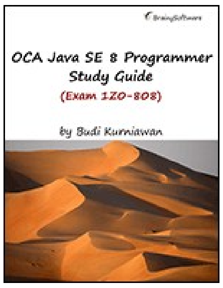

OCA Java SE 8 Programmer Study Guide (Exam 1Z0-808)

by Budi Kurniawan

Brainy Software Corp.. (c) 2015. Copying Prohibited.

Chapter 8: Error Handling

Overview

Error handling is an important feature in any programming language. A good error handling mechanism makes it easier for programmers to write robust applications and to prevent bugs from creeping in. In some languages, programmers are forced to use multiple **if** statements to detect all possible conditions that might lead to an error. This could make code excessively complex. In a larger program, this could easily lead to spaghetti like code.

Java offers the **try** statement as a nice approach to error handling. With this strategy, part of the code that could potentially lead to an error is isolated in a block. Should an error occur, this error is caught and resolved locally. This chapter teaches you this.

8.1 Catching Exceptions

There are two types of errors, compile error and runtime error. Compile errors or compilation errors are due to errors in the source code. For example, if you forgot to terminate a statement with a semicolon, the compiler will tell you that and refuse to compile your code. Compile errors are caught by the compiler at compile time. Runtime errors, on the other hand, can only be caught when the program is running, i.e. at runtime, because the compiler could not have caught them. For example, running out of memory is a runtime error and a compiler could not have predicted this. Or, if a program tries to parse a user input to an integer, the input is only available when the program is running. If the user enters non-digits, then the parsing process will fail and a runtime error thrown. A runtime error, if not handled, will cause the program to quit abruptly.

In your program you can isolate code that may cause a runtime error using a **try** statement, which normally is accompanied by the **catch** and **finally** statements. Such isolation typically occurs in a method body. If an error is encountered, Java stops the processing of the **try** block and jump to the **catch** block. Here you can gracefully handle the error or notify the user by 'throwing' a **java.lang.Exception** object. Another scenario is to re-throw the exception or a new **Exception** object back to the code that called the method. It is then up to the client how he or she would handle the error. If a thrown exception is not caught, the application will crash.

This is the syntax of the **try** statement.

```
try {
    [code that may throw an exception]
} [catch (ExceptionType-1 e) {
    [code that is executed when ExceptionType-1 is thrown]
}] [catch (ExceptionType-2 e) {
    [code that is executed when ExceptionType-2 is thrown]
}]
  …
} [catch (ExceptionType-n e) {
    [code that is executed when ExceptionType-n is thrown]
}]
[finally {
    [code that runs regardless of whether an exception was thrown]]
}]
```

The steps for error handling can be summarized as follows:

1. Isolate code that could lead to an error in the **try** block.
2. For each individual **catch** block, write code that is to be executed if an exception of that particular type occurs in the **try** block.
3. In the **finally** block, write code that will be run whether or not an error has occurred.

Note that the **catch** and **finally** blocks are optional, but one or both of them must exist. Therefore, you can have **try** with one or more **catch** blocks, **try** with **finally** or **try** with **catch** and **finally**.

The previous syntax shows that you can have more than one **catch** block. This is because some code may throw different types of exceptions. When an exception is thrown from a **try** block, control is passed to the first **catch** block. If the type of exception thrown matches the exception or is a subclass of the exception in the first **catch** block, the code in the **catch** block is executed and then control goes to the **finally** block, if one exists.

If the type of the exception thrown does not match the exception type in the first **catch** block, the JVM goes to the next **catch** block and does the same thing until it finds a match. If no match is found, the exception object will be thrown to the method caller. If the caller does not put the offending code that calls the method in a **try** block, the program will crash.

To illustrate the use of this error handling, consider the **NumberDoubler** class in Listing 8.1. When the class is run, it will prompt you for input. You can type anything, including non-digits. If your input is successfully converted to a number, it will double it and print the result. If your input is invalid, the program will print an "Invalid input" message.

Listing 8.1: The NumberDoubler class

```
package app08;
import java.util.Scanner;

public class NumberDoubler {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        String input = scanner.next();
        try {
            double number = Double.parseDouble(input);
            System.out.printf("Result: %s", number);
        } catch (NumberFormatException e) {
            System.out.println("Invalid input.");
        }
        scanner.close();
    }
}
```

The **NumberDoubler** class uses the **java.util.Scanner** class to take user input.

```
Scanner scanner = new Scanner(System.in);
String input = scanner.next();
```

It then uses the static **parseDouble** method of the **java.lang.Double** class to convert the string input to a **double**. Note that the code that calls **parseDouble** resides in a **try** block This is necessary because **parseDouble** may throw a **java.lang.NumberFormatException**, as indicated by the signature of the **parseDouble** method:

```
public static double parseDouble(String s)
        throws NumberFormatExcpetion
```

The **throws** statement in the method signature tells you that it may throw a **NumberFormatException** and it is the responsibility of the method caller to catch it.

Without the **try** block, invalid input will give you this embarrassing error message before the system crashes:

```
Exception in thread "main" java.lang.NumberFormatException:
```

8.2 try without catch

A try statement can be used with **finally** without a catch block. You normally use this syntax to ensure that some code always gets executed whether or not an unexpected exception has been thrown in the **try** block. For example, after opening a database connection, you want to make sure the connection's **close** method is called after you're done with it. To illustrate this scenario, consider the following pseudocode that opens a database connection.

```
Connection connection = null;
try {

    // open connection
    // do something with the connection and perform other tasks

} finally {
    if (connection != null) {
        // close connection
    }
}
```

If something unexpected occurs in the **try** block, the **close** method will always be called to release the resource.

8.3 Catching Multiple Exceptions

Java 7 and later allow you to catch multiple exceptions in a single **catch** block if the caught exceptions are to be handled by the same code. The syntax of the **catch** block is as follows, two exceptions being separated by the pipe character |.

```
catch(exception-1 | exception-2 … e) {

    // handle exceptions

}
```

For example, the **java.net.ServerSocket** class's **accept** method can throw four exceptions: **java.nio.channels.IllegalBlockingModeException, java.net.SocketTimeoutException, java.lang.SecurityException**, and **java.io.Exception**. If, say, the first three exceptions are to be handled by the same code, you can write your **try** block like this:

```
try {
    serverSocket.accept();
} catch (SocketTimeoutException | SecurityException |
        IllegalBlockingModeException e) {

    // handle exceptions

} catch (IOException e) {

    // handle IOException
}
```

8.4 The try-with-resources Statement

Many Java operations involve some kind of resource that has to be closed after use. Before JDK 7, you used **finally** to make sure a **close** method is guaranteed to be called:

```
try {

    // open resource

} catch (Exception e) {

} finally {
    // close resource
}
```

This syntax can be tedious especially if the **close** method can throw an exception and can be null. For example, here's a typical code fragment to open a database connection.

```
Connection connection = null;
try {

    // create connection and do something with it

} catch (SQLException e) {

} finally {
    if (connection != null) {
        try {
            connection.close();
        } catch (SQLException e) {
        }
    }
}
```

You see, you need quite a bit of code in the **finally** block just for one resource, and it's not uncommon to have to open multiple resources in a single **try** block. JDK 7 added a new feature, the try-with-resource statement, to make resource closing automatic. Its syntax is as follows.

```
try ( resources ) {

    // do something with the resources

} catch (Exception e) {
    // do something with e
}
```

For example, here is opening a database connection would look like in Java 7 and later.

```
Connection connection = null;
try (Connection connection = openConnection();
        // open other resources, if any) {

    // do something with connection

} catch (SQLException e) {

}
```

Not all resources can be automatically closed. Only resource classes that implement **java.lang.AutoCloseable** can be automatically closed. Fortunately, in JDK 7 many input/output and database resources have been modified to support this feature.

8.5 The java.lang.Exception Class

Erroneous code can throw any type of exception. For example, an invalid argument may throw a **java.lang.NumberFormatException**, and calling a method on a null reference variable throws a **java.lang.NullPointerException**. All Java exception classes derive from the **java.lang.Exception** class. It is therefore worthwhile to spend some time examining this class.

Among others, the **Exception** class overrides the **toString** method and adds a **printStackTrace** method. The **toString** method returns the description of the exception. The **printStackTrace** method has the following signature.

```
public void printStackTrace()
```

This method prints the description of the exception followed by a stack trace for the **Exception** object. By analyzing the stack trace, you can find out which line is causing the problem. Here is an example of what **printStackTrace** may print on the console.

```
java.lang.NullPointerException
    at MathUtil.doubleNumber(MathUtil.java:45)
    at MyClass.performMath(MyClass.java: 18)
    at MyClass.main(MyClass.java: 90)
```

This tells you that a **NullPointerException** has been thrown. The line that throws the exception is Line 45 of the **MathUtil.java** class, inside the **doubleNumber** method. The **doubleNumber** method was called by **MyClass.performMath**, which in turns was called by **MyClass.main**.

Most of the time a **try** block is accompanied by a **catch** block that catches the **java.lang.Exception** in addition to other **catch** blocks. The **catch** block that catches **Exception** must appear last. If other **catch** blocks fail to catch the exception, the last **catch** will do that. Here is an example.

```
try {
    // code
} catch (NumberFormatException e) {
    // handle NumberFormatException
} catch (Exception e) {
    // handle other exceptions
}
```

You may want to use multiple **catch** blocks in the code above because the statements in the **try** block may throw a **java.lang.NumberFormatException** or other type of exception. If the latter is thrown, it will be caught by the last **catch** block.

Be warned, though: The order of the **catch** blocks is important. You cannot, for example, put a **catch** block for handling **java.lang.Exception** before any other **catch** block. This is because the JVM tries to match the thrown exception with the argument of the **catch** blocks in the order of appearance. **java.lang.Exception** catches everything; therefore, the **catch** blocks after it would never be executed.

If you have several **catch** blocks and the exception type of one of the **catch** blocks is derived from the type of another **catch** block, make sure the more specific exception type appears first. For example, when trying to open a file, you need to catch the **java.io.FileNotFoundException** just in case the file cannot be found. However, you may want to make sure that you also catch **java.io.IOException** so that other I/O-related exceptions are caught. Since **FileNotFoundException** is a child class of **IOException**, the **catch** block that handles **FileNotFoundException** must appear before the **catch** block that handles **IOException**.

8.6 Throwing an Exception from a Method

When catching an exception in a method, you have two options to handle the error that occurs inside the method. You can either handle the error in the method, thus quietly catching the exception without notifying the caller (this has been demonstrated in the previous examples), or you can throw the exception back to the caller and let the caller handle it. If you choose the second option, the calling code must catch the exception that is thrown back by the method.

Listing 8.2 presents a **capitalize** method that changes the first letter of a **String** to upper case.

Listing 8.2: The capitalize method

```
public String capitalize(String s) throws NullPointerException {
    if (s == null) {
        throw new NullPointerException(
                "You passed a null argument");
    }
    Character firstChar = s.charAt(0);
    String theRest = s.substring(1);
    return firstChar.toString().toUpperCase() + theRest;
}
```

If you pass a null to **capitalize**, it will throw a new **NullPointerException**. Pay attention to the code that instantiates the **NullPointerException** class and throws the instance:

```
     throw new NullPointerException(
             "Your passed a null argument");
```

The **throw** keyword is used to throw an exception. Don't confuse it with the **throws** statement which is used at the end of a method signature to indicate that the method may throw an exception of the given type.

The following example shows code that calls **capitalize**.

```
String input = null;
try {
    String capitalized = util.capitalize(input);
    System.out.println(capitalized);
} catch (NullPointerException e) {
    System.out.println(e.toString());
}
```

Note A constructor can also throw an exception.

8.7 User-Defined Exceptions

You can create a user-defined exception by subclassing **java.lang.Exception**. There are several reasons for having a user-defined exception. One of them is to create a customized error message.

For example, Listing 8.3 shows the **AlreadyCapitalizedException** class that derives from **java.lang.Exception**.

Listing 8.3: The AlreadyCapitalizedException class

```
package app08;
public class AlreadyCapitalizedException extends Exception {
    @Override
    public String toString() {
        return "Input has already been capitalized";
    }
}
```

You can throw an **AlreadyCapitalizedException** from the **capitalize** method in Listing 8.2. The modified **capitalize** method is given in Listing 8.4.

Listing 8.4: The modified capitalize method

```
public String capitalize(String s)
        throws NullPointerException, AlreadyCapitalizedException {
    if (s == null) {
        throw new NullPointerException(
                "Your passed a null argument");
    }
    Character firstChar = s.charAt(0);
    if (Character.isUpperCase(firstChar)) {
        throw new AlreadyCapitalizedException();
    }
    String theRest = s.substring(1);
    return firstChar.toString().toUpperCase() + theRest;
}
```

Now, **capitalize** may throw one of two exceptions. You comma-delimit multiple exceptions in a method signature.

Clients that call **capitalize** must now catch both exceptions. This code shows a call to **capitalize**.

```
StringUtil util = new StringUtil();
String input = "Capitalize";
try {
    String capitalized = util.capitalize(input);
    System.out.println(capitalized);
} catch (NullPointerException e) {
    System.out.println(e.toString());
} catch (AlreadyCapitalizedException e) {
    e.printStackTrace();
}
```

Since **NullPointerException** and **AlreadyCapitalizedException** do not have a parent-child relationship, the order of the **catch** blocks above is not important.

When a method throws multiple exceptions, rather than catch all the exceptions, you can simply write a **catch** block that handles **java.lang.Exception**. Rewriting the code above:

```
StringUtil util = new StringUtil();
String input = "Capitalize";
try {
    String capitalized = util.capitalize(input);
    System.out.println(capitalized);
} catch (Exception e) {
    System.out.println(e.toString());
}
```

While it's more concise, the latter lacks specifics and does not allow you to handle each exception separately.

8.8 Note on Exception Handling

The **try** statement imposes some performance penalty. Therefore, do not use it over-generously. If it is not hard to test for a condition, then you should do the testing rather than depending on the **try** statement. For example, calling a method on a null object throws a **NullPointerException**. Therefore, you could always surround a method call with a **try** block:

```
try {
    ref.method();
…
```

However, it is not hard at all to check if **ref** is null prior to calling **methodA**. Therefore, the following code is better because it eliminates the **try** block.

```
if (ref != null) {
    ref.methodA();
}
```

The **NullPointerException** is one of the most common exceptions a developer has to handle. Java 8 adds the **java.util.Optional** class that can reduce the amount of code for handling the **NullPointerException**.

Self Test

1 Question

?

Consider the following code snippet:

```
int[] wheels = { 1, 2, 4, 5 };
try {
    int i = wheels[10];
} catch (Exception e) {
    e.printStackTrace();
} catch (ArrayIndexOutOfBoundsException e) {
    e.printStackTrace();
}
```

What will happen if you try to compile and run the code?

A. The code will compile and run without problems

B. The code will compile and throw an **ArrayIndexOutOfBoundsException** when run

C. The code will compile and throw a runtime exception other than an **ArrayIndexOutOfBoundsException**

D. The code will not compile

2 Question

Consider the following code fragment:

```
Car[] cars = …;
try {
    Car car = cars[8];
} catch (NullPointerException e) {
    e.printStackTrace();
} catch (ArrayIndexOutOfBoundsException e) {
    e.printStackTrace();
}
```

What will happen if you try to compile and run the code?

A. The code will compile and run without problems

B. The code will compile and may throw an ArrayIndexOutOfBoundsException when run

C. The code will compile and may throw a NullPointerException when run

D. The code will not compile

3 Question

Which of the following statements are correct?

A. A try statement must be accompanied by at least one catch clause

B. A try statement must have a finally clause or at least a catch clause

C. A try statement can stand alone with a finally clause or a catch clause

D. A try statement can have multiple catch clauses

4 Question

How do you write a method that may throw an exception?

A. By using the **throws** keyword

B. By using the **throw** keyword

C. By throwing an instance of **Exception**

D. None of the above

5 Question

Given

```
package test;
public class Printer {
    public void print() {
        print();
    }
    public static void main(String[] args) {
        Printer printer = new Printer();
        printer.print();
    }
}
```

What runtime exception or error will be thrown when the class is executed?

A. java.lang.NullPointerException

B. java.lang.OutOfMemoryError

C. java.lang.ArrayIndexOutOfBoundsException

D. java.lang.StackOverflowError

6 Question

?

Which statements about errors and exceptions are correct?

A. All Java exceptions are derived from **java.lang.Exception**

B. An error is a serious problem that the Java program should not try to catch

C. All errors are derived from **java.lang.Error**

D. Both **java.lang.Error** and **java.lang.Exception** are direct children of **java.lang.Throwable**

7 Question

?

Given

```
class ArrayMismatchedException {
}

public class ArrayUtil {
    public static void main(String[] args) {
        try {
            // some code
        } catch (ArrayMismatchedException e) {
        }
    }
}
```

What happens if you try to compile and run the ArrayUtil class?

A. The code will compile and run without problems

B. The code will not compile because there is no code in the try block

C. The code will not compile because **ArrayMismatchedException** is not derived from **java.lang.Exception**

D. The code will not compile because **ArrayMistmatchedException** is not derived from **java.lang.Throwable**

8 Question

?

Consider the following code.

```
package test;
import java.util.ArrayList;
import java.util.List;
public class ArtistManagement {
    public static void main(String[] args) {
        List<String> artists = new ArrayList<>();
        try {
            artists.add(1, "Will Biteman");
            artists.add(2, "Hermann Longlegs");
        } catch (Exception e) {
            e.printStackTrace();
        }
        for (String artist : artists) {
            System.out.println(artist);
        }
    }
}
```

What happens if you try to compile and run the ArrayUtil class?

A. The code will compile and run without problems

B. The code will not compile

C. The program will print the two elements in **artists**

D. The code will compile but will throw a runtime exception

9 Question

?

Given

```
package demo1;
public class ExceptionDemo {
    public static void main(String[] args) {
        try {
            int count = Integer.parseInt(args[0]);
        } catch (NumberFormatException e) {
            System.err.println("Error: Not a number");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

What exception will be thrown if the program is invoked without arguments?

A. NumberFormatException

B. NullPointerException

C. RuntimeException

D. ArrayIndexOutOfBoundsException

10 Question

?

What can be said of try-with-resource?

A. It is a new feature in JDK 1.7

B. The resource must implement **java.lang.AutoCloseable**

C. It can be used without a catch or a finally block

D. It can be used with any resource

Answers

1 **D**.

Because **ArrayIndexOutOfBoundsException** is a subclass of **Exception**, a catch block that catches an **ArrayIndexOutOfBoundsException** must appear before a catch block that catches an **Exception**. In this case, here is the compile error message: Unreachable catch block for **ArrayIndexOutOfBoundsException**.

2 **B, C**.

Because **ArrayIndexOutOfBoundsException** and **NullPointerException** do not have a parent-child relationship, the position of the two catch blocks may be interchangeable. Since it is unknown what is assigned to cars, the code may throw a runtime exception and there is no guarantee the code will run without problems.

3 **B, D**.

A try statement must have a finally clause or at least a catch clause. In addition, a try statement may be accompanied by multiple catch clauses.

4 **A**.

You use **throws** to indicate that a method may throw an exception.

5 **D**.

A **StackOverflowError** is thrown if a program recurses too deeply, as clearly is the case here when a method calls itself.

6 **A, B, C, D**.

All statements are correct. A problem that may occur when an application is running is either an exception or an error. An error is a serious problem that should not be caught. An exception, on the other hand, may be caught to make sure the program does not crash and cause embarrassment.

7 **D**.

All exceptions must be derived from **java.lang.Throwable**.

8 **D**.

The program will throw an ArrayIndexOutOfBoundsException because it attempts to insert an element at position 2 when the ArrayList is empty.

9 **D**.

If a program is invoked without arguments, the JVM will still create an array that is passed to the **main** method. Therefore, it will not throw a **NullPointerException**. Rather, it will throw an **ArrayIndexOutOfBoundsException** because it tries to access the first element of an empty array.

10 **A, B**.

try-with-resource is a new feature in JDK 1.7 that can close a resource automatically. The resource must implement the **java.lang.AutoCloseable** interface.