# Chapter 13 - Exceptions

## Section 13.1 - Exception basics

***Error-checking code*** is code a programmer writes to detect and handle errors that occur during program execution. An ***exception*** is a circumstance that a program was not designed to handle, such as if the user enters a negative height.

The following program, given a person's weight and height, outputs a person's body-mass index (BMI), which is used to determine normal weight for a given height. The program has no error checking.

## Figure 13.1.1: BMI example without error checking.

```java
import java.util.Scanner;

public class BMINoErrorCheck {
   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      int weightVal = 0;    // User defined weight (lbs)
      int heightVal = 0;    // User defined height (in)
      float bmiCalc = 0.0f; // Resulting BMI
      char quitCmd = 'a';   // Indicates quit/continu

      while (quitCmd != 'q') {

         // Get user data
         System.out.print("Enter weight (in pounds):
         weightVal = scnr.nextInt();

         System.out.print("Enter height (in inches):
         heightVal = scnr.nextInt();

         // Calculate BMI value
         bmiCalc = ((float) weightVal /
                    (float) (heightVal * heightVal))

         //Print user health info
         // Source: http://www.cdc.gov/
         System.out.println("BMI: " + bmiCalc);
         System.out.println("(CDC: 18.6-24.9 normal)"

         // Prompt user to continue/quit
         System.out.print("\nEnter any key ('q' to quit): ");
         quitCmd = scnr.next().charAt(0);
      }

      return;
   }
}
```

```
Enter weight (in pounds): 150
Enter height (in inches): 66
BMI: 24.207989
(CDC: 18.6-24.9 normal)

Enter any key ('q' to quit):
Enter weight (in pounds): -1
Enter height (in inches): 66
BMI: -0.1613866
(CDC: 18.6-24.9 normal)

Enter any key ('q' to quit):
Enter weight (in pounds): 150
Enter height (in inches): -1
BMI: 105450.0
(CDC: 18.6-24.9 normal)

Enter any key ('q' to quit):
```

Naively adding error-checking code using if-else statements obscures the normal code. And redundant checks are ripe for errors if accidentally made inconsistent with normal code. Problematic code is highlighted.

## Figure 13.1.2: BMI example with error-checking code but without using exception-handling constructs.

```java
import java.util.Scanner;

public class BMINaiveErrorCheck {
   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      int weightVal = 0;     // User defined weight (lbs)
      int heightVal = 0;     // User defined height (in)
      float bmiCalc = 0.0f; // Resulting BMI
      char quitCmd = 'a';    // Indicates quit/continue

      while (quitCmd != 'q') {

         // Get user data
         System.out.print("Enter weight (in pounds): ");
         weightVal = scnr.nextInt();

         // Error checking, non-negative weight
         if (weightVal < 0) {
            System.out.println("Invalid weight.");
         }
         else {
            System.out.print("Enter height (in inches): ");
            heightVal = scnr.nextInt();
            // Error checking, non-negative height

            if (heightVal < 0) {
               System.out.println("Invalid height.");
            }
         }

         // Calculate BMI and print user health info if no inp
         // Source: http://www.cdc.gov/
         if ((weightVal <= 0) || (heightVal <= 0)) {
            System.out.println("Cannot compute info.");
         }
         else {
            bmiCalc = ((float) weightVal /
                       (float) (heightVal * heightVal)) * 703.0f;

            System.out.println("BMI: " + bmiCalc);
            System.out.println("(CDC: 18.6-24.9 normal)");
            // Source: http://www.cdc.gov/
         }

         // Prompt user to continue/quit
         System.out.print("\nEnter any key ('q' to quit): ");
         quitCmd = scnr.next().charAt(0);
      }

      return;
   }
}
```

```
Enter weight (in pou
Enter height (in incl
BMI: 24.207989
(CDC: 18.6-24.9 norma

Enter any key ('q' to
Enter weight (in pou
Invalid weight.
Cannot compute info.

Enter any key ('q' to
Enter weight (in pou
Enter height (in incl
Invalid height.
Cannot compute info.

Enter any key ('q' to
```

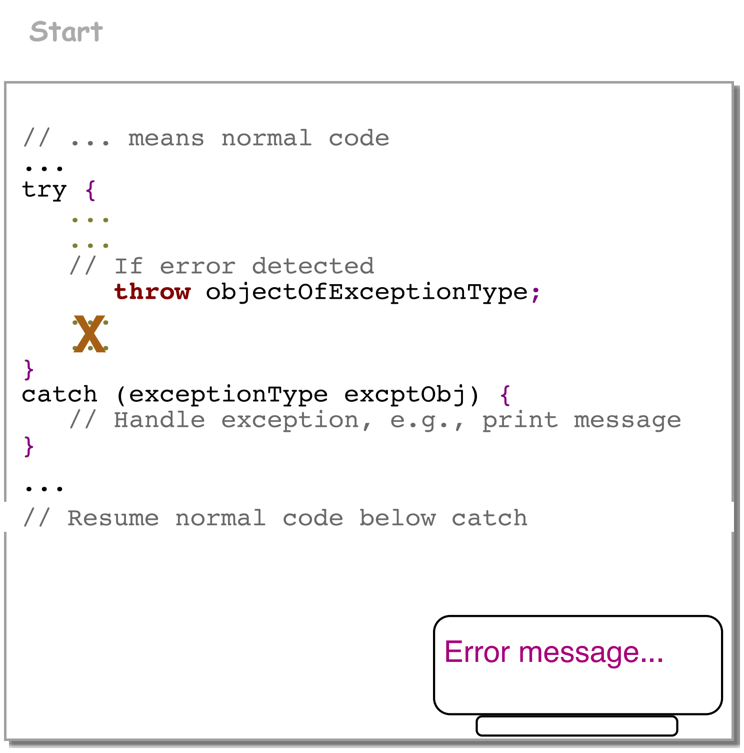
The language has special constructs, try, throw, and catch, known as ***exception-handling constructs***, to keep error-checking code separate and to reduce redundant checks.

Construct 13.1.1: Exception-handling constructs.

```
// ... means normal code
...
try {
   ...
   // If error detected
      throw objectOfExceptionType;
   ...
}
catch (exceptionType excptObj) {
   // Handle exception, e.g., print message
}
...
```

P Participation Activity

## 13.1.1: How try, throw, and catch handle exceptions.



```
// ... means normal code
...
try {
   ...
   ...
   // If error detected
      throw objectOfExceptionType;
   ...
}
catch (exceptionType excptObj) {
   // Handle exception, e.g., print message
}
...
// Resume normal code below catch
```

- A ***try*** block surrounds normal code, which is exited immediately if a throw statement executes.
- A ***throw*** statement appears within a try block; if reached, execution jumps immediately to the end of the try block. The code is written so only error situations lead to reaching a throw. The throw statement provides an object of type ***Throwable***, such as an object of type Exception or its subclasses. The statement is said to throw an exception of the particular type. A throw statement's syntax is similar to a return statement.
- A ***catch*** clause immediately follows a try block; if the catch was reached due to an exception thrown of the catch clause's parameter type, the clause executes. The clause is said to catch the thrown exception. A catch block is called a ***handler*** because it handles an exception.

The following shows the earlier BMI program using exception-handling constructs. Notice that the normal code flow is not obscured by error-checking/handling if-else statements. The flow is clearly: Get weight, then get height, then print BMI.

## Figure 13.1.3: BMI example with error-checking code using exception-handling constructs.

```
import java.util.Scanner;

public class BMIExceptHandling {
   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      int weightVal = 0;     // User defined weight (lbs)
      int heightVal = 0;     // User defined height (in)
      float bmiCalc = 0.0f; // Resulting BMI
      char quitCmd = 'a';    // Indicates quit/continue

      while (quitCmd != 'q') {

         try {
            // Get user data
            System.out.print("Enter weight (in pounds): ");
            weightVal = scnr.nextInt();

            // Error checking, non-negative weight
            if (weightVal < 0) {
               throw new Exception("Invalid weight.");
            }

            System.out.print("Enter height (in inches): ");
            heightVal = scnr.nextInt();

            // Error checking, non-negative height
            if (heightVal < 0) {
               throw new Exception("Invalid height.");
            }

            // Calculate BMI and print user health info if no
            // Source: http://www.cdc.gov/
            bmiCalc = ((float) weightVal
                    / (float) (heightVal * heightVal)) * 703.0

            System.out.println("BMI: " + bmiCalc);
            System.out.println("(CDC: 18.6-24.9 normal)");
         }
         catch (Exception excpt) {
            // Prints the error message passed by throw statement
            System.out.println(excpt.getMessage());
            System.out.println("Cannot compute health info");
         }

         // Prompt user to continue/quit
         System.out.print("\nEnter any key ('q' to quit): ");
         quitCmd = scnr.next().charAt(0);
      }

      return;
   }
}
```

```
Enter weight (in pou
Enter height (in inc
BMI: 24.208
(CDC: 18.6-24.9 norm

Enter any key ('q' t
Enter weight (in pou
Invalid weight.
Cannot compute healt

Enter any key ('q' t
Enter weight (in pou
Enter height (in inc
Invalid height.
Cannot compute healt

Enter any key ('q' t
```

The object thrown and caught must be of the Throwable class type, or a class inheriting from Throwable. As discussed elsewhere, Java offers several built-in Throwable types like Error, Exception, and classes derived from these. The Exception class (and other Throwable types) has a constructor that can be passed a String, as in `throw new Exception("Invalid weight.");`, which allocates a new Exception object and sets an internal String value that can later be retrieved using the getMessage() method, as in `System.out.println(excpt.getMessage());`.

P Participation Activity

## 13.1.2: Exceptions.

Select the one code region that is incorrect.

| # | Question |
|---|---|
| 1 | `try {`<br>`   if (weight < 0) {`<br>`      try new Exception("Invalid weight.");`<br>`   }`<br><br>`   //Print user health info`<br>`   // ...`<br>`}`<br>`catch (Exception excpt) {`<br>`   System.out.println(excpt.getMessage());`<br>`   System.out.println("Cannot compute health info");`<br>`}` |
| 2 | `try {`<br>`   if (weight < 0) {`<br>`      throw new Exception( "Invalid weight." );`<br>`   }`<br><br>`   //Print user health info`<br>`   // ...`<br>`}`<br>`catch (Exception excpt) {`<br>`      System.out.println( excpt() );`<br>`      System.out.println("Cannot compute health info");`<br>`}` |

P Participation Activity

13.1.3: Exception basics.

| # | Question | Your answer |
|---|---|---|
| 1 | After an exception is thrown and a catch block executes, execution resumes after the throw statement. | True |
| | | False |
| 2 | A compiler generates an error message if a try block is not immediately followed by a catch block. | True |
| | | False |
| 3 | If no throw is executed in a try block, then the subsequent catch block is not executed. | True |
| | | False |

Exploring further:

- More on Exceptions from Oracle's Java Tutorials
- Oracle's Java Exception class specification

## Section 13.2 - Exceptions with methods

The power of exceptions becomes clearer when used within a method. If an exception is thrown within a method and not caught within that method, then the method is immediately exited and the calling method is checked for a handler, and so on up the method call hierarchy. The following illustrates; note the clarity of the normal code.

## Figure 13.2.1: BMI example using exception-handling constructs along with methods.

```java
import java.util.Scanner;

public class BMIExceptHandling {
   public static int getWeight() throws Exception {
      Scanner scnr = new Scanner(System.in);
      int weightParam = 0; // User defined weight (lbs)

      // Get user data
      System.out.print("Enter weight (in pounds): ");
      weightParam = scnr.nextInt();

      // Error checking, non-negative weight
      if (weightParam < 0) {
         throw new Exception("Invalid weight.");
      }
      return weightParam;
   }

   public static int getHeight() throws Exception {
      Scanner scnr = new Scanner(System.in);
      int heightParam = 0; // User defined height (in)

      // Get user data
      System.out.print("Enter height (in inches): ");
      heightParam = scnr.nextInt();

      // Error checking, non-negative height
      if (heightParam < 0) {
         throw new Exception("Invalid height.");
      }
      return heightParam;
   }

   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      int weightVal = 0;     // User defined weight (lbs)
      int heightVal = 0;     // User defined height (in)
      float bmiCalc = 0.0f; // Resulting BMI
      char quitCmd = 'a';    // Indicates quit/continue

      while (quitCmd != 'q') {

         try {
            //Get user data
            weightVal = getWeight();
            heightVal = getHeight();

            // Calculate BMI and print user health info if no input error
            // Source: http://www.cdc.gov/
            bmiCalc = ((float) weightVal /
                       (float) (heightVal * heightVal)) * 703.0f;

            System.out.println("BMI: " + bmiCalc);
            System.out.println("(CDC: 18.6-24.9 normal)");
         } catch (Exception excpt) {
            // Prints the error message passed by throw statement
            System.out.println(excpt.getMessage());
```

```
Enter weight (in pou
Enter height (in incl
BMI: 24.207989
(CDC: 18.6-24.9 norma

Enter any key ('q' to
Enter weight (in pou
Invalid weight.
Cannot compute healtl

Enter any key ('q' to
Enter weight (in pou
Enter height (in incl
Invalid height.
Cannot compute healtl

Enter any key ('q' to
                  System.out.println(excpt.getMessage());
                  System.out.println("Cannot compute health info");
               }

               // Prompt user to continue/quit
               System.out.print("\nEnter any key ('q' to quit): ");
               quitCmd = scnr.next().charAt(0);
            }

            return;
         }

      }
```

Suppose GetWeight() throws an exception of type runtime_error. GetWeight() immediately exits, up to main() where the call was in a try block, so the catch block catches the exception.

Note the clarity of the code in main(). Without exceptions, GetWeight() would have had to somehow indicate failure, perhaps returning -1. Then main() would have needed an if-else statement to detect such failure, obscuring the normal code.

If a method throws an exception not handled within the method, a programmer must include a ***throws clause*** within the method declaration, by appending `throws Exception` before the opening curly brace. Java requires that a programmer either provides an exception handler or specifies that a method may throw an exception by appending a throws clause to all methods that may throw checked exceptions. A ***checked exception*** is an exception that a programmer should be able to anticipate and appropriately handle. Checked exceptions include Exception and several of its subclasses, discussed elsewhere in the context of file input/output. A common error is forgetting either to specify a throws clause or forgetting to enclose code that may throw exceptions with try-catch constructs, which results in a compiler error such as: "unreported exception java.lang.Exception; must be caught or declared to be thrown".

***Unchecked exceptions***, in contrast to checked expressions, are exceptions that result from hardware or logic errors that typically cannot be anticipated or handled appropriately, and instead should be eliminated from the program or at the very least should cause the program to terminate immediately. A programmer is not required to handle unchecked exceptions or even specify that a method may throw them. Unchecked exceptions are comprised of the Error and RuntimeException classes and their subclasses. Examples of built-in unchecked exceptions include NullPointerException, ArithmeticException, IndexOutOfBoundsException, and IOError, which are automatically thrown whenever a programmer attempts to use a null reference, divides an integer by zero, attempts to access a non-existing element within an array, or when a hardware failure causes an I/O operation to fail, respectively. The following table provides an overview of common unchecked exceptions with links to the corresponding class specification page from Oracle.

Table 13.2.1: Common unchecked exceptions.

| Unchecked exception | Notes |
|---|---|
| NullPointerException | Indicates a null reference. |
| IndexOutOfBoundsException | Indicates that an index (e.g., an index for an array) is outside the appropriate range. |
| ArithmeticException | Indicates the occurrence of an exceptional arithmetic condition (e.g., integer division by zero). |
| IOError | Indicates the failure of an I/O operation. |
| ClassCastException | Indicates an invalid attempt to cast an object to type of which the object is not an instance (e.g., casting a Double to a String). |
| IllegalArgumentException | Thrown by a method to indicate an illegal or inappropriate argument. |

P Participation Activity

### 13.2.1: Exceptions.

| # | Question | Your answer |
|---|---|---|
| 1 | For a method that may contain a throw, the method's statements must be surrounded by a try block. | True<br>False |
| 2 | A throw executed in a method automatically causes a jump to the last return statement in the method. | True<br>False |
| 3 | A goal of exception handling is to avoid polluting normal code with distracting error-handling code. | True<br>False |
| 4 | A checked exception must either be handled via try-catch constructs, or the throwing method must specify that the appropriate exception type may be thrown by appending a throws clause to the method's definition. | True<br>False |

## Section 13.3 - Multiple handlers

Different throws in a try block may throw different exception types. Multiple handlers may exist, each handling a different type. The first matching handler executes; remaining handlers are skipped.

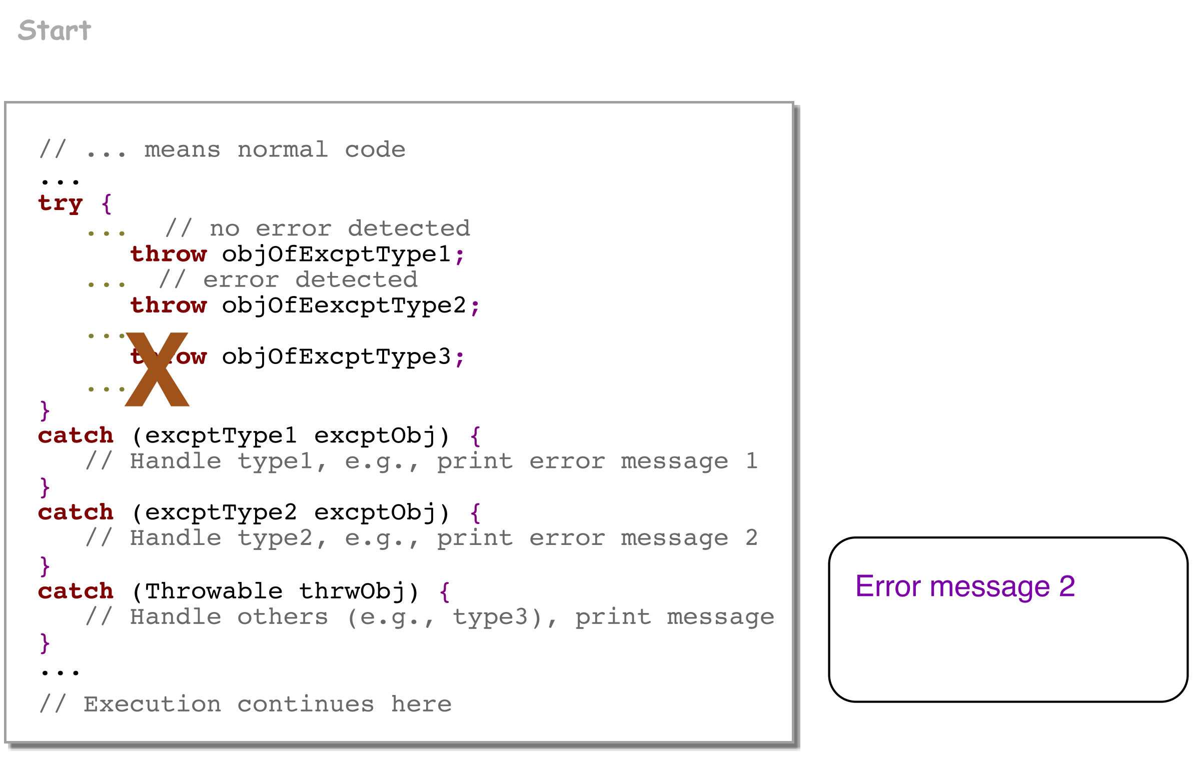
***catch(Throwable thrwObj)*** is a catch-all handler that catches any error or exception as both are derived from the Throwable class; this handler is useful when listed as the last handler.

### Construct 13.3.1: Exception-handling: multiple handlers.

```
// ... means normal code
...
try {
   ...
   throw objOfExcptType1;
   ...
   throw objOfExcptType2;
   ...
   throw objOfExcptType3;
   ...
}
catch (excptType1 excptObj) {
   // Handle type1
}
catch (excptType2 excptObj) {
   // Handle type2
}
catch (Throwable thrwObj) {
   // Handle others (e.g., type3)
}
... // Execution continues here
```

P Participation Activity

## 13.3.1: Multiple handlers.



```
// ... means normal code
...
try {
    ...   // no error detected
       throw objOfExcptType1;
    ...  // error detected
       throw objOfEexcptType2;
    ...
       throw objOfExcptType3;
    ...
}
catch (excptType1 excptObj) {
    // Handle type1, e.g., print error message 1
}
catch (excptType2 excptObj) {
    // Handle type2, e.g., print error message 2
}
catch (Throwable thrwObj) {
    // Handle others (e.g., type3), print message
}
...
// Execution continues here
```

```
Error message 2
```

A thrown exception may also be caught by a catch block meant to handle an exception of a base class. If in the above code, excptType2 is a subclass of excptType1, then objOfExcptType2 would always be caught by the first catch block instead of the second catch block, which is typically not the intended behavior. A common error is to place a catch block intended to handle exceptions of a base class before catch blocks intended to handle exceptions of a derived class, resulting in a compiler error with a message such as: "exception has already been caught".

P Participation Activity

### 13.3.2: Exceptions with multiple handlers.

Refer to the multiple handler code above.

| # | Question | Your answer |
|---|---|---|
| 1 | If an object of type objOfExcptType1 is thrown, three catch blocks will execute. | True<br>False |
| 2 | If an object of type objOfExcptType3 is thrown, no catch blocks will execute. | True<br>False |
| 3 | A second catch block can never execute immediately after a first one executes. | True<br>False |
| 4 | If excptType2 inherits from excptType1, then the inclusion of the second catch block (i.e., `catch (excptType2 excptObj)`) below the first catch block results in a compiler error as the second catch block would never be executed. | True<br>False |

## Section 13.4 - Exception handling in file input/output

A file input/output stream requires exception handling to ensure invalid or interrupted file operation exceptions are appropriately caught and handled. The following naively attempts to handle exceptions for reading and printing characters from a text file until reaching the end of the file.

Figure 13.4.1: First attempt at reading from a file with exception handling.

```java
import java.util.Scanner;
import java.io.FileReader;
import java.io.IOException;

public class FileReadChars {
   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      FileReader fileCharStream = null; // File stream for reading chars
      String fileName = "";             // User defined file name
      int charRead = 0;                 // Data read from file

      // Get file name from user
      System.out.print("Enter a valid file name: ");
      fileName = scnr.next();

      try {
         // Prompt user for input
         System.out.println("Opening file " + fileName + ".");
         fileCharStream = new FileReader(fileName); // May throw FileNotFoundExcept

         // Use file input stream
         System.out.print("Reading character values: ");
         while (charRead != -1) {             // -1 means end of file has been reac
            charRead = fileCharStream.read(); // May throw IOException
            System.out.print(charRead + " ");
         }

         // Done with file, so try to close it
         if (fileCharStream != null) {
            System.out.println("\nClosing file " + fileName + ".");
            fileCharStream.close(); // close() may throw IOException if fails
         }
      } catch (IOException excpt) {
         System.out.println("Caught IOException: " + excpt.getMessage());
      }

      return;
   }
}
```

letters.txt with seven characters:

```
abcdefg
```

```
Enter a valid file name: letters.txt
Opening file letters.txt.
Reading character values: 97 98 99 100 101 102 103 10 -1
Closing file letters.txt.

...

Enter a valid file name: badfile.txt
Opening file badfile.txt.
Caught IOException: badfile.txt (No such file or directory)
```

The ***FileReader*** class provides an input stream that allows a programmer to read characters from the file specified via FileReader's constructor. FileReader supports several overloaded read() methods for reading characters and a close() method for terminating the stream and closing the file. Most FileReader methods and constructors throw exceptions of type ***IOException*** (i.e., input/output exception), a built-in checked exception inheriting from the Exception class. Specifically, read() may throw an IOException if an error is encountered while reading the file, and close() may throw an IOException if an error occurs while attempting to close the file. FileReader's constructors may throw a ***FileNotFoundException***, which itself inherits from IOException, if the specified file cannot be opened for reading.

In the above program, the try block contains instructions that open a file, read from that file, and close the file. The single catch block is able to handle any exceptions thrown by each of these operations individually, and the program seems to function as intended. However, suppose that an interrupted read() operation causes the statement `charRead = fileCharStream.read();` to throw an IOException. Although the catch block successfully catches and reports the error, the program exits without closing the file. This behavior is due to the fact that program execution immediately jumps to the end of the try block as soon as a contained statement throws an exception. A programmer must ensure that files are closed when no longer in use, to allow the JVM to clean up any resources associated with the file streams.

One possible solution is to place the call to close() after the try-catch blocks:

Figure 13.4.2: Closing file after try-catch block.

```
import java.util.Scanner;
import java.io.FileReader;
import java.io.IOException;

public class FileReadChars {
   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      FileReader fileCharStream = null; // File stream for reading chars
      String fileName = "";             // User defined file name
      int charRead = 0;                 // Data read from file

      // Get file name from user
      System.out.print("Enter a valid file name: ");
      fileName = scnr.next();

      try {
         // Prompt user for input
         System.out.println("Opening file " + fileName + ".");
         fileCharStream = new FileReader(fileName); // May throw FileNotFoundExcept

         // Use file input stream
         System.out.print("Reading character values: ");
         while (charRead != -1) {                 // -1 means end of file has been reac
            charRead = fileCharStream.read(); // May throw IOException
            System.out.print(charRead + " ");
         }
      } catch (IOException excpt) {
         System.out.println("Caught IOException: " + excpt.getMessage());
      }

      // Done with file, so try to close it
      try {
         if (fileCharStream != null) {
            System.out.println("\nClosing file " + fileName + ".");
            fileCharStream.close(); // close() may throw IOException if fails
         }
      } catch (IOException excpt) {
         System.out.println("Caught IOException: " + excpt.getMessage());
      }

      return;
   }
}
```

letters.txt with seven characters:

```
abcdefg
```

```
Enter a valid file name: letters.txt
Opening file letters.txt.
Reading character values: 97 98 99 100 101 102 103 10 -1
Closing file letters.txt.

...

Enter a valid file name: badfile.txt
Opening file badfile.txt.
Caught IOException: badfile.txt (No such file or directory)
```

However, if the program throws an exception unrelated to the file IO (e.g., RuntimeException, NullPointerException) before calling close(), the program may still exit without first closing the file.

A better solution uses a finally block. A ***finally block*** follows all catch blocks, and executes after the program exits the corresponding try or catch blocks.

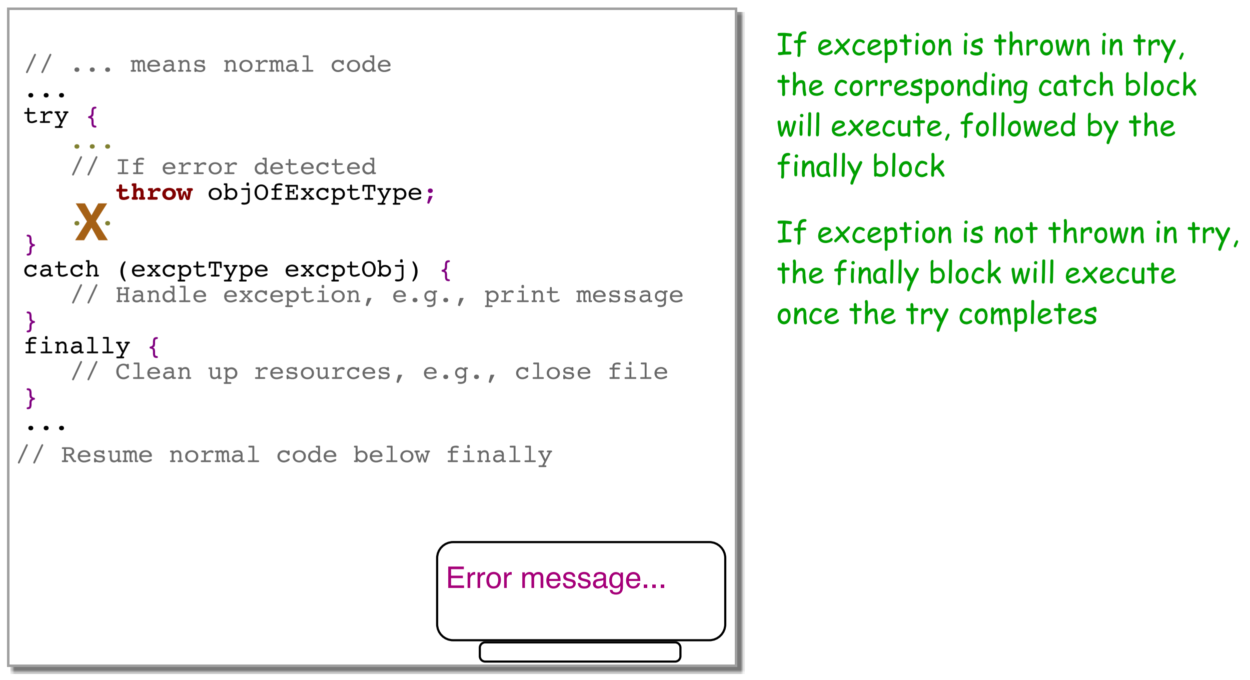
## Construct 13.4.1: Finally block.

```
// ... means normal code
...
try {
   ...
   // If error detected
      throw objOfExcptType;
   ...
}
catch (excptType excptObj) {
   // Handle exception, e.g., print message
}
finally {
   // Clean up resources, e.g., close file
}
...
```

P Participation Activity

## 13.4.1: Using a finally block to clean up resources.

Start



```
// ... means normal code
...
try {
   ...
   // If error detected
      throw objOfExcptType;
   ...
}
catch (excptType excptObj) {
   // Handle exception, e.g., print message
}
finally {
   // Clean up resources, e.g., close file
}
...
// Resume normal code below finally
```

If exception is thrown in try, the corresponding catch block will execute, followed by the finally block

If exception is not thrown in try, the finally block will execute once the try completes

A good practice is to use finally blocks for code that should be executed both when the program executes normally and when the program throws an exception, such as closing files that are opened and accessed within a try block.

The following is an improved version of the previous program, using a finally block to ensure that the input stream and file are always closed. The code within the try block opens a file and reads characters from that file. If an error occurs while reading from the file, an IOException will be thrown, and the catch block will report the exception. The finally block executes after the try and catch blocks have finished execution calling CloseFileReader(), which will close the file. As an exception may be thrown while closing a file, the CloseFileReader method also includes try and catch blocks.

Figure 13.4.3: Reading from a file with appropriate resource clean-up.

```
import java.util.Scanner;
import java.io.FileReader;
import java.io.IOException;

public class FileReadChars {
   /* Method closes a FileReader.
      Prints exception message if closing fails */
   public static void closeFileReader(FileReader fileName) {
      try {
         if (fileName != null) { // Ensure "file" references a valid object
            System.out.println("Closing file.");
            fileName.close(); // close() may throw IOException if fails
         }
      } catch (IOException closeExcpt) {
         System.out.println("Error closing file: " + closeExcpt.getMessage());
      }

      return;
   }

   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      FileReader fileCharStream = null; // File stream for reading chars
      String fileName = "";              // User defined file name
      int charRead = 0;                  // Data read from file

      // Get file name from user
      System.out.print("Enter a valid file name: ");
      fileName = scnr.next();

      try {
         // Prompt user for input
         System.out.println("Opening file " + fileName + ".");
         fileCharStream = new FileReader(fileName); // May throw FileNotFoundExcept

         // Use file input stream
         System.out.print("Reading character values: ");
         while (charRead != -1) { // -1 means end of file has been reached
            charRead = fileCharStream.read(); // May throw IOException
            System.out.print(charRead + " ");
         }
         System.out.println();
      } catch (IOException excpt) {
         System.out.println("Caught IOException: " + excpt.getMessage());
      } finally {
         closeFileReader(fileCharStream); // Ensure file is closed!
      }

      return;
   }
}
```

letters.txt with seven characters:

```
abcdefg
```

```
Enter a valid file name: letters.txt
Opening file letters.txt.
Reading character values: 97 98 99 100 101 102 103 10 -1
Closing file.

...

Enter a valid file name: badfile.txt
Opening file badfile.txt.
Caught IOException: bad.txt (No such file or directory)
```

While the previous programs simply print a message if the program cannot open a file for reading, the following example allows the user to enter a different file name if the file cannot be opened by catching the FileNotFoundException.

Figure 13.4.4: Reading from a file with an improved FileNotFoundException handler.

```
import java.util.Scanner;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class FileReadCharsInteractive {
   /* Method prints characters in a file using read().
      Throws IOException if read() operation fails. */
   public static void readFileChars(FileReader file) throws IOException {
      int charRead = 0; // Data read from file

      // Use file input stream
      System.out.print("Reading character values: ");
      while (charRead != -1) {   // -1 means end of file has been reached
         charRead = file.read(); // May throw IOException
         System.out.print(charRead + " ");
      }
      System.out.println();

      return;
   }

   /* Method closes a FileReader.
```

```
      Prints exception message if closing fails. */
   public static void closeFileReader(FileReader fileName) {
      try {
         if (fileName != null) { // Ensure "file" references a valid object
            System.out.println("Closing file.");
            fileName.close(); // close() may throw IOException if fails
         }
      } catch (IOException closeExcpt) {
         System.out.println("Error closing file: " + closeExcpt.getMessage());
      }

      return;
   }

   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      FileReader fileCharStream = null; // File stream for reading chars
      String fileName = "";             // User defined file name
      boolean validFile = true;         // Ensures file opened

      do {
         // Get file name from user
         System.out.print("Enter a valid file name (or 'q' to quit): ");
         fileName = scnr.next();

         if (fileName.equals("q")) {
            break; // Exit do-while loop
         }

         try {
            // Prompt user for input
            System.out.println("Opening file " + fileName + ".");
            fileCharStream = new FileReader(fileName); // May throw FileNotFoundExc

            validFile = true; // If reached this statement, file opened successfull

            // Read chars from file
            readFileChars(fileCharStream); // May throw IOException
         } catch (FileNotFoundException excpt) {
            System.out.println("Caught FileNotFoundException: " + excpt.getMessage(
            validFile = false;
         } catch (IOException excpt) {
            System.out.println("Caught IOException: " + excpt.getMessage());
         } finally {
            closeFileReader(fileCharStream); // Ensure file is closed!
         }
      } while (!validFile);

      return;
   }
}
```

letters.txt with seven characters:

```
Enter a valid file name (or 'q' to quit): letters.txt
Opening file letters.txt.
Reading character values: 97 98 99 100 101 102 103 10 -1
Closing file.

...
```

```
abcdefg
```

```
Enter a valid file name (or 'q' to quit): badfile.txt
Opening file badfile.txt.
Caught FileNotFoundException: badfile.txt (No such file or directory
Enter a valid file name (or 'q' to quit): q
```

The statements in main() execute in a do-while loop until the user enters a valid file name. The first catch catches the FileNotFoundException exception and sets a boolean variable to false indicating the file is not valid. The second catch catches any other IOException, such as an exception that occurs while reading from the file. As the FileNotFoundException exception is derived from an IOException, the catch for the FileNotFoundException must come before the catch for the IOException.

A programmer should also utilize exception handling when writing to file output streams in order to handle exceptions resulting from invalid or interrupted file operations, as shown below. The program opens a user-specified file, writes the first 10 letters of the alphabet, and safely closes the file.

Figure 13.4.5: Writing to a file with exception handling.

```
import java.util.Scanner;
import java.io.FileWriter;
import java.io.IOException;

public class FileWriteChars {
   /* Method closes a FileWriter.
      Prints exception message if closing fails. */
   public static void closeFileWriter(FileWriter fileName) {
      try {
         if (fileName != null) { // Ensure "file" references a valid object
            System.out.println("Closing file.");
            fileName.close(); // close() may throw IOException if fails
         }
      } catch (IOException closeExcpt) {
         System.out.println("Error closing file: " + closeExcpt.getMessage());
      }

      return;
   }

   public static void main(String[] args) {
      Scanner scnr = new Scanner(System.in);
      final int NUM_CHARS_TO_WRITE = 10; // Num chars to write to file
      int countVar = 0;                  // Track num chars written so far
      FileWriter fileCharStream = null;  // File stream for writing file
      String fileName = "";              // User defined file name
      char charWrite = 'a';              // Char data written to file

      // Get file name from user
      System.out.print("Enter a valid file name: ");
      fileName = scnr.next();

      try {
         System.out.println("Creating file " + fileName + ".");
         fileCharStream = new FileWriter(fileName); // May throw IOException
         // Use file output stream
         System.out.print("Writing " + NUM_CHARS_TO_WRITE + " characters: ");
         while (countVar < NUM_CHARS_TO_WRITE) {
            fileCharStream.write(charWrite);
            System.out.print(charWrite);

            charWrite++; // Get next char ready
            countVar++;  // Keep track of number chars written
         }
         System.out.println();
      } catch (IOException excpt) {
         System.out.println("Caught IOException: " + excpt.getMessage());
      } finally {
         closeFileWriter(fileCharStream); // Ensure file is closed!
      }

      return;
   }
}
```

```
Enter a valid file name: mywritefile.txt
Creating file mywritefile.txt.
Writing 10 characters: abcdefghij
Closing file.
```

outputfile mywritefile.txt with ten characters:

```
abcdefghij
```

The program creates an object of type ***FileWriter***, which provides overloaded write() methods to write a stream of characters and a close() method to flush the underlying buffer and close the stream. Both of these methods may throw IOExceptions in the event of a failure. A FileWriter's constructor may throw an IOException if the program or operating system cannot create (or open) the specified file.

P Participation Activity

## 13.4.2: Exceptions with file input and output.

| # | Question | Your answer |
|---|---|---|
| 1 | What type of exception may be thrown by a FileReader's constructor? | |
| 2 | What is the name of the base class from which all file i/o exceptions discussed above are derived? | |
| 3 | What type of exception may be thrown by a FileReader's read() and close() methods? | |
| 4 | What type of exception may be thrown by a FileWriter's constructor? | |
| 5 | What is the name of the clause used to ensure that contained statements are executed after the try and catch blocks. | |

Exploring further:

- More on the finally block from Oracle's Java Tutorials
- Oracle's Java FileReader class specification
- Oracle's Java FileWriter class specification
- Oracle's Java IOException class specification

## Section 13.5 - Java example: Generate number format exception

P Participation Activity

### 13.5.1: Catch number format error.

Running the below program with the given input causes a number conversion error. The program reads from System.in the following rows (also called records) that contain a last name, first name, department, and annual salary. The program uses the String class split() method to split an input row.

```
Argon,John,Operations,50000
Williams,Jane,Marketing,60000.0
Uminum,Al,Finance,70000
Jones,Ellen,Sales,80000
```

Note that the second row has a value that is type double, not type int, which is going to cause a problem.

1. Run the program and note the line number where the program fails. The line number can be found on the last row of the error messages.
2. Add try/catch statements to catch the number conversion error, which is called NumberFormatException. In this case, print a message, and do not add the item to the total salaries.
3. Run the program again and note the total salaries excludes the row with the error.

Reset

```
 1 import java.util.Scanner;
 2 
 3 public class StreamNumberException {
 4 
 5    public static void main(String [] args) {
 6       // Describe the format of a row of input. There are four fields in a row
 7       // separate by commas: last name, first name, department, salary
 8       final String SEPARATOR    = ",";  // Field separator in each row of data
 9       final int INDEX_LAST_NAME  = 0;  // # of the last name field
10       final int INDEX_FIRST_NAME = 1;  // # of the first name field
11       final int INDEX_DEPT       = 2;  // # of the department name field
12       final int INDEX_SALARY     = 3;  // # of the salary field
13       Scanner scnr = new Scanner(System.in);
14 
15       String [] field;        // Fields on one row in the input file
16       String row;             // One row of the input file
17       int nRows = 0;          // Counts # of rows in the input file
18       int totalSalaries = 0;  // Total of all salaries read
```

```
      int i = 0;                // Loop counter
```

```
Doe,John,Operations,50000
Doette,Jane,Marketing,60000.0
Uminum,Al,Finance,70000
en,Sales,80000
```

Run

P Participation Activity

## 13.5.2: Catch number format error (solution).

Below is a solution to the above problem.

Reset

```
import java.util.Scanner;

public class StreamNumberExceptionSolution {

   public static void main(String [] args) {
      // Describe the format of a row of input. There are four fields in a row
      // separate by commas: last name, first name, department, salary
      final String SEPARATOR    = ",";  // Field separator in each row of data
      final int INDEX_LAST_NAME  = 0;  // # of the last name field
      final int INDEX_FIRST_NAME = 1;  // # of the first name field
      final int INDEX_DEPT       = 2;  // # of the department name field
      final int INDEX_SALARY     = 3;  // # of the salary field
      Scanner scnr = new Scanner(System.in);

      String [] field;        // Fields on one row in the input file
      String row;             // One row of the input file
      int nRows = 0;          // Counts # of rows in the input file
      int totalSalaries = 0;  // Total of all salaries read
      int i = 0;              // Loop counter
```

```
Doe,John,Operations,50000
Doette,Jane,Marketing,60000.0
Uminum,Al,Finance,70000
en,Sales,80000
```

Run