
Chapter 3

# Assertions and Java Exceptions

Business Training

1

1Z0-809 Chapter 3

## Content

- Test Invariants by Using Assertions
- Use try-catch and throw Statements
- Use catch, Multi-catch, and finally Clauses
- Use Autoclose Resources with a try-with-resources Statement



2

# Test Invariants by Using Assertions

3



## Assertions Overview

- **Assertions let you test your assumptions during development, but the assertion code basically evaporates when the program is deployed, leaving behind no overhead or debugging code to track down and remove.**
  - Without using an assertion

```
private void methodA(int num) {
  if (num >= 0) {
    useNum(num + x);
  } else {  // num must be < 0
            // This code should never be reached!
    System.out.println("Yikes! num is a negative number! "
                       + num);
  }
}
```

  With an assertion !

```
private void methodA(int num) {
  assert (num>=0);   // throws an AssertionError
                     // if this test isn't true
  useNum(num + x);
}
```

  - The following code lists legal and illegal expressions for both parts of an assert statement.

```
void noReturn { }
int aReturn() { return 1; }
void go() {
  int x = 1;   boolean b = true;
  // the following six are legal assert statements
  assert (x == 1);      assert(b);      assert true;     assert(x == 1) : x;   assert(x == 1) : aReturn();
  assert(x == 1) : new ValidAssert();
  // the following six are ILLEGAL assert statements
  assert(x = 1);  // none of these are booleans
  assert(x);   assert 0;   assert(x == 1) : ;          // none of these return a value
  assert(x == 1) : noReturn();   assert(x == 1) : ValidAssert va;
}
```



4

# Enabling Assertions

- **Identifier vs. Keyword**
  - Prior to version 1.4, assert is used as an identifier. That's not a problem prior to 1.4.
- **Use Version 5 of java and javac**
  - As far as the exam is concerned, you'll ALWAYS be using version 5 of the Java compiler (javac), and version 5 of the Java application launcher (java).
- **Compiling Assertion-Aware Code**

| Command Line | If assert Is an Identifier | If assert Is a Keyword |
|---|---|---|
| `javac -source 1.3 TestAsserts .java` | Code compiles with warnings. | Compilation fails. |
| `javac -source 1.4 TestAsserts.java` | Compilation fails. | Code compiles. |
| `javac -source 1.5 TeotAsserts.Java` | Compilation fails. | Code compiles. |
| `javac -source 5 TestAsserts.java` | Compilation fails. | Code compiles. |
| `javac TestAsserts.java` | Compilation fails. | Code compiles. |

- **Running with Assertions**
  - You enable assertions at runtime with: java **-ea** com.insat.TestClass
  - You disable assertions at runtime with: java **-da** com.insat.TestClass
  - Remember, assertions are disabled by default.



5



# Using Assertions Appropriately

- **Not all legal uses of assertions are considered appropriate.**
- **Don't Use Assertions to Validate Arguments to a Public Method**
  - Because public methods are part of your interface to the outside world, you're supposed to guarantee that any constraints on the arguments will be enforced by the method itself. But since assertions aren't guaranteed to actually run (they're typically disabled in a deployed application), the enforcement won't happen if assertions aren't enabled.
- **Do Use Assertions to Validate Arguments to a Private Method**
  - When you assume that the logic in code calling your private method is correct, you can test that assumption with an assertion.
- **Don't Use Assertions to Validate Command-Line Arguments**
- **Do Use Assertions, Even in Public Methods, to Check for Cases that You Know are Never, Ever Supposed to Happen**
- **Don't Use Assert Expressions that can Cause Side Effects!**
  - assert expressions aren't guaranteed to always run, so you don't want your code to behave differently depending on whether assertions are enabled.



6

# Use try-catch and throw Statements

7

## Catching an Exception Using Try and Catch



- **Here's how it looks in pseudocode:**

```
1. try {
2.   // This is the first line of the "guarded region"
3.   // that is governed by the try keyword.
4.   // Put code here that might cause some kind of exception.
5.   // We may have many code lines here or just one.
6. }
7. catch(MyFirstException) {
8.   // Put code here that handles this exception.

9.   // This is the next line of the exception handler.
10.  // This is the last line of the exception handler.
11. }
12. catch(MySecondException) {
13.  // Put code here that handles this exception
14. }
15.
16. // Some other unguarded (normal, non-risky) code begins here
```



8

# Using Finally

- **A finally block encloses code that is always executed at some point after the try block, whether an exception was thrown or not.**
  - Even if there is a return statement in the try block, the finally block executes right after the return statement is encountered, and before the return executes!
- **The following legal code demonstrates a try, catch, and finally:**

```
try {
  // do stuff
} catch (SomeException ex) {
  // do exception handling
} finally {
  // clean up
}
```

  - Exam Watch
    - It is illegal to use a try clause without either a catch clause or a finally clause.

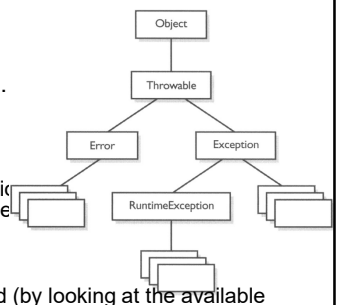






# Defining Exceptions

- **Exception Hierarchy**
  - All exception classes are subtypes of class **Exception.** This class derives from the class Throwable (which derives from the class Object).
    - Java provides many exception classes, most of which have quite descriptive names.
    - There are two ways to get information about an exception. The first is from the type of the exceptio itself. The next is from information that you can ge from the exception object.




- **Exception Matching**
  - When an exception is thrown, Java will try to find (by looking at the available catch clauses from the top down) a catch clause for the exception type. If it doesn't find one, it will search for a handler for a supertype of the exception.
  - The following will not compile:

```
try {
  // do risky IO things
} catch (IOException e) {
  // handle general IOExceptions
} catch (FileNotFoundException ex) {
  // handle just FileNotFoundException
}
```

You'll get a compiler error something like this:

TestEx.java:15: exception java.io.FileNotFoundException has already been caught

## Exception Declaration and the Public Interface



- **The throws keyword is used as follows to list the exceptions that a method can throw:**

```
void myFunction() throws  MyException1, MyException2 {
 // code for the method here
}
```

  - Remember this:
    - Each method must either handle all checked exceptions by supplying a catch clause or list each unhandled checked exception as a thrown exception.
    - This rule is referred to as Java's "**handle or declare**" requirement. (Sometimes called "**catch or declare**.")

- **Rethrowing the Same Exception**

```
public void doStuff() throws IOException {
 try {
   // risky IO things
 } catch( IOException ex) {
   // can't handle it
    throw ex;
 }
}
```



11

## Use the try Statement with multi-catch and finally Clauses



- **Java 7 made handling this sort of situation nice and easy with a feature called multi-catch:**
  - This avoids code duplication.

```
try {
  // access the database and write to a file
} catch (SQLException e) {
  handleErrorCase(e);
} catch (IOException e) {
  handleErrorCase(e);
}
```

```
try {
  // access the database and write to a file
} catch (SQLException | IOException e) {
  handleErrorCase(e);
}
```

  - Remember, multi-catch is only for exceptions in different inheritance hierarchies. To make sure this is clear, what do you think happens with the following code ?
    - `Catch(IOException | Exception e)` This Won't compile

- **Multi-catch and catch Parameter Assignment**
  - Don't assign a new value to the *catch* parameter. It is not good practice and creates confusing, hard-to-maintain code. But it is legal Java code to assign a new value to the *catch* block's parameter when there is only one type listed, and it will compile.
    - This code compiles

```
try {
  // access the database and write to a file
} catch (SQLException | IOException e) {
  e = new IOException();
}
```

    - This code does not copmile

```
try {
  // access the database and write to a file
} catch (SQLException | IOException e) {
  e = new IOException();
}
```



12

# Use Autoclose Resources with a try-with-resources Statement

13

## AutoCloseable Resources with a try-with-resources Statement

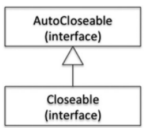


- **You can use a try-with-resources statement to simplify your code and auto-close resources.**

```
try (MyResource mr = MyResource.createResource();   // first resource
     MyThingy mt = mr.createThingy()) {             // second resource
  // do stuff
}
```

  - You can auto-close multiple resources within a try-with-resources statement. These resources need to be separated by semicolons in the try-with-resources statement header.
  - If a try block throws an exception, and a finally block also throws exception(s), then the exceptions thrown in the finally block will be added as suppressed exceptions to the exception that gets thrown out of the try block to the caller.The try-with-resources statement is logically calling a finally block to close the reader.

- **AutoCloseable and Closeable**
  - In order to use try-with-resource the resource should implement the AutoCloseable interface
    - AutoCloseable has only one method : close()
  - The Closeable interface was introduced in Java 5. When try-with-resources was invented in Java 7, the language designers wanted to change some things but needed backward compatibility with all existing code. So they created a superinterface with the rules they wanted.






14

# Autoclosing multiple resources

- Let's consider the following two resources

```
1: class One implements AutoCloseable {
2:   public void close() {
3:     System.out.println("Close - One");
4: } }
5: class Two implements AutoCloseable {
6:   public void close() {
7:     System.out.println("Close - Two");
8: } }
```

- The consider this main program

```
 9: class TryWithResources {
10:   public static void main(String[] args) {
11:     try (One one = new One(); Two two = new Two()) {
12:       System.out.println("Try");
13:       throw new RuntimeException();
14:     } catch (Exception e) {
15:       System.out.println("Catch");
16:     } finally {
17:       System.out.println("Finally");
18: } } }
```

- Running the preceding code will print:

```
Try
Close - Two
Close - One
Catch
Finally
```