6 - Exceptions and Assertions

< Previous | Next >

§6.1 Assertions

- Assertions can be enabled or disabled for specific packages or classes. To specify a class, use the class name. To specify a package, use the package name followed by "..." (three dots): `java -ea:<class> myPackage.MyProgram` `java -da:<package>... myPackage.MyProgram`
- The command line switch that allows you to enable assertions for the system level classes (i.e. the classes that come bundled with the JDK) is `-esa` or `-enablesystemassertions`.
- Note that when you use a package name in the ea or da flag, the flag applies to that package as well as its subpackages.
- You can enable or disable assertions in the unnamed root (default)package (the one in the current directory) using the following commands: `java -ea:... myPackage.myProgram` `java -da:... myPackage.myProgram.
- Here assert is being used as an identifier (a method name is an identifier). However, beginning Java 1.4, assert has become a keyword. You cannot use a keyword as an identifier. Therefore, to use 'assert' as an identifier instead of a keyword, you have to tell the compiler that your code is 1.3 compliant. It will generate a warning but it will compile. To do so just use the -source option like this: `javac -source 1.3 Assertion.java`. Remember that you CANNOT use 'assert' as a keyword as well as an identifier at the same time.
- Assertions require changes at the API level, not at the JVM level. Besides the 'assert' keyword, new methods are added in java.lang.Class and java.lang.ClassLoader classes.
- An overriding method must not throw any new or broader checked exceptions than the ones declared in the overridden method. This means, the overriding method can only throw the exceptions or the subclasses of the exceptions declared in the overridden method. It can throw any subclass of Error or RuntimeException as well because it is not mandatory to declare Errors and RuntimeExceptions in the throws clause. An overriding method may also choose not to throw any exception at all.
- AssertionError is a subclass of Error.
- List.clear() returns void.

# The two valid ways of writing assertions are:

1. `assert <boolean_expression>;` In this style, the boolean expression must evaluate to true or false. If it evaluates to true, everything is fine. If it evaluates to false, a new AssertionError is thrown. Note that, can also be a method call that returns a boolean. The above line is thus equivalent to the following code: `if( !<boolean_expression> ) throw new AssertionError();`
2. `assert <boolean_expression> : <any_expression_but_void>;` This is same as the above except that if evaluates to false, is evaluated and its value is passed to the constructor of the AssertionError object. should evaluate to any object or a primitive. Thus, it cannot be a method call that returns void. **The second operand can be null but it cannot be void.** Note that if evaluates to true, is not evaluated. The above line is thus equivalent to the following code: `if( !<boolean_expression> ) throw new AssertionError(<any_expression_but_void>);`

# Valid uses of `assert`:

- `assert false : "Message";`
- `assert c.isEmpty();`

# Invalid uses of `assert`:

- `assert obj != null : throw new AssertionError();` Second operand must evaluate to an object or a primitive. (It can also be null)
- `assert o1, o2 != null;` --> `assert (o1 != null && o2!= null );`
- `assert;` assert must be followed by at least one operand.
- `assert null;` First operand must be a boolean. `assert <someboolean> : null;` is valid.

# Basic rules which govern where assertions should be used and where they should not be used.

- Assertions should be used for:
  1. Validating input parameters of a private method.(But NOT for public methods. public methods should throw regular exceptions when passed bad parameters.)
  2. Anywhere in the program to ensure the validity of a fact which is almost certainly true.
  3. Validating post conditions at the end of any method. This means, after executing the business logic, you can use assertions to ensure that the internal state of your variables or results is consistent with what you expect. For example, a method that opens a socket or a file can use an assertion at the end to ensure that the socket or the file is indeed opened.

- Assertions should not be used for:
  1. Validating input parameters of a public method. Since assertions may not always be executed, the regular exception mechanism should be used.
  2. Validating constraints on something that is input by the user. Same as above.
  3. Should not be used for side effects.

§6.2 Exceptions

- **Remember that most of the I/O operations (such as opening a stream on a file, reading or writing from/to a file) throw IOException.** Handle or declare rule!
- You cannot include classes that are related by inheritance in the same multi-catch block.
- You can combine `IOException|RuntimeException` in a multi-catch block.
- The exception parameter in a multi-catch clause is implicitly final. Thus, it cannot be reassigned.
- Within a multi-catch block, you can use only those methods of the exception which are common to all the exceptions that you are catching in the clause. In other words, you can use only the methods that are available in the most specific class that is a super class of all the Exceptions.

- You can add a catch or finally block to a try-with-resources block!
- Catch and finally blocks are executed after the resource opened in try-with-resources is closed.
- Both ClassNotFoundException and NoSuchFieldException, are checked exceptions and are thrown when you use Java reflection mechanism to load a class and access its fields.

# try-with-resources:

1. The resource class must implement java.lang.AutoCloseable interface. Many standard JDK classes such as BufferedReader, BufferedWriter implement java.io.Closeable interface, which extends java.lang.AutoCloseable.
2. Resources are closed at the end of the try block and before any catch or finally block.
3. Resources are not even accessible in the catch or finally block. For example:
   `try (Device d = new Device()) { d.read(); } finally { d.close(); //This will not compile because d is not accessible here.`
4. Resources are closed in the reverse order of their creation.
5. Resources are closed even if the code in the try block throws an exception.
6. java.lang.AutoCloseable's close() throws Exception but java.io.Closeable's close() throws IOException.
7. If an exception is thrown within the try-with-resources block, then that is the exception that the caller gets. But before the try block returns, the resource's close() method is called and if the close() method throws an exception as well, then this exception is added to the original exception as a supressed exception.
8. The auto-closeable variables defined in the try-with-resources statement are implicitly final. Thus, they cannot be reassigned.
9. In a try-with-resources statement the type of the resource must be specified in the try itself. `try(stmt = c.createStatement())` should be:
   `try(Statement stmt = c.createStatement())`

< Previous | Next >