreamException
{
    Stream input = null;
    try {
        input = new Stream(file);
        while (!input.eof())
            if (input.next().equals(word))
                return true;
        return false;        // not found
    } finally {
        if (input != null)
            input.close();
    }
}
```

- `finally` clause is run at *every* exit from the `try` block: can also be used to clean up for `break`, `continue`, and `return`

- `finally` "remembers" the reason it was reached: normal flow of execution, an exception, a `return`, `break` or `continue`, and proceeds with that reason after it finishes

  – but any transfer of control triggered inside the `finally` block supersedes the initial reason:

```
try {
    // ... do something ...
    return 1;
} finally {
    return 2;
}
```

```
public double[] getDataSet(String setName)
    throws BadDataSetException
{
    String file = setName + ".dset";
    FileInputStream in = null;
    try {
        in = new FileInputStream(file);
        return readDataSet(in);
    } catch (IOException e) {
        throw (BadDataSetException)
            new BadDataSetException().initCause(e);
    } finally {
        try {
            if (in != null)
                in.close();
        } catch (IOException e) {
            ;    // ignore: we either read the data OK
                 // or we're throwing BadDataSetException
        }
    }
}
// ... definition of readDataSet ...
```

(Exercise 12.2 in the book) Decide which way the following conditions should be communicated to the programmer:

- Someone tries to set the capacity of a `PassengerVehicle` object to a negative value.
- A syntax error is found in a configuration file that an object uses to set its initial state.
- A method that searches for a programmer-specified word in a string array cannot find any occurrence of the word.
- A file provided to an "open" method does not exist.
- A file provided to an "open" method exists, but security prevents the user from using it.
- During an attempt to open a network connection to a remote server process, the remote machine cannot be contacted.
- In the middle of a conversation with a remote server process, the network connection stops operating.

Assertions

```
public boolean remove(Object value) {
    assert count >= 0 : "count below zero";
    if (value == null)
        throw new NullPointerException("value");
    int orig = count;
    boolean foundIt = false;
    try {
        // remove element from list (if it's there)
        return foundIt;
    } finally {
        assert ((!foundIt && count == orig) ||
                count == orig - 1);
    }
}
```

Do not execute side-effects inside assert:
assertions are turned off by default. Turn on by passing:

`java -enableassertions` or `java -ea`.

Packages

- Groups of related interfaces and classes.
- Create namespaces that help avoid naming conflicts between types (allow popular names like `List` and `Constants`).
- Provide a protection domain for developing application frameworks (can restrict access "exported" to the outside world).
- At the top of a `.java` source file for package `attr`:

  ```
  package attr;
  ```

- Below the `package` declarations:

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

- Packages often named after Internet domains associated with the project:

  ```
  package com.magic.japan.attr;
  ```

- Recall the default – package – access level. Only accessible methods are overridden in subclasses (e.g. only `protected` and `public` from a different package).
- Nesting of packages does not provide additional functionality.

Multiple File Projects

- A compiled class `C` of nested package `a.b` should be in file *path*`/a/b/C.class`, where *path* is either the current directory, or the system variable `CLASSPATH` if set, or is given by the command-line option `-cp` *path* or `-classpath` *path* if provided.
  - Compiler looks for source files there also, or under `-sourcepath` if provided.
- Several alternative paths can be provided, separated by `;` under Windows and `:` under Linux.
- The compiler puts files in the same directory as the source file, unless passed option `-d` *cpath*, then puts a class `C` of nested package `a.b` into file *cpath*`/a/b/C.class`, etc.
- An inner class `N` of class `C` is put into its own file `N$C.class`.

Example Separating Source and Class Files: Windows

```
C:> dir
 classes\ lib\ src\
C:> dir src
 farewells\
C:> dir src\farewells
 Base.java GoodBye.java
C:> dir lib
 Banners.jar
C:> dir classes
C:> javac -sourcepath src -classpath classes;lib\Banners.jar\
      src\farewells\GoodBye.java -d classes
C:> dir classes
 farewells\
C:> dir classes\farewells
 Base.class GoodBye.class
```

Example Separating Source and Class Files: Linux

```
% ls classes/
 lib/ src/
% ls src
 farewells/
% ls src/farewells
 Base.java GoodBye.java
% ls lib
 Banners.jar
% ls classes
% javac -sourcepath src -classpath classes:lib/Banners.jar \
    src/farewells/GoodBye.java -d classes
% ls classes
 farewells/
% ls classes/farewells
 Base.class GoodBye.class
```

Java ARchive files

- The *Java Archive* (JAR) file format enables you to bundle multiple files into a single archive file. Typically a JAR file contains the class files and auxiliary resources associated with applets and applications.
- The files are compressed based on the ZIP format.
- The input-file(s) argument can contain the wildcard * symbol. Contents of directories are added to the archive recursively. *main-class* contains the public static void main(String[] args) method.
- jar cfe myapp.jar mypack.MyApp mypack / java -jar myapp.jar

Operation	Command
To create a JAR file	jar cfe *jar-file main-class input-file(s)*
To view the contents of a JAR file	jar tf *jar-file*
To extract the contents of a JAR file	jar xf *jar-file*
To extract specific files from a JAR file	jar xf *jar-file archived-file(s)*
To run an application packaged as a JAR file (requires the Main-class manifest header)	java -jar *app.jar*
To invoke an applet packaged as a JAR file	<applet code=AppletClassName.class archive="JarFileName.jar" width=width height=height > </applet>

Documentation Comments

```
/**

 * The first sentence of a javadoc comment should be a
good
 * summary of the identifier. The spaces and stars that
can
 * start a comment line are ignored.
 */
public void identifier() throws IntentUnknownException;
```

- The **javadoc** comments are the comments between /** and */.
- Standard HTML tags can be used.
  - To insert the character <, >, or & use <, >, or &.

    For @ at the beginning of a line, use @.
- Only doc comments that immediately precede a class, interface, method, or field are processed.

Javadoc Tags

- Block tags start with `@`, as in `@see` or `@deprecated` and mark special paragraphs, links to other documentation, etc.
- In-line tags `{@tag-name args}` can occur anywhere within a documentation comment and are used to apply special formatting, such as `{@code}`, or to produce special text, such as a hypertext link using `{@link}`.
- The `@see` tag creates a cross-reference link to other javadoc-documented identifier. Specify members of types by a # before the member name. Overloaded methods need their signatures (argument types).

```
@see #getName
@see Attr
@see com.magic.attr.Attr
@see com.magic.attr.Deck#DECK_SIZE
@see com.magic.attr.Attr#getName
@see com.magic.attr.Attr#Attr(String)
@see com.magic.attr.Attr#Attr(String, Object)
@see com.magic.attr
@see <a href="spec.html#attr">attribute Specification</a>
@see "The Java Developer's Almanac"
```

- `{@link package.class#member [label]}` works like `@see`, but embedded in text, e.g.

  ```
  Changes the value returned by calls to {@link
  #getValue}.
  ```

- The `@param` tag documents a single parameter to a method or constructor, or else a type parameter in a class, interface, or generic method:

  ```
  @param max    The maximum number of words to read.
  ```

  When documenting type parameters you should use < and > around the type parameter name:

  ```
  @param <E>  The element type of this List
  ```

- Documenting the `@return` value of a method.

- Equivalent: a method `@throws` an `@exception`:

```
@throws UnknownName   The name is unknown.
@throws java.io.IOException
            Reading the input stream failed; this exception
            is passed through from the input stream.
@throws NullPointerException
          The name is <code>null</code>.
```

- Code using a `@deprecated` (unfit for continued use) type, constructor, method, or field may generate a warning when compiled.

```
/**
 * Do what the invoker intends.
 *
 * @deprecated  You should use dwishm instead
 * @see         #dwishm
 */
@Deprecated // annotation is upper-case
public void dwim() throws IntentUnknownException;
```

  - using together with `@Deprecated` annotation guarantees the compiler warning

- You can specify as many `@author` paragraphs as you desire.

- Specify `@version` for a class or interface. The `@since` tag lets you specify at which versionthe tagged entity was added to your system.
- In `{@literal text}` the `text` is not interpreted as HTML source (can use can use &, <, and >).
- The `{@code text}` in-line tag behaves like `{@literal text}` except that `text` is printed in code font.
- The `{@value static-field-name}` tag is replaced by the actual value of the specified constant static field. (Syntax as in `@see`.)

```
The valid range is 0 to {@value java.lang.Short#MAX_VALUE}.
```

  - Value of constant static field being documented is just `{@value}`.

- The `{@docRoot}` tag will be replaced with a relative path to the top of the generated documentation tree.

```
Check out <a href="{@docroot}/license.html">our license</a>.
```

- Not supplying a comment for a subclass/subinterface or its overriding method, copies the comment from the superclass.
  - The `{@inheritDoc}` tag copies a documentation comment from the supertype – which you can refine providing more information.

```
/**
 * @return  {@inheritDoc}
 * This implementation never returns null.
 */
```

  - `@throws` comments are inherited only for exceptions still present in the `throws` clause.
- A `package-info.java` file should contain a single `package` statement, preceded by a doc comment – documentation for the overall package (part of the package summary page that is produced by `javadoc`).
  - Any `@see` or `{@link}` tag that names a language element must use the fully qualified form of the entity's name, even for classes and interfaces within the package itself.

Usage: `javadoc`

- `-d`: directory where the documentation will be placed
- `-sourcepath`: where to find sources (if other than current directory)
- `-subpackages`: run on subpackages recursively
- packages, or class files (e.g. `C.java`); can have wildcards `*`
- `-exclude`: do not process given packages

```
% javadoc -d /home/html -sourcepath /home/src -subpackages
     java -exclude java.net:java.lang
```