Chapter 04 Exception Handling & Text I/O

Exceptions are runtime errors. Runtime errors occur while a program is running if the JVM detects an operation that is impossible to carry out. For example, if you access an array using an index that is out of bounds, you will get a runtime error with an `ArrayIndexOutOfBoundsException`. If you enter a double value when your program expects an integer, you will get a runtime error with an `InputMismatchException`.

If the exception is not handled, the program will terminate abnormally. **Exception Handling** enables a program to deal with runtime errors and continue its normal execution. This chapter introduces Exception handling, and text input and output.

1. Exception Types

Exceptions are objects, and objects are defined using classes. List of Java Exceptions

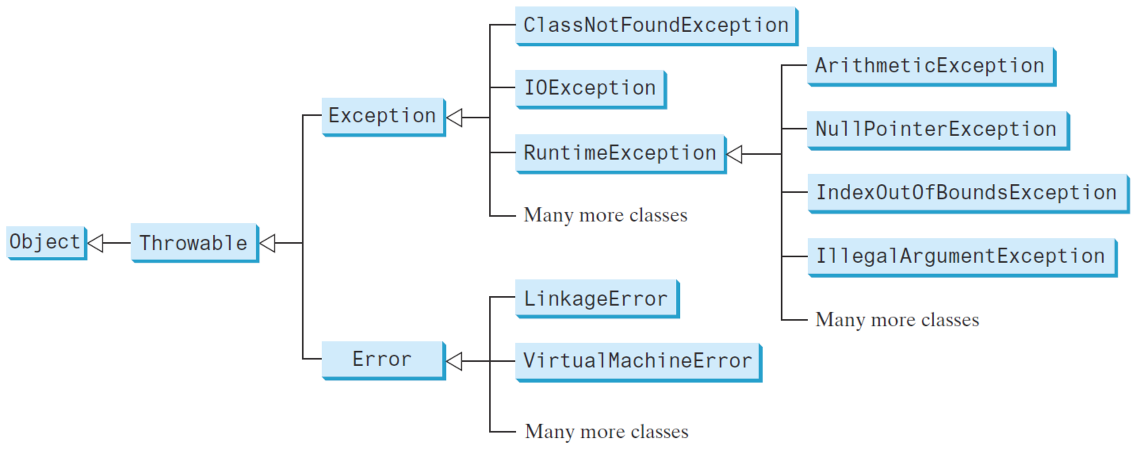

FIGURE 12.1 Exceptions thrown are instances of the classes shown in this diagram, or of subclasses of one of these classes.

2. Exception Handling

We know that exceptions abnormally terminate the execution of a program. This is why it is important to handle exceptions. Here is a list of different approaches to handle exceptions in Java:

- `try`...`catch` block
- `finally` block
- `throw` and `throws` keyword

2.1 `try`...`catch` Block

The try-catch block is used to handle exceptions in Java. Here is the syntax of `try`... `catch` block:

```
try {
    // code
}
catch(ExceptionType e) {
    // code
}
```

Here, we have placed the code that might generate an exception inside the try block. Every try block is followed by a `catch` block.

When an exception occurs, it is caught by the catch block. The `catch` block cannot be used without the `try` block.

Example 01: Exception handling using `try`...`catch`

```
class Program {
    public static void main(String[] args) {
        try {
            // code that generate an exception
            int divideByZero = 5 / 0;

            System.out.println("Rest of code in the try block");
        }
        catch (ArithmeticException e) {
            System.out.println(e);
        }
    }
}
```

Output

```
java.lang.ArithmeticException: / by zero
```

In the example, we are trying to divide a number by 0. Here, this code generates an exception. To handle the exception, we have put the code, 5 / 0 inside the `try` block. Now when an exception occurs, the rest of the code inside the `try` block is skipped. The `catch` block catches the exception and statements inside the `catch` block is executed. If none of the statements in the `try` block generates an exception, the `catch` block is skipped.

2.2 `finally` Block

The `finally` block is always executed no matter whether there is an exception or not. The `finally` block is optional. And, for each `try` block, there can be only one `finally` block.

The basic syntax of `finally` block is:

```
try {
    // code
}
catch (ExceptionType e) {
    // code
}
finally {
    // code that always executes
}
```

Example 02: Exception handling using `try`...`catch`...`finally`

```
class Program {
    public static void main(String[] args) {
        try {
            // code that generate an exception
            int divideByZero = 5 / 0;

            System.out.println("Rest of code in the try block");
        }
        catch (ArithmeticException e) {
            System.out.println(e);
        }
        finally {
            System.out.println("This is the finally block");
        }
    }
}
```

Output

```
java.lang.ArithmeticException: / by zero
This is the finally block
```

In the above example, we are dividing a number by 0 inside the `try` block. Here, this code generates an `ArithmeticException`. The exception is caught by the `catch` block. And, then the `finally` block is executed.

Note that it is a good practice to use the `finally` block in case you need to include important cleanup codes like such as:

- code that might be accidentally skipped by return or break
- closing a file or connection

2.3 Multiple `catch` Blocks

For each `try` block, there can be one or many `catch` blocks. Multiple catch blocks allow us to handle each exception differently.

This is called **Exception Filters**. Exception filters are useful when you want to handle different types of exceptions in different ways.

Example 03: Exception handling using multiple `catch` block.

```
class Program {
    public static void main(String[] args){
        try {
            int array[] = new int[10];
            array[20] = 30 / 0;
        }
        catch (ArithmeticException e) {
            System.out.println(e);
        }
        catch (ArrayIndexOutOfBoundsException e) {
            System.out.println(e);
        }
        catch (Exception e){
            System.out.print(e);
        }
    }
}
```

Output

```
java.lang.ArithmeticException: / by zero
```

Note that at a time only one exception occurs and at a time only one `catch` block is executed. Also, all `catch` blocks must be ordered from most specific to most general, i.e., catch for `ArithmeticException` must come before catch for `Exception`.

2.4 Handling Multiple Exceptions in a Single `catch` Block

We can catch more than one type of exception with one catch block. This reduces code duplication and increases code simplicity and efficiency. Each exception type that can be handled by the catch block is separated using a vertical bar |.

Syntax:

```
try {
    // code
} catch (ExceptionType1 | Exceptiontype2 e) {
    // catch block
}
```

Example 04: Handle multiple exceptions in a single catch block.

```java
class Program {
    public static void main(String[] args){
        try {
            int array[] = new int[10];
            array[20] = 30 / 1;
        }
        catch (ArithmeticException | ArrayIndexOutOfBoundsException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Output

```
Index 20 out of bounds for length 10
```

2.5 `throw` and `throws` Keywords

The `throw` keyword is mainly used to explicitly throw a **custom** exception **within** a method or block of code. We can specify the exception object which is to be thrown with a message with it that provides the error description. We can also define our own set of conditions and throw an exception explicitly using `throw` keyword.

The syntax of `throw` keyword:

```java
throw throwableObject;
```

Example 05: Exception handling using `throw` keyword

```java
class Program {
    public static void validate(int age){
        if(age < 18) {
            //throw exception if not eligible to vote
            throw new ArithmeticException("Person is not eligible to vote");
        }
        else {
            System.out.println("Person is eligible to vote!!");
        }
    }
    public static void main(String[] args){
        try {
            validate(2);
        }
        catch(ArithmeticException e){
            System.out.println(e.getMessage());
        }
    }
}
```

Output

```
Person is not eligible to vote
```

Similarly, the `throws` keyword is used to **declare** the type of exceptions that can be thrown from a method.

The syntax of `throws` keyword:

```
accessModifier returnType methodName() throws ExceptionType1, ExceptionType2,... {
    // method code
}
```

Example 06: Exception handling using `throws` keyword

```
class Program {
    // declare the type of exception
    public static void divide() throws ArithmeticException {

        // code that may generate an exception
        int divideByZero = 5 / 0;
    }
    public static void main(String[] args){
        try {
            divide();
            System.out.println("This is a text after calling divide() method.");
        }
        catch (ArithmeticException e) {
            System.out.println(e);
        }
    }
}
```

Output

```
java.lang.ArithmeticException: / by zero
```

Remember that if you want an exception to be handled by its caller, you should create an exception object and throw it. The caller will have to handle the exception using `try`...`catch` block or rethrow it.

2.6 Rethrowing Exceptions

Java allows an exception handler to rethrow the exception if the handler cannot process the exception, or simply wants to let its caller be notified of the exception.

The syntax for rethrowing an exception may look like this:

```
try {
    // code;
}
catch (ExceptionType e) {
    // perform operations before exits;
    throw e;
}
```

3. The File Class

Having learned exception handling, you are ready to step into file processing. Data used in a program is temporary; it is lost when the program terminates. To permanently store the data created in a program, you need to store it in a file.

3.1 Create a File Object

To create an object of File, we need to import the `java.io.File` first. Once we import the class, here is how we can create objects of file.

```
// Create a file object
File file = new File(String path/fileName);
```

For example:

```
File file1 = new File("myFile1.txt"); // Create a file object for the current
                                         directory

File file2 = new File("myFolder/myFile2.txt"); // myFolder directory must already
                                               // exist
```

Note: In Java, creating a file object does not mean creating a file. In the above example, we created two file objects which can be used to work with files.

3.2 File Methods

Method	Description	Class
`createNewFile`()	Creates a new file. And it will return `true` if a new file is created, `false` if the file already exists in the specified location.	`java.io.File`
`exists`()	returns `true` if the file exists, otherwise, returns `false`.	`java.io.File`
`delete`()	Deletes a specified file. It returns `true` if the file is deleted, and returns `false` if the file does not exist.	`java.io.File`
`read`()	`read`() - reads a single character from the reader. This method returns `-1` when the end of the file is reached. `read(char[] array)` - reads the characters from the reader and stores in the specified array. `read(char[] array, int start, int length)` - reads the number of characters equal to length from the reader and stores in the specified array starting from the position start.	`java.io.FileReader`
`readLine()`	read a file line-by-line to String. This method returns `null` when the end of the file is reached.	`java.io.BufferedReader`
`write`()	Writes data to a file. The old content will be overridden by the new content. `write`() - writes a single character to the writer. `write(char[] array)` - writes the characters from the specified array to the writer. `write(String data)` - writes the specified string to the writer.	`java.io.FileWriter`

`close`()	To close the file reader/write object, we can use the `close`() method.	`java.io.FileReader`
		`java.io.BufferedReader`
		`java.io.FileWriter`

Example 07: Create a new file.

```
import java.io.File;

class Program {
    public static void main(String[] args){
        // Create a file object
        File file = new File("myFile.txt");

        try {
            // try to create a file
            file.createNewFile(); // must be caught or declared to be thrown
        }
        catch(Exception e) {
            System.out.println(e);
        }
    }
}
```

Output

```
myFile.txt
```

Example 08: Check if creating a new file is done successfully.

```
import java.io.File;

class Program {
    public static void main(String[] args){
        // Create a file object
        File file = new File("myFile.txt");

        try {
            // try to create a file
            boolean isFileCreated = file.createNewFile();

            if (isFileCreated){
                System.out.println("The new file is created.");
            }
            else {
                System.out.println("The file already exists.");
            }
        }
        catch(Exception e) {
            System.out.println(e);
        }
    }
}
```

Output

```
The file already exists.
```

Example 09: Check if a file exists.

```java
import java.io.File;

class Program {
    public static void main(String[] args){

        File file = new File("myFile.txt");

        // Check if the file exists
        if(file.exists()) {
            System.out.println("File exists");
        }
        else {
            System.out.println("File does not exist");
        }
    }
}
```

Output

```
File exists
```

3.3 Writing to Files

The `FileWriter` class defined in the `java.io.FileWriter` class can be used to create a file and write data to a file.

First, you have to create a `FileWriter` object for a file as follows:

```
FileWriter writer = new FileWriter(filename);
```

Then, you can invoke the `write`() method to write data to a file. When we are done, the writer object needs to be closed.

Example 10: Write data to a text file.

```
import java.io.FileWriter;

class Program {
    public static void main(String[] args){
        try {
            // Create a Writer that is linked with the myFile.txt
            FileWriter writer = new FileWriter("myFile.txt"); // must be caught
                                                              // or declared to
                                                              // be thrown
            // Write data to the file
            writer.write("Welcome to Java\n");     // must be caught or declared
                                                   // to be thrown
            writer.write("Programming is Fun\n");  // must be caught or declared
                                                   // to be thrown

            // Closes the writer
            writer.close();   // must be caught or declared to be thrown
        }
        catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

Output

myFile.txt

```
Welcome to Java
Programming is Fun
```

3.4 Appending Files

To add new data to a file without overriding the old data, we can pass `true` as a second argument to `FileWriter` to turn on "append" mode.

```
FileWriter writer = new FileWriter("filename",  true);
```

For example, we have a file below:

Countries.txt

```
Cambodia
Thailand
China
Japan
```

Example 11: Append data to a file.

```
import java.io.FileWriter;

class Program {
    public static void main(String[] args){
        try {
            // Create a Writer that is linked with the myFile.txt
            FileWriter writer = new FileWriter("Countries.txt", true);

            // app the data to the file
            writer.write("Singapore\n");
            writer.write("Austria\n");

            // Closes the writer
            writer.close();
        }
        catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

Output

Countries.txt

```
Cambodia
Thailand
China
Japan
Singapore
Austria
```

3.5 Reading from Files

The FileReader class defined in the java.io.FileReader class can be used to read data from a file

First, you have to create a FileReader object for a file, for example:

```
FileReader reader = new FileReader("myFile.txt");
```

Then, you can invoke the read() method to read data from a file. When we are done, the reader object needs to be closed.

Example 12: Read data from a file.

```
import java.io.FileReader;

class Program {
    public static void main(String[] args){
        char[] array = new char[100];

        try {
            // Creates a reader that is linked with the myFile.txt
            FileReader reader = new FileReader("myFile.txt");

            // Reads characters
            reader.read(array);

            System.out.println(array);

            // Closes the reader
            reader.close();
        }
        catch(Exception e) {
            System.out.println(e);
        }
    }
}
```

Output

```
Welcome to Java
Programming is Fun
```

Example 13: Reads all characters from a file one char at a time.

```
import java.io.FileReader;

class Program {
    public static void main(String[] args){
        int i;

        try {
            // Creates a reader that is linked with the myFile.txt
            FileReader reader = new FileReader("myFile.txt");

            // Reads the character one by one and display it
            while((i = reader.read()) != -1){
                System.out.print((char)i + " ");
            }

            // Closes the reader
            reader.close();
        }
        catch(Exception e) {
            System.out.println(e);
        }
    }
}
```

Output

```
W e l c o m e   t o   J a v a
 P r o g r a m m i n g   i s   F u n
```

The `read`() method of the `FileReader` returns an int which contains the char value of the character read. If the `read`() method returns -1, there is no more data to read in the `FileReader`.

You can use the readLine() method from java.io.BufferedReader to read a file line-by-line to String. This method returns null when the end of the file is reached.

Example 14: Read data from a file line by line.

```java
import java.io.FileReader;
import java.io.BufferedReader;

class Program {
    public static void main(String[] args){
        String line;

        try {
            // Creates a reader that is linked with the myFile.txt
            BufferedReader reader = new BufferedReader(
                                                    new FileReader("myFile.txt"));

            // Reads the file line by line and display it
            while ((line = reader.readLine()) != null) {
                System.out.print(line + "\n");
            }

            // Closes the reader
            reader.close();
        }
        catch(Exception e) {
            System.out.println(e);
        }
    }
}
```

Output

```
Welcome to Java
Programming is Fun
```

3.6 Deleting Files

We can use the delete() method of the File class to delete the specified file. It returns true if the file is deleted, and returns false if the file does not exist.

Example 15: Delete a file.

```
import java.io.File;

class Program {
    public static void main(String[] args){
        // creates a file object
        File file = new File("myFile.txt");

        // deletes the file
        boolean isFileDeleted = file.delete();

        if(isFileDeleted) {
            System.out.println("The File is deleted.");
        }
        else {
            System.out.println("The File is not deleted.");
        }
    }
}
```

Output

```
The File is deleted.
```

Exercises

1. Using the two arrays shown below, write a program that prompts the user to enter an integer between 1 and 12 and then displays the months and its number of days corresponding to the integer entered. Your program should display "wrong number" if the user enters a wrong number by catching ArrayIndexOutOfBoundsException.

```
String[] months = {"January", "February", "March", "April",
"May", "June","July", "August", "September", "October",
"November", "December"};
int[] dom = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

2. The previous program works well as long as the user enters an integer. Otherwise, you may get another kind of exception. For instance, if you use nextInt() of Scanner, you could have an InputMismatchException. Modify it to prevent users entering anything other than an integer.
3. Implement a method called hex2Binary() to throw a NumberFormatException if the string is not a hex string.
4. (Count characters, words, and lines in a file) Write a program that will count the number of characters, words, and lines in a file. Words are separated by whitespace characters.

5. (Occurrences of each letter) Write a program that prompts the user to enter a file name and displays the occurrences of each letter in the file. Letters are case insensitive. Here is a sample run:

```
Enter a filename: Lincoln.txt
Number of As: 56
Number of Bs: 134
...
Number of Zs: 9
```

6. Define a class called `Employee` that contains:
   - Data fields: `id`, `name`, `gender` and `salary`.
   - Methods:
     - `readEmployee`() that asks the user to enter `id`, `name`, `gender` and `salary` of an employee.
     - `addEmployee`() that save the employee data into Employee.txt file.
     - `deleteEmployee`() that removes the employee data from Employee.txt file. Hint: read the content of the store it into a variable. Remove the employee and store the content back into the file.

   For the test program, store the data of three employees in the Employee.txt file. Then, display a menu that will allow the user to select any of the following features:

   a. Add a new employee
   b. Delete employee by id
   c. Search employee by id
   d. Display all employee
   e. Exit the program

   Note:
   - For searching, display the student if found, otherwise, displays `"Search Not Found"`.
   - When display employee(s), the employee data should be arranged in a tabular format, for example:

```
----------------------------------------------------------------------
ID              Name                      Gender              Salary
----------------------------------------------------------------------
1               Lucy                      F                   300
2               John                      M                   400
3               Alex                      M                   500
-----------------------------------------------------------------------
```

7. Write an ATM machine program. First, define a class called `Account` that contains:
   - Data fields: `accountNo`, `name`, `balance` and `password`.
   - Methods:
     - `login`() that accepts an account number and a password to login, then returns True if the login is successful, otherwise, returns False.
     - `displayBlance`() that displays the user's balance.
     - `widthraw`() that asks the user to enter an amount to withdraw then update the Account.txt file if the transaction is successful.
     - `deposit`() that asks the user to enter an amount to deposit then updates the Account.txt file.
     - `transfer`() that asks the user to enter an amount to transfer and the receiver's account number, then update the Account.txt file if the transaction is successful.

For the test program, store the data of five users in a file called Account.txt. When the program starts, ask the user to enter an account number and a password to login. If login succeeded, display the following menus:

a. Balance
b. Withdraw
c. Deposit
d. Transfer
e. Exit the program

Reference

[1] Y. Daniel Liang. ‘Introduction to Java Programming’, 11e – 2019

[2] https://www.programiz.com/java-programming