ORACLE PRESS

Also Covers **Java SE 11** Developer Exam

OCP

ORACLE CERTIFIED PROFESSIONAL

JAVA SE 21 Developer

Exam 1Z0-830

JAVA SE 17 Developer

Exam 1Z0-829

Programmer's Guide

Khalid A. Mughal
Vasily A. Strelnikov

ORACLE

Important Information About The Programmer's Guide

This Programmer's Guide is comprised of two parts:

- *Part I: OCP Java SE 21 Developer (Exam 1Z0-830)*
- *Part II: OCP Java SE 17 Developer (Exam 1Z0-829)*

Part I can be used in conjunction with Part II to prepare for the *OCP Java SE 21 Developer exam*.

Part II can be used to prepare for the *OCP Java SE 17 and Java 11 Developer exams*.

- The `throw` statement at (2) in the `rethrowA()` method cannot throw the exception, as the first condition is not satisfied: The `try` block does not throw an exception of the right type (`EOFException`). The compiler flags an *error* for the `catch` clause that is unreachable.
- The `throw` statement at (3) in the `rethrowB()` method cannot throw the exception, as the third condition is not satisfied: A preceding `catch` clause can handle the exception (`EOFException`). The compiler flags a *warning* for the `catch` clause which is unreachable.

Example 7.13 *Conditions for Rethrowing Final Exceptions*

```
import java.io.EOFException;
import java.io.FileNotFoundException;
import java.io.IOException;

public class RethrowMe {
  public static void main(String[] args) throws EOFException {
    try {
      switch (1) {
        case 1: throw new FileNotFoundException("File not found");
        case 2: throw new EOFException("End of file");
        default: System.out.println("OK");
      }
    } catch (FileNotFoundException fnfe) {
      System.out.println(fnfe);
    } catch (IOException ioe) {
      throw ioe;                                                    // (1)
    }
  }

  public static void rethrowA() throws EOFException {
    try {
      // Empty try block.
    } catch (EOFException eofe) { // Compile-time error: exception not thrown
                                  //                     in try block.
      throw eofe;                                                   // (2)
    }
  }

  public static void rethrowB() throws EOFException {
    try {
      throw new EOFException("End of file");
    } catch (EOFException eofe) {
      System.out.println(eofe);
    } catch (IOException ioe) {   // Compile-time warning: unreachable clause
      throw ioe;                                                    // (3)
    }
  }
}
```

Chaining Exceptions

It is common that the handling of an exception leads to the throwing of another exception. In fact, the first exception is the cause of the second exception being thrown. Knowing the *cause* of an exception can be useful, for example, when debugging the application. The Java API provides a mechanism for *chaining* exceptions for this purpose.

The class `Throwable` provides the following constructors and methods to handle chained exceptions:

> `Throwable(Throwable cause)`
> `Throwable(String msg, Throwable cause)`
>
> Sets the throwable to have the specified cause, and also specifies a detail message if the second constructor is used. Most exception types provide analogous constructors.
>
> `Throwable initCause(Throwable cause)`
>
> Sets the cause of this throwable. Typically called on exception types that do not provide a constructor to set the cause.
>
> `Throwable getCause()`
>
> Returns the cause of this throwable, if any; otherwise, it returns `null`.

Example 7.14 illustrates how a chain of cause-and-effect exceptions, referred to as the *backtrace,* associated with an exception can be created and manipulated. In the method `chainIt()`, declared at (2), an exception is successively caught and associated as a *cause* with a new exception before the new exception is thrown, resulting in a chain of exceptions. This association is made at (3) and (4). The `catch` clause at (1) catches the exception thrown by the method `checkIt()`. The causes in the chain are successively retrieved by calling the `getCause()` method. Program output shows the resulting backtrace: which exception was the cause of which exception, and the order shown being reverse to the order in which they were thrown, the first one in the chain being thrown last.

Example 7.14 *Chaining Exceptions*

```
import java.io.EOFException;
import java.io.FileNotFoundException;
import java.io.IOException;

public class ExceptionsInChain {
  public static void main(String[] args) {
    try {
      chainIt();
    } catch (Exception e) {                                  // (1)
      System.out.println("Exception chain: " + e);
      Throwable t = e.getCause();
      while (t != null) {
        System.out.println("Cause: " + t);
```

```
        t = t.getCause();
      }
    }
  }

  public static void chainIt() throws Exception {          // (2)
    try {
      throw new FileNotFoundException("File not found");
    } catch (FileNotFoundException e) {
      try {
        IOException ioe = new IOException("File error");
        ioe.initCause(e);                                  // (3)
        throw ioe;
      } catch (IOException ioe) {
        Exception ee = new Exception("I/O error", ioe);    // (4)
        throw ee;
      }
    }
  }
}
```

Output from the program:

```
Exception chain: java.lang.Exception: I/O error
Cause: java.io.IOException: File error
Cause: java.io.FileNotFoundException: File not found
```

A convenient way to print the backtrace associated with an exception is to invoke the `printStackTrace()` method on the exception. The `catch` clause at (1) in Example 7.14 can be replaced with the `catch` clause at (1') below that calls the `printStackTrace()` method on the exception.

```
...
try {
  chainIt();
} catch (Exception e) {                                    // (1')
  e.printStackTrace();                                     // Print backtrace.
}
...
```

The refactoring of Example 7.14 will result in the following analogous printout of the backtrace, showing as before the exceptions and their causes in the reverse order to the order in which they were thrown:

```
java.lang.Exception: I/O error
      at ExceptionsInChain.chainIt(ExceptionsInChain.java:23)
      at ExceptionsInChain.main(ExceptionsInChain.java:8)
Caused by: java.io.IOException: File error
      at ExceptionsInChain.chainIt(ExceptionsInChain.java:19)
      ... 1 more
Caused by: java.io.FileNotFoundException: File not found
      at ExceptionsInChain.chainIt(ExceptionsInChain.java:16)
      ... 1 more
```

7.7 The `try`-with-resources Statement

Normally, objects in Java are automatically garbage collected at the discretion of the JVM when they are no longer in use. However, *resources* are objects that need to be explicitly closed when they are no longer needed. Files, streams, and database connections are all examples that fall into this category of objects. Such resources also rely on underlying system resources for their use. By closing such resources, any underlying system resources are also freed, and can therefore be recycled. Resource leakage (i.e., failure to close resources properly) can lead to performance degradation as resources get depleted.

Best practices recommend the following idiom for using a resource:

- Open the resource—that is, allocate or assign the resource.
- Use the resource—that is, call the necessary operations on the resource.
- Close the resource—that is, free the resource.

Typically, all three steps above can throw exceptions, and to ensure that a resource is always closed after use, the recommended practice is to employ a combination of `try-catch-finally` blocks. The first two steps are usually nested in a `try` block, with any associated `catch` clauses, and the third step is executed in a `finally` clause to ensure that the resource, if it was opened, is always closed regardless of the path of execution through the `try-catch` blocks.

Example 7.15 shows a naive approach to resource management, that can result in resource leakage. It uses a `BufferedReader` associated with a `FileReader` to read a line from a text file (§20.3, p. 1255). The example follows the idiom for resource usage. As we can see, an exception can be thrown from each of the steps. If any of the first two steps throw an exception, the call to the `close()` method at (4) will never be executed to close the resource and thereby any underlying resources, as execution of the method will be terminated and the exception propagated.

Example 7.15 *Naive Resource Management*

```
import java.io.EOFException;
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.IOException;

public class NaiveResourceUse {
  public static void main(String[] args)
      throws FileNotFoundException, EOFException, IOException {

    // Open the resource:
    var fis = new FileReader(args[0]);             // (1) FileNotFoundException
    var br = new BufferedReader(fis);

    // Use the resource:
    String textLine = br.readLine();               // (2) IOException
```

```
      if (textLine != null) {
        System.out.println(textLine);
      } else {
        throw new EOFException("Empty file.");       // (3) EOFException
      }

      // Close the resource:
      System.out.println("Closing the resource.");
      br.close();                                    // (4) IOException
    }
  }
```

Running the program:

```
>java NaiveResourceUse EmptyFile.txt
Exception in thread "main" java.io.EOFException: Empty file.
      at NaiveResourceUse.main(NaiveResourceUse.java:20)
```

Running the program:

```
>java NaiveResourceUse Slogan.txt
Code Compile Compute
Closing the resource.
```

Example 7.16 improves on Example 7.15 by explicitly using `try-catch-finally` blocks to manage the resources. No matter how the `try-catch` blocks execute, the `finally` block at (4) will always be executed. If the resource was opened, the `close()` method will be called at (5). The `close()` method is only called on the `BufferedReader` object, and not on the `FileReader` object. The reason is that the `close()` method of the `BufferedReader` object implicitly calls the `close()` method of the `FileReader` object associated with it. This is typical of how the `close()` method of such resources works. Also, calling the `close()` method on a resource that has already been closed normally has no effect; the `close()` method is said to be *idempotent*.

Example 7.16 *Explicit Resource Management*

```
import java.io.EOFException;
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.IOException;

public class TryWithoutARM {
  public static void main(String[] args) {
    BufferedReader br = null;
    try {
      // Open the resource:
      var fis = new FileReader(args[0]);             // (1) FileNotFoundException
      br = new BufferedReader(fis);

      // Use the resource:
      String textLine = br.readLine();               // (2) IOException
```

```
        if (textLine != null) {
          System.out.println(textLine);
        } else {
          throw new EOFException("Empty file.");       // (3) EOFException
        }
      } catch (FileNotFoundException | EOFException e) {
        e.printStackTrace();
      } catch (IOException ioe) {
        ioe.printStackTrace();
      } finally {                                      // (4)
        if (br != null) {
          try {
            System.out.println("Closing the resource.");
            br.close();                                // (5) IOException
          } catch(IOException ioe) {
            ioe.printStackTrace();
          }
        }
      }
    }
  }
```

Running the program:

```
>java TryWithoutARM EmptyFile.txt
java.io.EOFException: Empty file.
      at TryWithoutARM.main(TryWithoutARM.java:20)
Closing the resource.
```

Running the program:

```
>java TryWithoutARM Slogan.txt
Code Compile Compute
Closing the resource.
```

A lot of boilerplate code is required in explicit resource management using `try-catch-finally` blocks, and the code can get tedious and complex, especially if there are several resources that are open and they all need to be closed explicitly. The try-with-resources statement takes the drudgery out of associating `try` blocks with corresponding `finally` blocks to ensure proper resource management. Any resource declared in the header of the try-with-resources statement will be automatically closed, regardless of how execution proceeds in the `try` block. The compiler expands the try-with-resources statement (and any associated `catch` or `finally` clauses) into basic `try-catch-finally` blocks to guarantee this behavior. It implicitly generates a `finally` block that calls the `close()` method of each resource declared in the header of the try-with-resources statement. There is no need to call the `close()` method of the resource, let alone provide any explicit `finally` clause for this purpose.

The `close()` method provided by the resources declared in a try-with-resources statement is the implementation of the sole `abstract` method declared in the `java.lang.AutoCloseable` interface. In other words, these resources must implement the `AutoCloseable` interface (p. 414).