Chapter 14

Error Handling

Introduction

- Error handling refers to the response and recovery procedures from error conditions present in a software application.
- Error handling helps in maintaining the normal flow of program execution.
- Error handling helps in handling both hardware and software errors gracefully and helps execution to resume when interrupted.
- There are four main categories of errors:
  - Logical errors
  - Generated errors
  - Compile-time errors
  - Runtime errors
- There are different techniques and processes that are involved in dealing with errors of each of these different categories.
- The errors in certain categories may be solved and avoided altogether with the help of some basic proofreading of the code.
- Other types of errors may require the programmer or developer to identify the problem with the help of test data and deal with it using resolution programs.
- When it comes to error handling in software, either the programmer develops the necessary codes to handle errors or makes use of software tools to handle the errors.
- Special applications known as error handlers are available for certain applications to help in error handling.

Logical Errors

- Logical errors are often considered to be the most difficult types of errors to spot.
- A program with logical errors and exceptions causing a program to terminate or stop working altogether,
- Sometimes a program with logical errors will run perfectly fine, but produce incorrect results.
- The example below produces different outputs of 11, 13, 9, and 8, due to different usage of parentheses.

```
public class LogicalErrors {

        public static void main(String[] args) {
                // Create variables Var1 thru Var4
                int Var1 = 5 + 4 * 3 / 2;
                int Var2 = (5 + 4) * 3 / 2;
                int Var3 = (5 + 4) * (3 / 2);
                int Var4 = (5 + (4 * 3)) / 2;

                // Print the results.
                System.out.println(
                        “Var1: ” + Var1 +
                        “\nnVar2: ” + Var2 +
                        “\nnVar3: ” + Var3 +
                        “\nnVar4: ” + Var4
                );
        }
}
```

Output:

```
        Var1: 11
        nVar2: 13
        nVar3: 9
        nVar4: 8
```

- In certain cases, logical errors that have not been spotted at an early stage in the code or program may even have the potential to affect data and values that appear thousands of lines of code after the error was initially made.
- One of the most common logical errors that bother programmers is misuse or misplacing of a semicolon.
- Here is an example of how misplaced semicolons work:

```
public class ErrorForLoop {

        public static void main(String[] args) {

                // Variable Declaration.
                int Counter ;

                // Create For Loop.
                for (Counter = 1; Counter <= 10; Counter++);
                {
                        // Print the result.
                        System.out.println(“Counter is ” + Counter);
                }
        }
}
```

Output:

```
Counter is 11
```

- Most logical errors can be removed by proofreading.
- Techniques that are employed for error handling are often termed as *debugging* or *troubleshooting*.

Syntactical Errors

- Syntactical errors are errors in which the wrong language, or syntax, is used to write a code or program.
- Not writing the condition in parentheses for an *if* statement.
- Syntactical errors may not be as difficult to spot as their logical counterparts because the compiler will most likely catch the majority of these errors.
- Some of the most common syntactical errors are.

- **Capitalization**
    - Java is a case-sensitive language.
    - Java programmers start capitalizing keywords instead of writing them in lowercase.
    - Making a change as apparently insignificant as writing myCount instead of MyCount has the potential to ruin the entire code and fill it with more errors
    - Using the right capitalization is essential for all class names, variable names, and any other piece of code, when writing a program in Java.

- **Splitting Strings**
    - Dividing your code into a number of lines often does not matter when you are coding in Java. Splitting a string so that it comes on more than one line or contains a new line character in it will cause the compiler to throw an exception or object to the code.
    - The solution to splitting problem is to add a double quote to the string that appears on the first line, and then add a plus sign right after this first half of the string ends to show the compiler that whatever follows in the next one or more lines needs to be added or concatenated to the same string. An example of how this can be done is as follows:

```
System.out.print ( “This is the first half of the string ” +
“this is the second half of the string that needs to be concatenated. ” );
```

- **Not Importing Classes**

  - One of the most common semantic errors occurs in Java program due to programmers forgetting to include an associated class when they wish to make use of a particular API feature.
  - For instance, if you wish to incorporate the String data type in your code, it is essential to add the class to your application using Import Java.lang. String; for the String class to be imported in your application.

- **Different Methods**

  - In Java, the static methods and instance methods work differently in this language.
  - Static methods are those that are associated with a specific class, whereas instance methods are associated with the object that is created from a certain class.
  - In case if a static method is treated as an instance method in Java, then the java compiler will display with a syntactical error since the way in which both of these are dealt with differ greatly.
  - **Instance method**
    - Instance methods are the methods defined under a class and we can call such functions only after creating an object of that class.
    - Some of the properties of the instance method are the following:
      - Instance methods are related to an object rather than a class as they can be invoked after creating an object of the class.
      - We can override an instance method as they are resolved using dynamic binding during run time.
      - The instance methods are stored in a single memory location.
    - **Example:** the following program illustrating the working of instance method:

```
class myClass {

        // Data member of type String
        String name = "";

        // Defining an instance method
        public void assignName(String name) {
                this.name = name;
        }
}

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

                // Create an object of the class
                myClass object = new myClass();

                // Calling an instance method in the class 'myClass'.
                object.assignName("Hello World");

                // Display the name
                System.out.println(object.name);
        }
}
```

Output:

```
    Hello World
```

- **Static methods**
  - An static method in Java is a method that can be called without creating an object of the class.
  - It can reference such methods by the class name or direct reference to the object of that class.
  - Some of the properties of the static method are the following:
    - An static method is related to a class rather than an object of the class. We can call the static method directly without creating an object of the class.
    - Static methods are designed in such a way that they can be shared among all objects created using the same class.
    - We cannot override static methods as they are resolved using the static binding by the compiler during compile time.
  - **Example:** The following program illustrating the working of static method:

```
// Create a class
class myClass {

    // Create a data member
    public static String name = "";

    // Create static method
     public static void assignName(String passedName) {
        name = passedName;
    }
}
public class HelloWorld {
    public static void main(String[] args) {

        // Accessing the static method assignName()
        myClass.assignName("Hello World");
        System.out.println(myClass.name);
```

```
                    // Accessing the static method assignName()
                    // having the object reference
                    myClass object = new myClass();

                    // Assign the name
                    object.assignName("Java Program");

                    // Display the name
                    System.out.println(object.name);
          }
}
```

Output:

```
          Hello World
          Java Program
```

- **Static versus instance methods**

Static	**Instance**
Static methods can access the static variables and static methods directly.	Instance methods on the other hand can access instance variables and instance methods directly.
Static methods can't access instance variables and instance methods directly without using the reference of an object.	Instance methods can access static variables and static methods directly.
"this" operator cannot be used by static methods.	Instance methods can use "this" operator.

- **Curly Braces**
  - When programming in Java, the programmer creates a block of code enclosed in curly braces to ensure that the compiler understands where the code that the feature needs to be applied to starts and finishes.
  - For instance, if a programmer forgets to finish the contents of a class with curly braces, then the compiler catch this syntactical and notified of missing curly braces.

Semantic Errors

- Figuring out the difference between semantic and syntactical errors is one of the biggest problems.

- But there are certain significant differences between these types of errors.
- While syntactical errors have to do with the syntax of the code, semantic errors are related to the usage of the code.
- It is possible that semantic errors in the code even if the syntax is correct.
- The most common type of semantic errors are those in which variables are used without proper initialization. Fortunately, these errors will be caught by the compiler in most cases and notification about the variable in question.
- The most common semantic errors are discussed below:

- **Improper Use of Operators**

    - Sometimes, in case of operators are used improperly on variables, it may give an impression of syntactical error. However, in reality it is more of a semantic error.
    - For example
        - The increment operator (++), for instance, cannot use Boolean variables and attempting to do so will be a semantic error.
        - Another common operator error mistake the use of comparator (==) operator with objects. Since this is not allowed and comparator operators can only be used with primitive types
    - An error message will be generated showing that the operation intended is not permissible.

- **Incompatible Types**

    - Incompatible error is done by accident or simply due to the lack of knowledge.
    - It is a semantic error that may or may not be caught by the compiler.
    - For instance,
        - Assign a float value to an int variable, the compiler will present an error message.
        - Assign an int to a float, the compiler will automatically convert the integer value into a float value and does not present error message.

- **Precision**

    - While a float variable can be converted to an int variable by applying casting, as a result loss of precision, which will be affected along with the results in all of the other lines or blocks of code where the value in question will be used.
    - Therefore, recommended that use casting when you absolutely must and know that output can potentially be affected.

- **Scoping**

    - Scoping is an issue that defines what is and is not allowed within a certain scope.

- For instance, if you try to declare a private static int variable inside a method, you will receive an error.
- **Example:**

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

                private static int intPrivate = 5;

                // The contents of the method will go here
                System.out.println("intPrivate value : " + intPrivate);
        }
}
```

Output:

```
        ERROR!
        /tmp/iKJ9j4oZCY/VariablePrivate.java:5: error: illegal start of expression
        private static int intPrivate = 5;
```

- Instead, you should declare the variable globally so that it can be used properly.
- **Example: the f**ollowing programs is the correct way to declare the *private static int intPrivate* variable.

```
public class VariablePrivate {

        private static int intPrivate = 5;
        public static void main(String[] args) {

                // The contents of the method will go here
                System.out.println("intPrivate value : " + intPrivate);
        }
}
```

Output:

```
        intPrivate value : 5
```

- On the other hand, if you try to declare the private static int variable called intPrivate within the method itself, you will get an error message.
- Following is the incorrect way of declaring the private static int variable:

```
public class VariablePrivateIncorrect {
        public static void main(String[] args) {
                private static int intPrivate = 5;
                // The contents of the method will go here
                System.out.println("intPrivate value : " + intPrivate);
        }
}
```

Importance of Error Handling

- Error handling does not only ensure smooth operation but also guarantees that the code will not malfunction and will provide the desired results.
- Since even the simplest of applications are easily a few thousand lines of code long and comprise code written by a number of programmers, it is extremely important to take care of all errors and exceptions as you move forward. This will ensure that errors, exceptions, and other problems of the sort do not carry forward until the end of the program.

Java Exceptions

- **In Java, Exception** is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program's instructions.
- When executing Java code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.
- When an error occurs, Java will normally stop and generate an error message. The technical term for this is: Java will throw an **exception** (throw an error).
- In Java, exceptions are broadly categorized into two sections:
  - Checked exceptions
  - Unchecked exceptions

Checked Exceptions

- Checked exceptions are those that are identified at the time when the code is being compiled.
- Checked exceptions are called compile-time exceptions.
- Checked exceptions occur when the program interacts with other systems/network resources e.g. *database errors*, *network connection errors*, *missing files*, etc.
- To handle checked Exception *it* is mandatory to provide try-catch or try -finally block, otherwise it will result in a compile-time error.
- Note that all checked exceptions are subclasses of Exception class. For example,
  - ClassNotFoundException
  - IOException
  - SQLException

- **Example**

```
import java.io.*;
public class CheckedExceptionsExample {
        public static void main(String[] args){
                FileReader file = new FileReader("D:\\newfolder\\example.txt");
                BufferedReader fileInput = new BufferedReader(file);
                for (int counter = 0; counter < 3; counter++)
```

```
                        System.out.println(fileInput.readLine());
                        // This block of code will output the first 3 lines of the file
                        // "D:\newfolder\example.txt"
                fileInput.close();
        }
}
```

Output:

```
ERROR!
/tmp/Y2spvw8DqF/CheckedExceptionsExample.java:4: error: unreported exception
FileNotFoundException; must be caught or declared to be thrown
FileReader file = new FileReader("D:\\newfolder\\example.txt");
            ^
ERROR!
/tmp/Y2spvw8DqF/CheckedExceptionsExample.java:7: error: unreported exception
IOException; must be caught or declared to be thrown
System.out.println(fileInput.readLine());
                         ^
ERROR!
/tmp/Y2spvw8DqF/CheckedExceptionsExample.java:10: error: unreported exception
IOException; must be caught or declared to be thrown
fileInput.close();
          ^
3 errors
```

Runtime Exceptions

- Runtime exceptions are identified when the code is being run.
- A Runtime exception is called unchecked exceptions.
- Runtime exceptions are not checked by the compiler.
- RuntimeException does not mondatary use of try-catch or try -finally block.
- Runtime exceptions will come into life and occur in the program, once any buggy code is executed.
- Unchecked Exceptions are subclasses of RuntimeException class.
    - ArithmeticException
    - ArrayStoreException
    - ClassCastException

- **Example**

```
public class RuntimeExceptionExample {
        public static void main(String args[]) {
                int var1 = 0;
                int var2 = 10;
                int var3 = var2 / var1;
        }
}
```

Output:

```
ERROR!
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at RuntimeExceptionExample.main(RuntimeExceptionExample.java:6)
```

Java try and catch

- The try statement allows you to define a block of code to be tested for errors while it is being executed.
- The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.
- The try and catch keywords come in pairs:
- Syntax

```
try {
    //  Block of code to try
}

catch(Exception e) {
     //  Block of code to handle errors
}
```

- Consider the following example: This will generate an error, because **myNumbers[10]** does not exist.

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

                int[] myNumbers = {1, 2, 3};
                System.out.println(myNumbers[10]); // error!
        }
}
```

Output:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
10 at Main.main(Main.java:4)
```

- If an error occurs, we can use try...catch to catch the error and execute some code to handle it:

```
public class Main {
        public static void main(String[ ] args) {
                try {
                        int[] myNumbers = {1, 2, 3};
                        System.out.println(myNumbers[10]);
                }

                catch (Exception e) {
                        System.out.println("Something went wrong.");
                }
        }
}
```

Output:

```
        Something went wrong.
```

- The following is another example of how the try-catch statement can be used in action to prevent the user from trying to divide the value of a variable by 0.

```
public class DivideByZero {

    public static void main(String[] args) {

        int var1 = 5;
        int var2 = 0;    // 0 is assigned to var2 to cause an exception by dividing
                         //var1 by 0
        try {
                int var3 = var1 / var2;  // This is the statement that will cause the
                                         //exception to be thrown
        }

        catch (ArithmeticException e) {
                System.out.println("It is not possible to divide by zero");
        }
    }
}
```

Finally

- The finally statement lets you execute code, after try...catch, regardless of the result:

```
public class Main {

        public static void main(String[] args) {
        try {
                int[] myNumbers = {1, 2, 3};
                System.out.println(myNumbers[10]);
```

```
                    }

                    catch (Exception e) {
                              System.out.println("Something went wrong.");
                    }

                    finally {
                              System.out.println("The 'try catch' is finished.");
                    }
          }
}
```

Output:

```
   Something went wrong.
   The 'try catch' is finished.
```

- Here is an example of a program where the finally block will be executed:

```
public class FinallyBlockExample {

        public static void main(String args[]){

                try {
                        int var = 30 / 6;
                        System.out.println(var);
                }

                catch (NullPointerException e) {

                        System.out.println(e);
                }

                finally {
                        System.out.println(“These are the contents of the finally Block.”);

                }
                System.out.println(“The finally block has been executed.”);
        }
}
```

Output:

```
        5
        These are the contents of the finally block.
        The finally block has been executed.
```

Throw Exception

- Java provides a keyword “throw” using which we can explicitly throw the exceptions in the code.
- The throw keyword is also used to throw custom exceptions.
- Using the throw keyword, we can throw the *checked* or *unchecked* exceptions.

- **Example 1: Throwing Unchecked Exception**
    - In this example, created a method named validate() that accepts an integer as a parameter.
    - If the age is less than 18, then throwing the ArithmeticException otherwise print a message welcome to vote.

```
public class TestThrow1 {

        //function to check if person is eligible to vote or not
        public static void validate(int age) {
                if(age<18) {
                        //throw Arithmetic exception if not eligible to vote
                        throw new ArithmeticException("Person is not eligible to vote");
                }
                else {
                        System.out.println("Person is eligible to vote!!");
                }
        }

        /main method
        public static void main(String args[]){

                //calling the function
                //validate(20);

                //calling the function
                validate(15);

                System.out.println("rest of the code...");
        }
}
```

Output:

```
java -cp /tmp/LVYC3IYyB4/TestThrow1
ERROR!
```

```
Exception in thread "main" java.lang.ArithmeticException: Person is not eligible
to vote
        at TestThrow1.validate(TestThrow1.java:6)
        at TestThrow1.main(TestThrow1.java:18)
```

- **Example 2: Throwing Checked Exception**

```
public class TestThrow2 {

        //function to check if person is eligible to vote or not
        public static void method() throws FileNotFoundException {

                FileReader file = new FileReader("C:\\Users\\Anurati\\Desktop\\abc.txt");
                BufferedReader fileInput = new BufferedReader(file);

                throw new FileNotFoundException();
        }

        //main method
        public static void main(String args[]){

                try {
                        method();
                }

                catch (FileNotFoundException e) {
                        e.printStackTrace();
                }

                System.out.println("rest of the code...");
        }
}
```

Output:

```
java -cp /tmp/ZmS1h2Y8tM/TestThrow2
java.io.FileNotFoundException: C:\Users\Anurati\Desktop\abc.txt (No such file
or directory)
        at java.base/java.io.FileInputStream.open0(Native Method)
        at java.base/java.io.FileInputStream.open(FileInputStream.java:219)
        at java.base/java.io.FileInputStream.<init>(FileInputStream.java:157)
        at java.base/java.io.FileInputStream.<init>(FileInputStream.java:112)
        at java.base/java.io.FileReader.<init>(FileReader.java:60)
        at TestThrow2.method(TestThrow2.java:8)
        at TestThrow2.main(TestThrow2.java:19)
rest of the code...
```