BACHELOR OF COMPUTER APPLICATIONS

SEMESTER 4

DCA2202

JAVA PROGRAMMING

Unit 6

Exception Handling

Table of Contents

1. INTRODUCTION

In the last unit, we have seen the implementation of inheritance, packages and interfaces in Java. In this unit, we will discuss about exception handling. Even though every programmer tries to write an error free program but they seldom succeed. It's because programmer often can't foresee the situation he may encounter. Errors can be runtime, programming error, invalid input, faulty hardware devices and so on. Such situations must be handled and possible remedies could be:

- Notifying the user about the error that has occurred.
- Save the work environment.
- The program could be exited without effecting other program in running at that moment.

This unit aims at providing information about handling situations occurring due to error or faulty conditions. This unit explains the mechanisms available in Java to handle such situations. The error or exception handling mechanisms in Java enables programmers to write more robust and secure codes. Java enforces the programmers to specify the exceptions that might occur while executing a method.

1.1 Objectives

After studying this unit, you should be able to:

- *Define Exception in Java*
- *List various Exception Classes*
- *Describe some common exceptions in Java*
- *Explain various exception handling techniques s*

2. DEFINITION OF AN EXCEPTION

The term **exception** in Java means the occurrence of an exceptional event. It is defined as an abnormality that occurs while the program is in execution which hinders the successful execution of the program, which means the way the program is meant to produce the result it doesn't do so.

When you are developing apps Error Handling is very necessary. The abnormalities or unexpected situations that you can encounter while the execution of programs are:

- Memory shortage.
- Errors in resource allocation.
- Not able to locate a file.
- Error in network connectivity.

If any of the above-mentioned errors are encountered the program stops executing. As a programmer, you cannot expect your app to stop working or crash in between execution. Generally, programmers use return values of methods to find an error that has occurred during runtime. A variable named ***errno*** is used to represent the error numerically and in case of multiple errors, the mentioned variable retains the value of the last error that has occurred.

In Java Object-Oriented way is employed to handle. A hierarchy of exception classes is used to manage runtime errors.

Self-Assessment Questions - 1

1. The term exception denotes a/an_____.
2. Java handles exceptions in the_____way.

3. EXCEPTION CLASSES

The topmost class in the hierarchy is a **Throwable** class. Two classes are further derived from it – **Error** and **Exception**. The **Exception** class traps the exception that occurs in program and **Error** class defines the condition that doesn't occur as it should have under normal condition. In other words, the **Error** classis used for catastrophic failures such as ***VirtualMachineError***. These classes are available in the ***java.lang*** package.

Self-Assessment Questions - 2

3. The class at the top of the exception classes hierarchy is _____.
4. _____ and _____ classes are derived from Throwable class.
5. Exception classes are available in _____ package.

4. COMMON EXCEPTIONS

There are several predefined exceptions in Java, a few of them are mentioned below:

- **Arithmetic Exception**

  When exceptional arithmetic condition occurs for example division by zero then this exception is thrown.

- **ArrayIndexOutOfBounds Exception**

  When an attempt is made to access an element of an array which falls beyond the mention the index; for example your array is of size 10 and you are trying to access an element which is at position fifteen, then this exception is thrown.

- **NullPointer Exception**

  When an application tries to return null when some object is required then this exception occurs. Null values are allocated to object that doesn't have been allocated memory. Different situation under which this exception occurs are:

  - An object is used that has not been allocated memory.
  - Methods of null object are called.
  - Trying to access or modify the attributes of null objects.

Self-Assessment Questions - 3

6. _____ exception will be thrown when you divide by zero.
7. _____ exception will be thrown if you try to access the array element beyond its index value.

5. EXCEPTION HANDLING TECHNIQUES

Java creates an object of appropriate exception class whenever there a error occurs in your method. Once the exception objects are created it is passed to your program. This is called throwing an exception. These exception object have the information about the present state of the program and the type of exception that has occurred. You need to handle the exception using exception-handler and process the exception.

You can implement exception-handling in your program by using following keywords:

- try
- catch
- finally
- throw/throws

The try Block

You need to guard the statements that may throw an exception in the ***try*** block. The following skeletal code illustrates the use of the try block.

```
try
{
        // statement that may cause an exception
}
```

The try block governs the statements that are enclosed within it and defines the scope of the exception handlers associated with it. In other words, if an exception occurs within try block, the appropriate exception-handler that is associated with the try block handles the exception. A try block must have at least one catch block that follow it immediately.

Nested try Statements

You can nest a **try** statement; i.e. you can use a **try** statement within another. Every time **try** statement is executed the information is pushed into a stack. If a **try** statement does not have a **catch** statement then the successive **try** statement's **catch** is checked and it keeps going unless a proper match is formed or all the **try** statements are exhausted. Below example shows the use of **try** statement:

Example:

```
// An example of nested try statements.
class NestTry {
	public static void main(String args[]) {
		try {
			int a = args.length;
			/*
			 * If no command-line args are present, the
			 * following statement will
			 * generate a divide-by-zero exception.
			 */
			int b = 42 / a;
			System.out.println("a = " + a);
			try {
				// nested try block
				/*
				 * If one command-line arg is used, then
				 * a divide-by-zero exception will be
				 * generated by the following code.
				 */ if (a == 1)
					a = a / (a - a); // division by zero
				/*
				 * If two command-line args are used,
				 * then generate an
				 * out-of-bounds exception.
				 */

				if (a == 2) {
					int c[] = { 1 };
					c[42] = 99;
					// generate an out-of-bounds exception
				}
			} catch (ArrayIndexOutOfBoundsException e) {
			}
		} catch (ArithmeticException e) {
			System.out.println("Divide by 0: " + e);
		}
	}
}
```

The above example nests one **try** blocks within another. The above program works as follows: A divide by zero exception is generated by the outer try block when you execute the program with no command line. Execution of the program by one command-line argument generates a divide-by-zero exception from within the nested **try** block. The exception is passed to the outer block by the inner block as it does not catch the exception. If you execute the program with two command-line arguments, an array boundary exception is generated from within the inner **try** block. Here are sample runs that illustrate each case:

Output:

1. java NestTry

   Divide by 0: java.lang.ArithmeticException: / by zero

2. java NestTry One

   a = 1

   Divide by 0: java.lang.ArithmeticException: / by zero

3. java NestTry One Two

   a = 2

   Array index out-of-bounds:

   java.lang.ArrayIndexOutOfBoundsException: 42

Nesting of **try** statements can occur in less obvious ways when method calls are involved. For example, you can enclose a call to a method within a **try** block. Inside that method is another **try** statement. In this case, the **try** within the method is still nested inside the outer **try** block, which calls the method. Here is the previous program recoded so that the nested **try** block is moved inside the method **nesttry( )**:

```
/* Try statements can be implicitly nested via calls to methods. */
class MethNestTry {
	static void nesttry(int a) {
		try {
			// nested try block
			/*
			 * If one command-line arg is used,
			 * then a divide-by-zero exception
			 * will be generated by the following code.
			 */ if (a == 1)
				a = a / (a - a);
			// division by zero
			/*
			 * If two command-line args are used,
			 * then generate an out-of-bounds
			 * exception.
			 */ if (a == 2) {
				int c[] = { 1 };
				c[42] = 99; // generate an out-of-bounds exception
			}
		} catch (ArrayIndexOutOfBoundsException e) {
			System.out.println("Array index out-of-bounds: " + e);
		}
	}
	public static void main(String args[]) {
		try {
			int a = args.length;
```

```
                    /*
                     * If no command-line args are present,
                     * the following statement will
                     * generate a divide-by-zero exception.
                     */ int b = 42 / a;
                    System.out.println("a = " + a);
                    nesttry(a);
                } catch (ArithmeticException e) {
                    System.out.println("Divide by 0: " + e);
                }
        }
}
```

The output of this program is identical to that of the preceding example.

The catch Block

You associate an exception-handler with the try block by providing one or more catch handlers immediately after try block. The following skeletal code illustrates the use of the catch block.

```
try
{
    //statements that may cause an exception
}
catch ()
{
    // error handling code
}
```

The catch statement takes an object of an exception class as a parameter. If an exception is thrown, the statements in the catch block are executed. The scope of the catch block is restricted to the statements in the preceding try block only.

The finally Block

When an exception is raised, the rest of the statements in the try block are ignored. Sometimes, it is necessary to process certain statements irrespective of whether an exception is raised or not. The finally block is used for this purpose.

```
try
{
    openFile();
    writeFile(); //may cause an exception
```

```
}
catch (...)
{
        //process the exception
}
```

If for example, the file has to be closed irrespective of whether an exception is raised or not. You can place the code to close the file in both the try and catch blocks. To avoid duplication of code, you can place the code in the **finally** block. The code in the **finally** block is executed regardless of whether an exception is thrown or not. The **finally** block follows the catch blocks. You have only one finally block for an exception-handler. However, it is not mandatory to have a finally block.

```
finally {
        closeFile ();
        }
```

6. THROW AND THROWS

As mentioned in the previous segment, Java libraries throw various exceptions based on the condition encountered. For example, if the argument received is null, the java library method can throw a 'NullPointerException'. These exceptions can be thrown by the code we write as well.

For example, if we are validating the inputs received for a customer, if the inputs are empty, we can explicitly **throw** a NullPointer exception. However, we will have to mention this using the keyword '**throws'**.

7. CUSTOM EXCEPTION

Very often, there is a need to create custom exception classes which is specific to the application that you are writing. It could be for debugging purpose or for handling some unique conditions in a special way. Java provides ways to create such custom exception classes , 'throw' them based on some conditions, 'catch' them and handle it in a specific way.

For example, if you are writing a program to handle the characteristics of a customer and we are checking for the Age , We will have to check if Age is a valid and if not throw an exception, say, InvalidAgeException.

This new class of exception can be created like any other ordinary class. However, this class will have to extend one of the below classes

1. java.lang.Throwable
2. java.lang.Exception
3. java.lang.RuntimeException

Methods using classes extending RuntimeException need not explicitly mention the same. Neither does the compiler check the handling. However, these should only be used in exceptional cases.

Most common way of creating custom exception classes is by extending java.lang.Exception classes. This is a 'checked' exception. This means that the code using the methods that 'throws' this exception is expected to handle the same. Else, the compilation will result in errors.

Besides extending java.lang.Exception, No other methods needs to be defined within these custom exception classes. However, it is a good practice to create constructor methods within it that displays additional information.

Given below is a typical example to get the age and quantity purchased by a customer. We can get the inputs and validate the same. Depending on the type of input that is an invalid entry, different type of exceptions can be thrown and appropriate message can be displayed.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

// Custom exception classes can be created
// by extending the class exception
class InvalidAgeException extends Exception {
	// It is a good practise to override
	// the default constructor to provide some
	// additional messages.
	InvalidAgeException() {
		System.out.println("Invalid Age encountered");
	}
}

class InvalidQuantityException extends Exception {
	InvalidQuantityException() {
		System.out.println("Invalid Quantity encountered");
	}
}

public class CustomException {

	// The method that throws the custom exception
	// should explicitly state them. In the below
	// example the method 'validate' throws exception
	// InvalidAgeException and InvalidQuantityException
	public static void validate(int age, double quantity)
		 throws InvalidAgeException, I
		 nvalidQuantityException {
		if (age <= 0) {
			// throw the specific custom exception
			throw new InvalidAgeException();
		}
		if (quantity <= 0) {
			// throw the specific custom exception
			throw new InvalidQuantityException();
		}
	}
	public static int getAge() {
		Integer age = 0;
		try {
			BufferedReader reader = new BufferedReader(  new
			InputStreamReader(System.in));
			System.out.println("Enter Age:");
			age = new Integer(reader.readLine());
		} catch (Exception e) {
			e.printStackTrace();
		}
		return age;
	}
	public static double getQuantity() {
		Double quantity = 0.0;
		try {
			BufferedReader reader = new BufferedReader(
				new InputStreamReader(System.in));
			System.out.println("Enter Quantity:");
			quantity = new Double(reader.readLine());
```

```
			} catch (Exception e) {
				e.printStackTrace();
			}
			return quantity;
		}
		public static void main(String args[]) {
			boolean valid_inputs = false;
			int age = 0;
			double quantity = 0;
			// Get inputs and validate.
			// Continue till all the inputs are valid
			while (valid_inputs == false) {
				try {
					valid_inputs = true;
					age = getAge();
					quantity = getQuantity();
					validate(age, quantity);
				} catch (Exception e) {
					valid_inputs = false;
					System.out.println("Re-enter inputs!");
				}
			}
			System.out.println("Inputs valid!");
		}
}
```

Output:

Enter Age:

-12

Enter Quantity:

3

Invalid Age encountered

Re-enter inputs!

Enter Age:

13

Enter Quantity:

0

Invalid Quantity encountered

Re-enter inputs!

Enter Age:

13

Enter Quantity:

3

Inputs valid!

8. SUMMARY

The term exception denotes an exceptional event. It can be defined as an abnormal event that occurs during program execution and disrupts the normal flow of instructions. The Exception class is used for the exceptional conditions that have to be trapped in a program.

Java has several predefined exceptions. The most common exceptions that you may encounter are:

- Arithmetic Exception
- NullPointer Exception
- ArrayIndexOutOfBounds Exception

The following keywords are used for exception-handlings:

- try
- catch
- finally
- throw/throws

If an exception occurs within the try block, the appropriate exception-handler that it associated with the try block handles the exception.

You associate an exception-handler with the try block by providing one or more catch handlers immediately after the try block.

Sometimes, it is necessary to process certain statements, no matter whether an exception is raised or not. The **finally** block is used for this purpose.

Custom Exception Classes can be created by extending the java.lang.Exception class. This can be thrown based on some specific conditions. The method that throws this exception should explicitly mention it in the signature using the keyword -'throws'.

9. TERMINAL QUESTIONS

1. What is the difference between errors and exceptions?
2. What are the different types of Exception?
3. What are the different Exception handling techniques?
4. How do we create custom exception classes. Provide Example.

10. ANSWERS

Self-Assessment Questions

1. An exceptional event.
2. Object oriented.
3. Throwable.
4. Error and Exception.
5. java.lang
6. Arithmetic Exception.
7. ArrayIndexOutOfBounds.

Terminal Questions

1. The term exception denotes an exceptional event. It can be defined as an abnormal event that occurs during program execution and disrupts the normal flow of instruction. (Refer section 2)
2. Arithmetic, NullPointer, ArrayIndexOutOfBounds. (Refer section 4)
3. try, catch, finally. (Refer section 5)
4. Refer section 7