Module 8

Exceptions and Assertions

Objectives

Upon completion of this module, you should be able to:

- Define exceptions
- Use `try`, `catch`, and `finally` statements
- Describe exception categories
- Identify common exceptions
- Develop programs to handle your own exceptions
- Use assertions
- Distinguish appropriate and inappropriate uses of assertions
- Enable assertions at runtime

This module covers the error handling facilities built into the Java programming language.

Relevance

Discussion – The following question is relevant to the material presented in this module:

- In most programming languages, how do you resolve runtime errors?

  _____

  _____

  _____

- If you make assumptions about the way your code works, and those assumptions are wrong, what might happen?

  _____

  _____

  _____

- Is it always necessary or desirable to expend CPU power testing assertions in production programs?

  _____

  _____

  _____

Exceptions and Assertions

Exceptions are a mechanism used by many programming languages to describe what to do when something unexpected happens. Typically, something unexpected is an error of some sort, for example when a method is invoked with unacceptable arguments, or a network connection fails, or the user asks to open a non-existent file.

Assertions are a way to test certain assumptions about the logic of a program. For example, if you believe that at a particular point the value of a variable will always be positive, then an assertion can test this. Assertions are used generally to test assumptions about local program logic inside a method, and usually not to test that external expectations are met.

One important feature of assertions is that they can be removed entirely from the code when it runs. This makes it possible for you to enable assertions during program development, but to not have the tests executed at runtime when the final product is delivered to a customer. This is an important difference between assertions and exceptions.

Exceptions

The Java programming language provides two broad categories of exceptions, known as *checked* and *unchecked* exceptions.

Checked exceptions are those that the programmer is expected to handle in the program, and that arise from external conditions that can readily occur in a working program. Examples would be a requested file not being found or a network failure.

Unchecked exceptions might arise from conditions that represent bugs, or situations that are considered generally too difficult for a program to handle reasonably. They are called *unchecked* because you are not required to check for them or do anything about them if they occur. Exceptions that arise from a category of situations that probably represent bugs are called *runtime exceptions*. An example of a runtime exception is attempting to access beyond the end of an array.

Exceptions that arise as a result of environmental issues that are rare enough or hard enough to recover from are called *errors*; an example of an error is running out of memory. Errors are also unchecked exceptions.

The `Exception` class is the base class that represents checked and unchecked exceptions. Rather than terminating the program, it is best if you write code to handle the exception and continue.

The `Error` class is the base class used for the unchecked, serious error conditions from which your program is not expected to attempt recovery. In most cases, you should let the program terminate when you encounter this type of error, although you might attempt to preserve as much of the user's work as possible.

The `RuntimeException` class is the base class that is used for the unchecked exceptions that might arise as a result of program bugs. In most cases, you should terminate the program when one of these arises. Again, attempting to preserve as much of the user's work as possible is a good idea.

When an exception occurs in your program, the method that finds the error can handle the exception itself or *throw* the exception back to its caller to signal that a problem has occurred. The calling method then has the same choices: handle the exception or throw it back to its caller. If an exception reaches the top of a thread, then that thread is killed. This scheme gives the programmer the option of writing a *handler* to deal with the exception at an appropriate point in the code.

You can determine the checked exceptions a method throws by browsing the API. Sometimes you might also find documentation describing one or more unchecked exceptions, but this is usually unnecessary, and you cannot rely on such information being listed.

Exception Example

Code 8-1 shows a program that adds all of the command-line arguments.

Code 8-1 Example Program that Throws an Exception

```
1   public class AddArguments {
2     public static void main(String args[]) {
3       int sum = 0;
4       for ( int i = 0; i < args.length; i++ ) {
5         sum += Integer.parseInt(args[i]);
6       }
7       System.out.println("Sum = " + sum);
8     }
9   }
```

This program works fine if all of the command-line arguments are integers. For example:

```
java AddArguments 1 2 3 4
Sum = 10
```

This program fails if any of the arguments are not integers:

```
java AddArguments 1 two 3.0 4
Exception in thread "main" java.lang.NumberFormatException:
For input string: "two"at
java.lang.NumberFormatException.forInputString(NumberFormat
Exception.java:48)
    at java.lang.Integer.parseInt(Integer.java:447)
    at java.lang.Integer.parseInt(Integer.java:497)
    at AddArguments.main(AddArguments.java:5)
```

The try-catch Statement

The Java programming language provides a mechanism for figuring out which exception was thrown and how to recover from it. Code 8-2 demonstrates a modified AddArguments program that uses a try-catch statement to catch the exception, in this case a java.lang.NumberFormatException.

Code 8-2 Example Program that Handles an Exception

```
1   public class AddArguments2 {
2     public static void main(String args[]) {
3       try {
4         int sum = 0;
5         for ( int i = 0; i < args.length; i++ ) {
6           sum += Integer.parseInt(args[i]);
7         }
8         System.out.println("Sum = " + sum);
9       } catch (NumberFormatException nfe) {
10        System.err.println("One of the command-line "
11                           + "arguments is not an integer.");
12      }
13    }
14  }
```

This program captures the exception and then quits with an error message:

```
java AddArguments2 1 two 3.0 4
One of the command-line arguments is not an integer.
```

Fine-grained Exception Handling

The try-catch statement can be used on smaller chunks of code. Code 8-3 shows the try-catch statement used inside of the for loop. This enables the program to discard only those command-line arguments that are not integers. This code demonstrates a more graceful degradation in behavior than the AddArguments2 program.

Code 8-3 A Refined AddArguments Program

```
1   public class AddArguments3 {
2     public static void main (String args[]) {
3       int sum = 0;
4       for ( int i = 0; i < args.length; i++ ) {
5         try {
6           sum += Integer.parseInt(args[i]);
7         } catch (NumberFormatException nfe) {
8           System.err.println("[" + args[i] + "] is not an integer"
9                              + " and will not be included in the sum.");
10        }
11      }
12      System.out.println("Sum = " + sum);
13    }
14  }
```

This program captures the exception for each non-integer, command-line argument and generates an warning message. However, this program proceeds to sum all of the valid integers:

```
java AddArguments3 1 two 3.0 4
[two] is not an integer and will not be included in the
sum.
[3.0] is not an integer and will not be included in the
sum.
Sum = 5
```

Using Multiple catch Clauses

There can be multiple catch blocks after a try block, each handling a different exception type. Code 8-4 illustrates the syntax of using multiple catch clauses.

Code 8-4 Example Using Multiple catch Clauses

```
try {
  // code that might throw one or more exceptions

} catch (MyException e1) {
  // code to execute if a MyException is thrown

} catch (MyOtherException e2) {
  // code to execute if a MyOtherException is thrown

} catch (Exception e3) {
  // code to execute if any other exception is thrown
}
```

If you have multiple catch clauses for a try block, the order of the catch clauses matters. That is because an exception thrown from the try block will be handled by the first catch clause that can handle it. If in this example the Exception catch clause is put first, then it would handle all exceptions, and the MyException or MyOtherException catch clauses would never be invoked.

Call Stack Mechanism

If a statement throws an exception, and that exception is not handled in the immediately enclosing method, then that exception is thrown to the calling method. If the exception is not handled in the calling method, it is thrown to the caller of that method. This process continues. If the exception is still not handled by the time it gets back to the main() method and main() does not handle it, the exception terminates the program abnormally.

Consider a case in which the main() method calls another method named first(), and this, in turn, calls another method named second(). If an exception occurs in second() and is not handled there, it is thrown back to first(). Imagine that in first() there is a catch for that type of exception. In this case the exception is handled and goes no further. However, if first() does not have any catch for this type of exception then the next method in the call stack, main(), is checked. If the exception is not handled in main(), then the exception is printed to the standard output and the program stops executing.

The finally Clause

The finally clause defines a block of code that *always* executes, regardless of whether an exception was caught. The following sample code and description is taken from the white paper, *Low Level Security in Java*, by Frank Yellin:

```
1   try {
2     startFaucet();
3     waterLawn();
4   } catch (BrokenPipeException e) {
5     logProblem(e);
6   } finally {
7     stopFaucet();
8   }
```

In the previous example, the faucet is turned off regardless of whether an exception occurs while starting the faucet or while watering the lawn. The code inside the braces after the try is called the *protected code*.

The only situations that prevent the finally clause executing are virtual machine shutdown (for example if the System.exit method is executed, or the machine is shut down or has its power turned off). This implies that the control flow can deviate from normal sequential execution. For example, if a return statement is embedded in the code inside the try block, the code in the finally clause executes before the return.

Exception Categories

Earlier in this module you were introduced to three broad categories of exceptions. This section examines the class hierarchy that represents those categories. The class `java.lang.Throwable` acts as the parent class for all objects that can be thrown and caught using the exception-handling mechanisms. Methods defined in the `Throwable` class retrieve the error message associated with the exception and print the stack trace showing where the exception occurred. There are three key subclasses of `Throwable`: `Error`, `RuntimeException`, and `Exception`, which are shown in Figure 8-1.

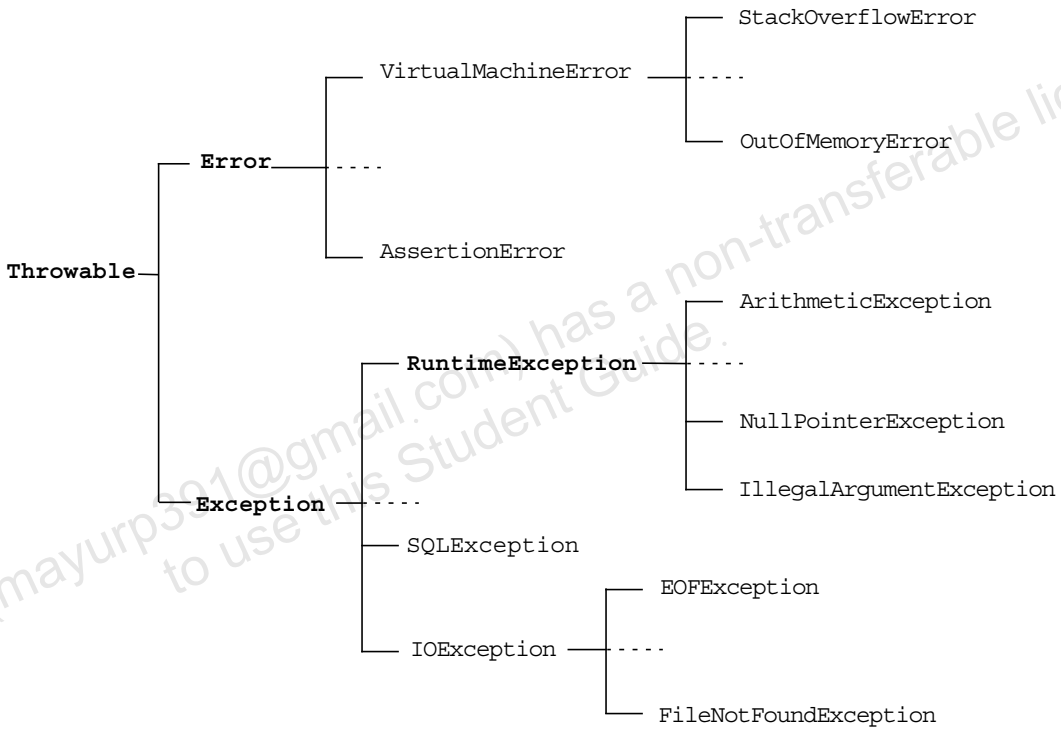

Figure 8-1 Subclasses and Exceptions

You should not use the `Throwable` class; instead, use one of the subclass exceptions to describe any particular exception. The following describes the purpose of each exception:

- The value `Error` indicates a severe problem from which recovery is difficult, if not impossible. An example is running out of memory. A program is not expected to handle such conditions.
- The value `RuntimeException` indicates a design or implementation problem. That is, it indicates conditions that should never happen if the program is operating properly. A `NullPointerException`, for example, should never be thrown if the developer remembers to associate reference variables with new or existing objects on the heap. Because a program that is designed and implemented correctly never issues this type of exception, it is usual to leave it unhandled. This results in a message at runtime, and ensures that action is taken to correct the problem, rather than hiding it where (you think) no one will notice.
- Other exceptions indicate a difficulty at runtime that is most often caused by environmental effects and can be handled. Examples include a file not found or invalid URL exceptions (the user typed a wrong URL), both of which could occur easily if the user mistyped something. Because these occur usually as a result of user error, you are encouraged to handle them.

Common Exceptions

The Java programming language provides several predefined exceptions. Some of the more common exceptions are:

- `NullPointerException`

  This is an attempt to access an attribute or method of an object using a variable that does not refer to an object, for example, when the variable has not been initialized or when no object has been instantiated.

  ```
  Employee emp = null;
  System.out.println( emp.getName() );
  ```

- `FileNotFoundException`

  This exception is an attempt to read from a file that does not exist.

- `NumberFormatException`

  This exception is an attempt to parse a string into a number (integer or floating point number) that has an illegal number format.

- `ArithmeticException`

  This is the result of a divide-by-zero operation for integers.

  ```
  int i = 0;
  int j = 12 / i;
  ```

- `SecurityException`

  Typically thrown in a browser, the `SecurityManager` class throws an exception for applets that attempt to perform any operation that might be dangerous to the host or its files. The following are examples of operations that can cause security exceptions:

  - Access the local file system.
  - Open a socket to a host that is not the same host that served the applet.
  - Execute another program in a runtime environment.

The Handle or Declare Rule

To encourage the writing of robust code, the Java programming language requires that if any checked exception (that is, a subclass of `Exception` but not a subclass of `RuntimeException`) might occur at any given point in the code, then the method that contains that point must define explicitly what action is to be taken if the problem arises.

You can do the following to satisfy this requirement:

- Handle the exception by using the `try-catch-finally` block.

  Have the enclosing method handle the exception by including a `try-catch` block around the possible problem. The `catch` clause must name either the expected exception class or a superclass. This counts as handling the situation, even if the `catch` clause is empty.

- Declare exceptions that a method can throw.

  Have the calling method indicate that it does not handle the exception, and that the exception is thrown back to *its* calling method. This is declaring the exception and clearly passes responsibility to declare or handle to the calling method.

  A method can declare that an exception might be thrown in the body of the method with a `throws` clause as follows:

  ```
  void trouble() throws IOException {...}
  ```

  Following the keyword `throws` is a list of all the exceptions that the method can throw back to its caller. Although only one exception is shown here, you can use a comma-separated list if this method throws multiple possible exceptions, like this:

  ```
  void trouble() throws IOException, OtherException {...}
  ```

You do not need to declare runtime exceptions or errors. All subclasses of `RuntimeException` and `Error` are called unchecked and do not need to be included in the `throws` clause of the method declaration.

You can choose to handle runtime exceptions. Usually, most runtime exceptions indicate a programming logic bug. It is most appropriate to discover these bugs during testing and fix them before the code is put into production. Therefore, there is no need to include catch clauses for runtime exceptions. There are a few notable exceptions to this rule. For example, catching a `NumberFormatException` when parsing a user-entered string is useful.

Whether you choose to handle or declare an exception depends on whether you consider yourself or your caller a more appropriate candidate for dealing with the exception.

Note – Because the exception classes are organized into hierarchies, the same as other classes, and because you can use a class whenever a subclass is expected, you can catch *groups* of exceptions and handle them with the same catch code. For example, although there are several different types of `IOExceptions` (`EOFException`, `FileNotFoundException`, and so on), by trapping `IOException` you can also catch instances of any subclass of `IOException`.

Method Overriding and Exceptions

When overriding a method that throws exceptions, the overriding method can declare only exceptions that are either the same class or a subclass of the exceptions. For example, if the superclass method throws an `IOException`, then the overriding method can throw an `IOException`, a `FileNotFoundException` (a subclass of `IOException`), but not an `Exception` (the superclass of `IOException`). You can declare fewer or more specific exceptions in the `throws` clause.

In the following example, three classes are declared: `TestA`, `TestB1`, and `TestB2`. `TestA` is the superclass of `TestB1` and `TestB2`.

```
1   public class TestA {
2     public void methodA() throws IOException {
3       // do some number crunching
4     }
5   }
```

```
1   public class TestB1 extends TestA {
2     public void methodA() throws EOFException {
3       // do some number crunching
4     }
5   }
```

```
1   public class TestB2 extends TestA {
2     public void methodA() throws Exception {
3       // do some number crunching
4     }
5   }
```


The class `TestB1` compiles because `EOFException` is a subclass of `IOException`. However, class `TestB2` fails to compile because `Exception` is a superclass of `IOException`.

It is permitted to declare that an overriding method throws fewer exceptions than the superclass method, including no exceptions at all. The new restricted set will become the limits on what might be thrown by any sub-subclasses that are created.

Creating Your Own Exceptions

User-defined exceptions are created by extending the `Exception` class. Exception classes contain anything that a *regular* class contains. The following is an example of a user-defined exception class containing a constructor, some variables, and methods.

```
1   public class ServerTimedOutException extends Exception {
2     private int port;
3
4     public ServerTimedOutException(String message, int port) {
5       super(message);
6       this.port = port;
7     }
8
9     public int getPort() {
10      return port;
11    }
12  }
```

Note – Use the `getMessage` method, inherited from the `Exception` class, to get the reason the exception was made

To throw an exception that you have created, use the following syntax:

```
throw new ServerTimedOutException("Could not connect", 80);
```

Always instantiate the exception on the same line on which you throw it, because the exception can carry line number information that will be added when the exception is created.

Throwing a User-Defined Exception

Consider a client-server program. In the client code, you try to connect to the server and expect the server to respond within five seconds. If the server does not respond, your code could throw an exception (such as a user-defined `ServerTimedOutException`) as follows:

```
1   public void connectMe(String serverName)
2          throws ServerTimedOutException {
3     boolean successful;
4     int portToConnect = 80;
5
6     successful = open(serverName, portToConnect);
7
8     if ( ! successful ) {
9       throw new ServerTimedOutException("Could not connect",
10                                        portToConnect);
11    }
12  }
```

Handling a User-Defined Exception

To catch your exception, use the `try` statement:

```
1     public void findServer() {
2       try {
3         connectMe(defaultServer);
4       } catch (ServerTimedOutException e) {
5         System.out.println("Server timed out, trying alternative");
6         try {
7           connectMe(alternativeServer);
8         } catch (ServerTimedOutException e1) {
9           System.out.println("Error: " + e1.getMessage() +
10                             " connecting to port " + e1.getPort());
11        }
12      }
13    }
```

Note – You can nest the `try` and `catch` blocks, as shown in the previous example.

You can also process an exception partially and then throw it; for example:

```
try {
  connectMe(defaultServer);
} catch (ServerTimedOutException e) {
  System.out.println("Error caught and rethrown");
  throw e;
}
```

Assertions

Two syntactic forms are permitted for assertion statements, these are:

```
assert <boolean_expression> ;
assert <boolean_expression> : <detail_expression> ;
```

In either case, if the Boolean expression evaluates to `false`, then an `AssertionError` is thrown. This error should not be caught, and the program should be terminated abnormally.

If the second form is used, then the second expression, which can be of any type, is converted to a `String` and used to supplement the message that prints when the assertion is reported.

Note – Because the `assert` keyword is relatively new, the assertion mechanism can break code that was written for the JDK version 1.3 or older if it uses the word `assert` as a variable name or label. You can use the `-source 1.3` option to tell the `javac` command to compile the code for the JDK 1.3 version.

Recommended Uses of Assertions

Assertions can provide valuable documentation about the programmer's assumptions and expectations. Because of this, assertions can be valuable especially during maintenance when other programmers are working on the code.

Assertions should be used generally to verify the internal logic of a single method or a small group of tightly coupled methods. Generally, assertions should not be used to verify that code is being used correctly, but rather that its own internal expectations are being met.

Examples of good uses of assertions are internal invariants, control flow invariants, postconditions, and class invariants.

Internal Invariants

An internal invariant exists when you believe that a situation is always or never the case and you code accordingly. For example consider this code:

```
1   if (x > 0) {
```

```
2     // do this
3   } else {
4     // do that
5   }
```

If you have made the assumption that `x` has the value `0`, and is never negative, and yet `x` turns out to be negative, then the code will do something unplanned. Almost certainly the behavior will not be correct. Worse, because the code does not stop and report the problem, you will not find out about the problem until much later when far more damage has been done. The original source of the problem then becomes very difficult to determine.

In this case, adding an assertion statement to the opening of the `else` block helps ensure the correctness of your assumption, and brings the bugs to light quickly if you are wrong, or if someone changes the code so as to invalidate the assumption during maintenance. With the assertion in place, the code would look like this:

```
1   if (x > 0) {
2     // do this
3   } else {
4     assert (x == 0);
5     // do that, unless x is negative
6   }
```

Control Flow Invariants

Control flow invariants describe assumptions similar to the assumptions of internal invariants, but they relate to the way the execution flows, rather than to the value or relationships of variables. For example, in a `switch` statement, you might believe that you have enumerated every possible value of the control variable, and that therefore a `default` statement would never be executed. This should be verified by adding an assertion statement like this:

```
1   switch (suit) {
2      case Suit.CLUBS: // ...
3        break;
4      case Suit.DIAMONDS: // ...
5        break;
6      case Suit.HEARTS: // ...
7        break;
8      case Suit.SPADES: // ...
9        break;
```

```
10    default: assert false : "Unknown playing card
suit";
11        break;
12  }
```

Postconditions and Class Invariants

Postconditions are assumptions about the value or relationships of variables at the completion of a method. A simple example of a postcondition test would be that after a pop method on a stack, the stack should contain one less element than when the method was called, unless the stack was already empty. This might be coded like this:

```
1   public Object pop() {
2     int size = this.getElementCount();
3     if (size == 0) {
4       throw new RuntimeException("Attempt to pop from empty stack");
5     }
6
7     Object result = /* code to retrieve the popped element */ ;
8
9     // test the postcondition
10    assert (this.getElementCount() == size - 1);
11
12    return result;
13  }
```

Notice here that if the pop method did not throw an exception if called on an empty stack, then it would be harder to express the assertion, because an original size of zero would still result in a zero final size. Also notice that the precondition test, that is the test that determines if the method is being called on an empty stack, does not use an assertion. This is because it is not an error in the local logic of the method if such a situation occurs, rather it is an error in the way the object is being used. Such tests on external behavior should, in general, not use assertions, but should use simple exceptions. This ensures that the test is always made and cannot be disabled as is the case with assertions.

A class invariant is something that can be tested at the end of every method call for the class. For the example of a stack class, an invariant condition is one in which the element count is never negative.

Inappropriate Uses of Assertions

Do not use assertions to check the parameters of a public method. Instead the method should test each parameter and thrown an appropriate exception, such as `IllegalArgumentException` or `NullPointerException`. The reason why you should not use assertions for parameter checking is because the assertion mechanism can be turned off, but you still want to perform parameter checking.

Do not use methods in the assertion check that can cause side-effects, because the assertion checks may be turned off at runtime. If your application depends upon these side-effects, then your application will exhibit different behavior when the assertion checks are turned off.

Controlling Runtime Evaluation of Assertions

One of the major advantages of assertions over using exceptions is that assertion checking can be disabled at runtime. If this is done, then there is no overhead involved in checking the assertions, and the code runs as fast as if the assertion test had never been present. This is better than using conditional compilation for two reasons. First, it does not require a different compile phase for production. Second, if required, the assertions can be re-enabled in the field to determine if the assertions have become invalid due to some unforeseen environmental effect.

By default, assertions are disabled. To turn assertions on, use either of these forms:

```
java -enableassertions MyProgram
java -ea MyProgram
```

Assertions can be controlled on packages, package hierarchies, or individual classes. For information on this, consult the documentation at: `docs/guide/language/assert.html` under the installation of your JDK software.