Chapter 5
Exception Handling

1. Introduction

An exception is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a run-time error. Exception handling is one of the powerful mechanisms to handle the exceptions so that the normal flow of the application can be maintained. For example, while doing the arithmetic operation if we get the situation when some number is divided by zero then it is an exception and if we do not handle this exception, the rest of the code after the exception will not be executed.

```
import java.util.Scanner;
public class ExceptionTest
{
   public static void main(String[] args)
   {
      int dividend, divisor, quotient;
      Scanner sc = new Scanner(System.in);
      System.out.println("Enter dididend and divisor:");
      dividend = sc.nextInt();
      divisor = sc.nextInt(); // run the program with divisor 0
      quotient = dividend / divisor;
      System.out.println("Quotient = " + quotient);
      System.out.println("The End");
   }
}
```

2. Exception Handling Fundamentals

When an exception occurs within a method, the method creates an **exception object** that contains information about the exception, including its type and state of the program when the exception occurred. The object is then **thrown** in the method that caused the exception. This is called **throwing an exception**. The method may choose to handle the exception itself or pass it on. The block of code that is used to handle the exception is called **exception handler**. The appropriate exception handler chosen is called **catch the exception**.

In Java, exception handling is managed by the use of five keywords: **try, catch, throw, throws,** and **finally**. Program structures that you want to monitor for exceptions are contained within a **try** block. If an error occurs within a try block, it is thrown. Your code can catch this exception using **catch** and handle it. System generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword **throw**. Any exception that is thrown out of a method must be specified by **throws** clause. Any code that absolutely must be executed before a method returns is put into a **finally** block. The general form of exception handling block is given below.

```
try
{
   //block of code to monitor for errors
}
catch(ExceptionType1 exOb)
{
   //exception handler for exception type 1
}
catch(ExceptionType2 exOb)
{
   //exception handler for exception type 2
}
.....
catch(ExceptionTypeN exOb)
{
   //exception handler for exception type N
}
finally
{
   //block of codes for finally block
}
```

3. Using try and catch

Although the default exception handler provided by the Java run-time system is useful for debugging, you will usually want to handle an exception yourself. Doing so provides two benefits. First, it allows you to fix the exception. Second, it prevents the program from automatically terminating. To guard against and handle an exception, simply enclose the code that you want to monitor inside a **try** block. Immediately following the try block, include a catch **clause** that specifies the exception type that you wish to catch. For example, the following program includes a try block and a catch clause that processes the ArithmeticException generated by the division-by-zero error.

```
import java.util.Scanner;
public class ExceptionTest
{
   public static void main(String[] args)
   {
      int dividend, divisor, quotient;
      Scanner sc = new Scanner(System.in);
      System.out.println("Enter dididend and divisor:");
      dividend = sc.nextInt();
      divisor = sc.nextInt(); // run the program with divisor = 0
      try
      {
         quotient = dividend / divisor;
         System.out.println("Quotient = " + quotient);
      }
      catch(ArithmeticException e)
      {
            System.out.println(e);
        }
        System.out.println("The End");
    }
}
```

4. Multiple catch Clauses

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more catch clauses, each catching a different type of exception. When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed. After one catch statement executes, the others are bypassed, and execution continues after the try / catch block. For example,

```
import java.util.Scanner;
import java.util.InputMismatchException;
public class ExceptionTest
{
	public static void main(String[] args)
	{
		int dividend, divisor, quotient;
		Scanner sc = new Scanner(System.in);
		System.out.println("Enter divdidend and divisor:");
		try
		{
			dividend = sc.nextInt();
			divisor = sc.nextInt(); // run the program with divisor 0
			quotient = dividend / divisor;
			System.out.println("Quotient = " + quotient);
		}
		catch(ArithmeticException ae)
		{
			System.out.println(ae);
		}
		catch(InputMismatchException ime)
		{
			System.out.println(ime);
		}
		System.out.println("The End");
	}
}
```

When you use multiple catch statements, it is important to remember that exception subclasses must come before any of their superclasses. This is because a catch statement that uses a superclass will catch exceptions of that type plus any of its subclasses. Thus, a subclass would never be reached if it came after its superclass.

Note: We can also use multicatch instead of using multiple catch blocks as given below.

```
catch(ArithmeticException |InputMismatchException e)
{
	System.out.println(e);
}
```

5. Nested try

We can put the **try** statement inside another **try** statement. If the inner try statement does not have the catch handler for the particular exception then the program try to use the catch statement of the outer try statement. This process continues for any level of nesting. If no catch statement is matched then the java run-time system handles the exception. For example,

```
public class ExceptionTest
{
	public static void main(String []args)
	{
		try
		{
			int argnum = args.length;
			System.out.println("Number of arguments = " + argnum);
			int quot = 33/argnum;
			try
			{
				if (argnum == 1)
					argnum = 34/(argnum-argnum);
				if(argnum == 2)
				{
					int c[] = {1, 2, 3};
					c[22] = 34;
				}
			}
			catch(ArithmeticException e)
			{
			System.out.println("Divide by zero: " + e);
			}
		}
```

```
			catch(ArrayIndexOutOfBoundsException e)
			{
				System.out.println("Array index out of bounds: " + e);
			}
		}
}
```

Note: Run this program with different number of command line arguments including no command line argument.

6. The *throw* Keyword

So far, we have only been catching exceptions that are thrown by the Java runtime system. However, it is possible for your program to throw an exception explicitly, using the throw statement. The general form of throw is shown here:

throw ThrowableInstance;

Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable. There are two ways you can obtain a Throwable object: using a parameter in a catch clause or creating one with the new operator. The flow of execution stops immediately after the throw statement; any subsequent statements are not executed. The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on. If no matching catch is found, then the default exception handler halts the program and prints the stack trace. Here is a sample program that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler. For example,

```
public class ExceptionTest
{
   public static void throwDemo()
   {
      try
      {
         throw new NullPointerException("This is demo");
      }
      catch(NullPointerException e)
      {
         System.out.println("Caught inside throwDemo method");
         throw e; //rethrow the exception
      }
   }
   public static void main(String []args)
   {
      try
      {
         throwDemo();
      }
      catch(NullPointerException e)
      {
         System.out.println("Rethrown: Caught inside Main method " + e);
      }
   }
}
```

7. The *throws* Keyword

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You do this by including a throws clause in the method's declaration. A throws clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses. All other exceptions that a method can throw must be declared in the throws clause. If they are not, a compile-time error will result. The general form of a method declaration that includes a throws clause is given below.

```
[access] [modifier] type methodName(paramlist) throws commaseparatedexceptionslist
{
         //method body
}
```

For example,

```
public class ExceptionTest
{
   public static void throwsDemo(int x) throws ClassNotFoundException,
InterruptedException
   {
      if(x < 100)
      {
         System.out.println("I am Inside throwsDemo with integer < 100" );
         throw new ClassNotFoundException("Class not found");
      }
      else
      {
         System.out.println("I am Inside throwsDemo with integer >= 100" );
         throw new InterruptedException("Interrupted");
      }
   }
   public static void main(String[]args) throws InterruptedException
   {
      try
      {
         throwsDemo(5);
      }
      catch(ClassNotFoundException e)
      {
         System.out.println("Caught " + e);
      }
      try
      {
         throwsDemo(500);
      }
      catch(ClassNotFoundException e)
      {
         System.out.println("Caught " + e);
      }
   }
}
```

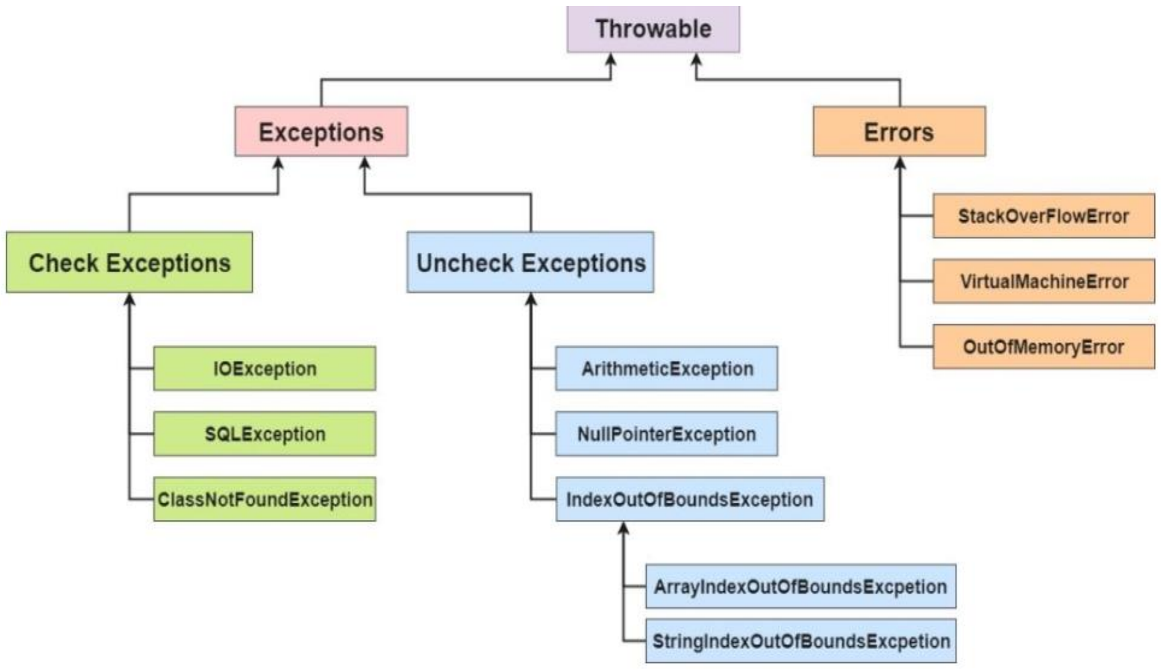
8. The *finally* Keyword

The **finally** keyword creates a block of code that will be executed after a try /catch block has completed and before the code following the try/catch block. The finally block will execute whether or not an exception is thrown. If an exception is thrown, the finally block will execute even if no catch statement matches the exception. Any time a method is about to return to the caller from inside a try/catch block, via an uncaught exception or an explicit return statement, the finally clause is also executed just before the method returns. This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning. The finally clause is optional. However, each try statement requires at least one catch or a finally clause. For example,

```
import java.util.Scanner;
public class ExceptionTest
{
   public static void main(String[] args)
   {
      int dividend, divisor, quotient;
      Scanner sc = new Scanner(System.in);
      System.out.println("Enter dididend and divisor:");
      dividend = sc.nextInt();
      divisor = sc.nextInt();
      try
      {
         quotient = dividend / divisor;
         System.out.println("Quotient = " + quotient);
      }
      catch(ArithmeticException e)
      {
         System.out.println(e);
      }
      finally //finally block
      {
            System.out.println("This statement will always execute");
        }
        System.out.println("The End");
    }
}
```

9. Java’s Built-in Exceptions

Java defines several exception classes inside **java.lang** package. The most general of these exceptions are subclasses of the RuntimeException class. These exceptions are called **unchecked exceptions** because the compiler does not check to see if a method handles or throws these exceptions. These exceptions need not be included in any method’s throws list. **Checked exceptions** must be included in a method’s throws list if that method can generate one of these exceptions and does not handle it itself. In addition to the exceptions in java.lang, Java also defines several more exceptions that relate to its other standard packages. Some common exceptions in java.lang package are listed below.



9.1. Unchecked Exceptions

- **ArithmeticException:** Arithmetic error like divide by zero.
- **ArrayIndexOutOfBoundsException:** Array index is out of bounds.
- **ArrayStoreException:** Assignment to an array element of an incompatible type.
- **ClassCastException:** Invalid class cast.
- **EnumConstantNotPresentException:** An attempt is made to use an undefined enumeration value.
- **IllegalArgumentException:** Illegal argument used to invoke a method.
- **IllegalCallerException:** A method cannot be legally executed by the calling code.
- **IllegalMonitorStateException:** Illegal monitor operation, such as waiting on an

unlocked thread.

- **IllegalStateException:** Environment or application is in incorrect state.
- **IllegalThreadStateException:** Requested operation not compatible with current thread state.
- **IndexOutOfBoundsException:** Some types of index are out of bounds.
- **LayerInstantiationException:** A module layer cannot be created.
- **NegativeArraySizeException:** Array created with negative size.
- **NullPointerException:** Invalid use of null reference.
- **NumberFormatException:** Invalid conversion of a string to a numeric format.
- **SecurityException:** Attempt to violate security.
- **StringIndexOutOfBoundsException:** Attempt to index outside the bounds of a string.
- **TypeNotPresentException:** Type not found.
- **UnsupportedOperationException:** An unsupported operation was encountered.

9.2. Checked Exceptions

- **ClassNotFoundException**: Class not found.
- **CloneNotSupportedException:** Attempt to clone (duplicate) an object that does not implement the **Cloneable** interface.
- **IllegalAccessException:** Access to a class is denied.
- **InstantiationException:** Attempt to create an object of an abstract class or interface.
- **InterruptedException:** One thread has been interrupted by another thread.
- **NoSuchFieldException:** A requested field does not exist.
- **NoSuchMethodException:** A requested method does not exist.

10. User Defined Exceptions

Although Java’s built-in exceptions handle most common exceptions, you will probably want to create your own exception types to handle situations specific to your applications. This is quite easy to do: just define a subclass of Exception class. For example,

```
class InvalidAgeException extends Exception
{
   public InvalidAgeException (String str)
   {
      super(str);
   }
}
public class ExceptionTest
{
   public static void validate (int age) throws InvalidAgeException
   {
      if(age < 1 || age > 150)
      {
         throw new InvalidAgeException("Not a valid age.");
      }
```

```
        else
        {
            System.out.println("Valid age.");
        }
    }
    public static void main(String args[])
    {
        try
        {
            validate(151);
        }
        catch (InvalidAgeException e)
        {
            System.out.println("Exception occured: " + e);
        }
    }
}
```

Exercises

1. Why do we need exception handling? Explain.
2. Explain different keywords that are used in exception handling.
3. Compare try with catch.
4. Compare throw with throws.
5. What is the use of finally keyword in exception handling?
6. Differentiate checked exceptions with unchecked exceptions.
7. How do you create your own exceptions in Java?