UNIT 3 ASSERTIONS, ANNOTATIONS AND EXCEPTION HANDLING

3.0 Introduction
3.1 Objectives
3.2 Assertion and its use
3.3 Annotations
    3.3.1 Built-In Java Annotations
3.4 Exception Handling
    3.4.1 Checked and Unchecked Exceptions
    3.4.2 Using try, catch and finally method to handle exception
3.5 Throw and Throws
3.6 Final vs Finally vs Finalize
3.7 User Defined Exception
3.8 Summary
3.9 Solutions/ Answer to Check Your Progress
3.10 References/Further Reading

3.0 INTRODUCTION

Java is known for its robustness, which comes from its strong exceptions handling feature. In this unit we will discuss Assertions, Annotations and Exception Handling. Assertions are infrequently used, but they play a vital role in testing and verifying various assumptions which developers make while writing their code based on the RSD/TSD (Requirement/Technical Specification Document) which they receive from Architect team. First, we will try to understand what benefits usage of assertions brings. The next part in this unit touches upon the usage of annotations, which helps us to maintain the documentation and changes of our code during the life-cycle of our code with respect to the changes that have happened over time in the language itself. In the third and final part of this unit will discuss at length one of the reasons why java has been perceived as such a robust language, as it not only had vast constructs to handle normal flow of program but also a completely structured methodology to handle even the exceptions which might occur. We Will study Exception Handling in Java, in detail.

At the end of this unit you will be able to define and use annotations in Java. Also you should be able to explain what assertions are ? where are they used and what advantages do they bring in. Then you will go through importance of metadata and significance of writing meaningful comments in your program. This will underline the fact that documentation for any program is very important. It is so important that a language like java had to introduce a new construct which will help it easily and effectively maintain documentation using annotations. Also you will learn about exception handling in Java. We will go through various types of exceptions, the exceptions hierarchy, distinguish between different types of exceptions and study in detail about Runtime and IOExceptions. You will appreciate to learn that how apt and wide coverage of exceptions is provided in java. Towards the end you will learn to define your own Exception.

3.1 OBJECTIVES

After going through this unit we will be able to :

- Define assertions,
- Differentiate Assertions and Exception Handling,
- Explain usage and Advantages of Assertions,
- Describe Annotations, Annotation types and Metadata,
- Use Annotations,
- Explain Exception handling,
- Differentiate between errors and exceptions,
- Describe Exception hierarchy,
- Describe types of Exceptions,
- Difference between throw and throws,
- Use final finally and finalize keywords, and
- Define your own exception methods.

3.2 ASSERTIONS AND ITS USE

We all would have heard about the role of a proofreaders in publishing industry, the main objective of proofreading a document is to ensure that none of the mistakes/ undesired statements actually reach the print or the printing stage. Effectively this is the task in which assertions help us perform over our programs . Now let us understand how we use assertions in Java.

The meaning of the term assert stated in dictionary is to **'declare positively'**. But then, is not it all that we do when we write our programs? Then what is the need to have a specific statement for this purpose and if it is there, how should we use it ? What is the applicability and, even more importantly, where we *should not* apply it? We will also study the difference between using an assertion statement versus a more commonly known procedure of **exception handling**.

Let's start with a question. How often do you write a code that will not go to production or final use? well, companies hire and pay programmers to write code they can use in production, isn't it? Then why will anybody write a code that will not go to production and, more importantly, gets paid for something that will not go to production? It sounds dubious isn't it?

Well, let's try and understand this using another simple situation. During the normal course of execution of a program if a piece of code or condition is *unreachable* it should be tested beforehand. This is where an *assert a statement* comes into picture. Effectively the Java assert statement is meant to be used in **non production environment.** In fact, it is one of those rare statements which are specifically made for usage in non-production environments only, but how? To be more precise assertions are by default disabled in any environment and have to be ***explicitly enabled***, in order for them, to be used. In other words, to utilize assertions in any environment you need to enable it, and in production, as a recommended practice, we do not enable assertions. Simply put, when assertions are enabled they can be utilized to detect code issues i.e. only if there is a bug in a program then only an assert statement will be triggered.

So only an occurrence of an **extraordinary condition** will lead Java assert to trigger, resulting in an **AssertionError** to be thrown. The execution of AssertionError will thus prevent a Java application to reach an otherwise unreachable code and ***terminate abruptly.*** As an add-on, it also provides the provision to display a meaningful message if the extraordinary condition has been encountered in Java application.

So why is Java assert so different or unique? To answer this, first, let us understand these two points :

1. Firstly, the use of *assert keyword* is handy in a test environment, or for **temporarily** performing a difficult debugging routine on production systems (though debugging and troubleshooting your code in production is not a recommended approach to take). There is no other keyword in Java that is specifically targeted towards testing and debugging the Java code.
2. Secondly, the Java assert keyword is ignored by default (not processed when the Java virtual machine(JVM) is run using its default configuration).

To use assertion a special flag namely **enable assertions** is passed as a command line parameter at runtime to the JVM, which signifies the code associated with the Java assert to be executed. Uniquely enough, no other keyword in Java language is disabled by default at runtime except our assert.

To learn these topics, we will take a hands-on approach for:

- Enabling assertions in Java
- Disabling assertions

Also, you need to learn:

- Where to utilize assertion and
- Where not to utilize assertion

Lets take the situation below and see how we can enable and disable assertions in java.
We make a lot of assumptions while writing our programs, a Java assertion helps us to check whether the assumptions which we made during the writing of our code, were correct or not. It sounds like we are doing some sort of testing, testing of our own code. Well, you got it exactly right, Java assert is actually a helpful tool that is used for troubleshooting applications during and the development and testing phase of SDLC. The main purpose of using assertions is to test for conditions that should never be evaluated during the normal course of execution of our code. In case if such conditions did occur, an error message should be generated and the application terminated after throwing an AssertionError.
There are two different syntax usage of Java asserts statement either with a single Boolean articulation or boolean expression and execution statement thereof.

- Assert expression;
- Assert expression1: expression2;

where:

- Expression1 is a boolean expression.
- Expression2 is an expression that can either be a simple statement or has some value (barring invocation of a method that is declared void.)

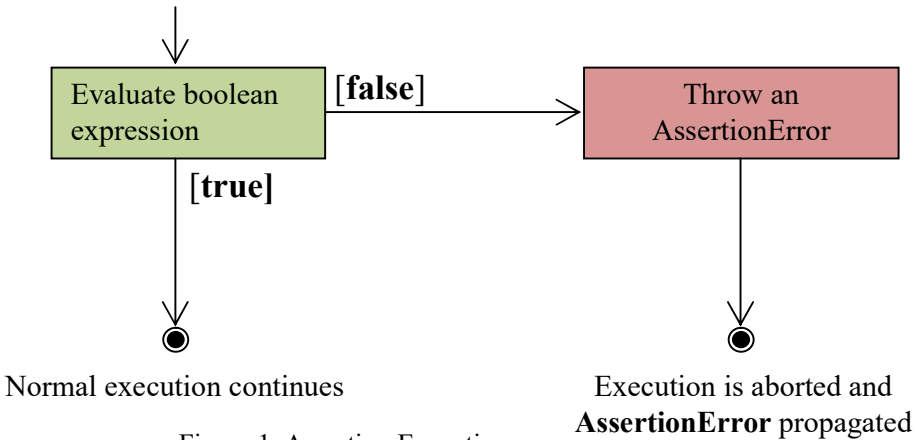


Figure 1: Assertion Execution

Following is the syntax used for a Java assert :
assert : < boolean condition > : < string message for logging>

Another variation and more meaningful usage syntax is Java assert, which will trigger and log/display/print a string message to the Java console as follows:

assert : (true == true) : " OOPS!!! Something bad just happened.";

Let us take a situation in which if a person going in for RTPCR test for testing of Covid19 will be declared positive or negative based on the Cycle Threshold(CT) value. If the CT value is greater than 29(any random value not medically correct), the person will be declared as Covid positive.

Below is a simple example :

```
class CovidTest
{
        public static void main( String args[] )
        {
                int RTPCRvalue = 31;
                assert value >= 29 : " Covid-19 Positive";
                System.out.println("value is "+ RTPCRvalue);
        }
}
```

Compile and run the above Java program. What is the output?

Exception in thread "main" java.lang.AssertionError: Covid-19 Positive
at Test.main(Test.java:8)

If you received the above output that means assertions are enabled on your system and if not, then let us learn how to enable or disable assertions.

Enabling and Disabling Assertions in Java

The syntax for the Enabling assertions is-

```
java –ea Test
```

or

```
java –enableassertions Test
```

The syntax for disabling Java Assert-

```
java –da Test
```

or

```
java –disableassertions Test
```

There are many variations for enabling assertions using the command line.

#1) java –ea
When we issue the above command at command-line, then the assertions are enabled in all classes except for system classes.

#2) java –ea Main
The above command is used to enable assertion for all classes in the Main program.

#3) java –ea DemoClass Main

This command enables assertions for only one class – 'DemoClass' in the Main program.

#4) java –ea com.packageName… Main

The above command enables assertion for package com.packageName and its sub-packages in the Main program.

#5) java –ea … Main
Enables assertion for the unnamed package , which is in the current working directory.

#6) java –esa: arguments OR java –enablesystemassertions: arguments
This command enables assertions for the system classes.

To add the assertions, we have to simply add an assert statement :
Imagine a situation where Bluetooth enabled app on the phone establishes connections with nearby mobile phones, and if a connection is established to a phone of infectious person whose data is updated in app or there are people who are at high risk of COVID-19 (again based on data in app which you're sharing) in your proximity, then you would like it to flash a message "proximity to infection".

```
public void setup_connetion ()
{
        Connection conn = getConnection ();
        assert conn == null;
}
```

The above assert, also can be given differently as shown below:

```
public void setup_connection ()
{
        Connection conn = getConnection ();
        assert conn == null: "Proximity to infection";
}
```

Both of the above code constructs check if the connection returns a null value. If it returns a non-null value, then JVM will throw an error – AssertionError. But you can see, in the second case, a message is provided in the assert statement so that the given message will be used to construct AssertionError. When the second case with assertions enabled, is executed, the exception will look like this:

Exception in thread "main" java.lang.AssertionError: Proximity to infection.

Why Use Java Assertions?

When a programmer needs to check if his/her assumptions are right or not.

- Pass on arguments to private methods. Developers provide private arguments to check their code, and developers may also want to check his/her own assumptions about the arguments.
- Cases that are conditional
- To evaluate conditions at the beginning of any method.

Significance of Assertions on Performance

Developers have an interesting motive in using assertions as well. Consider a possibility when using a conditional statement to detect an issue, developers write these statements and then forgets to remove them from code once their purpose (of

detection of issue) is solved, this might result in a performance bottleneck where several CPU cycles are wasted if they are not removed from code before shipping to production. This (addition and removal of temporary code for debugging) is a tedious task to do and involves multiple teams. To understand this consider a situation where a team is following *extreme programming* principles and programmer 'A' submits his/her code for testing in which he/she utilizes conditional statements to check bugs/assumptions. In case the bugs are identified and corrected or the assumptions are dumbfounded, they will still remain in the code and further released to the next environment and also to production. Unless these are explicitly removed (which is a sort of rework again and thus inefficient), since these lines were only intended for development and testing, once they reach production, they will consume precious processor cycles needlessly as the conditions will still be getting evaluated even after the bug had been removed.

Now compare this to what happens in the case of when the developer uses assertions. As you know that the default behavior of JVM is to ignore and assert statement at runtime the piece of code with assertion statements thus will never get executed and save precious processor cycles. All the assert statements whose purpose was to troubleshoot are thus ignored (remember assertions are disabled by default in every java environment), and the precious Clock cycles are saved.

Now we know that there is a performance benefit for using Java assert statement rather than the programmers writing conditional statements for the same purpose.

Where to Utilize Assertion in Java :

Java Assert can be utilised in the following ways;

- To ensure that an inaccessible looking code is really inaccessible.
- Java Assertions of private techniques. Private contentions are given by designer's code just as a developer might need to check his/her assumptions about contentions.
- In Restrictive cases.
- Conditions toward the start of any technique.
- To verify that the 'default' switch case is not reached.
- Ensure that presumptions written in remarks are correct.
- Check the protest's state.
- At the start of the strategy.
- After strategy summon.

Where not to use Assertions :

- Don't use assertions to replace error messages
- Don't use assertions to check arguments in the public methods as they may be provided by user.
- Don't use assertions on command line arguments.
- You should use Error handling to handle errors where user input is sought instead of Assertions.

Assertions vs Exception Handling

Assertions are primarily used to check logically impossible situations or inconceivable circumstances. For instance, assertions can be used to check the expression or state that a piece of code expects before it starts execution (pre-condition) or a state that it expects at the end or completion of its execution (post-condition). Unlike and the normal exception handling constructs in Java, assertions are generally disabled at runtime.

Two examples of common usage scenarios for assertions

1. Unreachable code
2. Documenting Assumptions

Example :

```
switch (dayOfWeek)
{
  case "Sunday":
    System.out.println("It's Sunday!");
    break;
  case "Monday":
    System.out.println("It's Monday!");
    break;
  case "Tuesday":
    System.out.println("It's Tuesday!");
    break;
  case "Wednesday":
    System.out.println("It's Wednesday!");
    break;
  case "Thursday":
    System.out.println("It's Thursday!");
    break;
  case "Friday":
    System.out.println("It's Friday!");
    break;
  case "Saturday":
    System.out.println("It's Saturday!");
    break;
}
```

The above switch statement indicates that the days of the week can be only one of the above 7 values. Having no default case means that the programmer believes that one of these cases will always be executed , which logically is true in our example, but if an invalid input is received, the assumption might prove wrong. Thus, we try adding a default case to document assumption while still maintaining that the default statement is logically Unreachable for all valid reasons.

```
default:
   assert false: dayofWeek + " is invalid day";
```

If dayOfWeek – the parameter passed to switch-case has a value other than the valid days, an AssertionError is thrown.

☞ Check your progress -1

1. What is the default state of assertions in Java (Enabled or Disabled) ? How can we enable or disable assertions in Java?

.....

.....

.....

.....

2. Does assert throw an exception Java?

.....

.....

.....

.....

3. What happens when an assert fails in Java?

.....

.....

.....

.....

4. What does an assert return in Java?

.....

.....

.....

.....

5. Can we catch the assertion error?

.....

.....

.....

.....

6. How do you assert an exception?

.....

.....

.....

.....

3.3 ANNOTATIONS

Data that describes or holds additional information about your data is called Metadata, and that is what Annotations are. Annotations are a form of metadata. Here in our case, while studying Java, Annotations provide data about a program/code that itself is not part of the program. Since Annotations do not directly impact the functioning of the code, they still serve an essential purpose when embedded in code/program they annotate – they provide documentation to the code.

Though Annotations are a relatively new addition to Java , They make our life very easy in terms of our documentation – without putting too many comments, suggestions, notifying other programmers/users not to use a function etc. Following are the significant uses of annotations, including:

- Providing information to the compiler - The Java compiler uses Annotations to suppress warnings and detect errors.
- Compile time and delivery time handling - We can use Annotations XML files, etc., generated with some software tools.
- Runtime processing - some types of Annotation can be evaluated at runtime.

Let us try to find answer of some frequently asked questions like applying Annotation; where do we use annotations? What types of annotations can we use in Java? Are there predefined types of Annotation? Can we write our own Annotations in Java? What kind of Annotations can be used in combination with pluggable type systems to write robust code with stronger type checking, and how we implements repeated Annotations.

3.3.1 Built-In Java Annotations

There are different types of built in associate annotations in Java. One Annotation category is applied to Java code and the second category annotates the first category.

Built-In Java Annotations applied to Java code :
- @Override
- @SuppressWarnings
- @Deprecated

Built-In Java Annotations Applied to other Java annotations :
- @Target
- @Retention
- @Inherited
- @Documented

Let us understand the built-in annotations first, as they are very simple and straightforward.

@ Override
Inheritance is a fundamental function of any object-oriented programming language that we generally use to infer the properties of the parent unchanged and to override or change some of them. The @Override annotation helps us determine that the derived class method is overriding its main class method. In case it does not, the @Override annotation ensures that a compile-time error occurs.
It can be a silly mistake, such as a spelling mistake, incorrect capitalization (remember that Java is a case-sensitive language), or a simple typo. Therefore, it is recommended to mark the activation override annotation that provides the assurance that a method is overridden.

```
class Animal
{
  void eatSomething()
  {
  System.out.println("eating something");
  }
}
class Dog extends Animal
{
  @Override
  void eatsomething()
  {
  System.out.println("eating foods");
  }//should be eatSomething
}
class TestAnnotation1
{
  public static void main(String args[])
  {
    Animal a=new Dog();
    a.eatSomething();
  }
```

```
}
```

Output when there is a spelling mistake:

Output once the correction is made :


```
class Viral
{
    void testVirus(){System.out.
}

class Covid extends Viral
{
    @Override
    void testvirus(){System.out.
```

```
}

class VerifyAnnotation1
{
     public static void main(String args[])
     {
          Viral v=new Covid();
          v.testVirus();
     }
}
```

Output when there is a spelling mistake:
Replace
Output once the correction is made :

Example 2 :

```
class ViralTest
{
     public void conductTest()
     {
       System.out.println("This is test for presence of Vital Infection.");
     }
}

class covid19Test extends ViralTest
{
    @Override
    public void conductTest()
    {
      System.out.println("This is test to detect presence of Covid19 Infection.");
    }
}

class Main
{
    public static void main(String[] args)
    {
         covid19Test c1 = new covid19Test();
         c1.conductTest();
    }
}
```

Output
In this example, the method conductTest() is present in both the superclass ViralTest and subclass covid19Test. When the method conductTest() is called, the method of the subclass covid19Test is called instead of the method in the superclass ViralTest.

@SuppressWarnings

@SuppressWarnings annotation: This annotation is used for suppressing the warnings issued by the compiler.

```
import java.util.*;
class VerifyAnnotation2
{
  //@SuppressWarnings("This will suppress compiler warnings")
  public static void main(String args[])
  {
```

```
    ArrayList list=new ArrayList();
    list.add("sonoo");
    list.add("vimal");
    list.add("ratan");
    for(Object obj:list)
    System.out.println(obj);
  }
}
```

Output: Comple Time Error

@SuppressWarnings

@SuppressWarnings annotation: is used to suppress warnings issued by the compiler.

```
import java.util.*;
class TestAnnotation2
{
  @SuppressWarnings("This will suppress warnings")
  public static void main(String args[])
  {
    ArrayList list=new ArrayList();
    list.add("Ravi");
    list.add("Vimal");
    list.add("Rameshwar");
    for(Object obj:list)
    System.out.println(obj);
  }
}
```

Now no warning at compile time.

Focus on the statement @SuppressWarnings("This will suppress compiler warnings ") which is an annotation, If you remove it , it will result in a compile time warning as we are using non-generic collection.

@Deprecated

Languages evolve and so does the validity of the keywords and features which they offer some new keywords come to usage and few older ones are @Deprecated. Deprecated means that a feature or a keyword may see sunset and thus can be removed in the future versions/releases of the language. @Deprecated annoation helps us mark the same to a method . Once the compiler detects a deprecated method it prints a warning informing user that it may be removed in the future versions, therefor avoid use of such methods.

Example 1:

```
class Mycode
{
      /*
       -> @deprecated
       -> This method is deprecated and has been replaced by newMethod()
       */
       @Deprecated
```

```
    public static void deprecatedMethod()
    {
        System.out.println("This is a deprecated method");
    }

    public static void main(String args[])
    {
        deprecatedMethod();
    }
}
```

Output :

Deprecated method

Example 2:

```
class A
{
    void m(){System.out.println("hello m");}
    @Deprecated
    void n(){System.out.println("hello n");}
}
class TestAnnotation3
{
    public static void main(String args[])
    {
        A a=new A();
        a.n();
    }
}
```

Now, at the time of compilation :

Note: Test.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.

Where as when you execute/run it :

Output : hello n

Java Custom Annotations

Java Custom annotations or Java User-defined annotations are easy to create and use.
The *@interface* element is used to declare an annotation. For example:

```
1. @interface MyAnnotation{}
```

Here, MyAnnotation is the custom annotation name.

☞ Check Your Progress - 2

1. Describe some basic Annotations from the Standard Library?

---

2. Which Java package is used for Annotations ?

3. Can you create your own annotation – Describe How?

4. Are Annotations inherited in Java? If not, what can be done so that they are inherited?

5. Annotation class is extended from which class in Java?

6. What is the package name for Annotation class?

7. What are meta-annotations? Name a few ?

3.4 EXCEPTION HANDLING

In simple words, “an exception is a problem that arises during the execution of a program”. It can occur for many different reasons to exemplify the few:

- say a user has entered an invalid data or
- in case of a file that needs to be opened cannot be found or
- a network connection that has been lost in the middle of communications or
- the JVM has run out of memory.

Many such exceptional cases may occur while executing Java programs. If we as a programmer do not handle them, it leads to a system failure. So handling an exception is very important, and that is where Java introduces the feature of exception handling so that the normal execution flow will not be abruptly terminated in case of occurrence of exception. Mechanism of exception handling provides an opportunity for a graceful exit from the program in case of the occurrence of an exception. For example, if exception conditions such as ‘Classnotfound’, IOException, SQLexception etc. occur, then by appropriate exception handling, users may be informed about such situations, and program execution can be closed gracefully. Before we learn more about exception handling, let us see the difference between error and exception.

- Errors are impossible to recover, but exception can be recovered by handling them.
- Errors are of type unchecked, but exceptions can be either checked or unchecked. (you will learn checked and unchecked types of exceptions later in this unit)
- Errors are something that happened at runtime but exception can occur or happen either at compile time or runtime.
- Exceptions are caused by the application itself, whereas errors are caused by the environment on which the application is running.

Now let us see what is exception hierarchy in Java. All exceptions and errors types are subclasses of **class Throwable,** which is the base class of the hierarchy. The **Object class** is the parent **class** of all the **classes in Java** by default, and class Throwable is a subclass of Object class. Here one branch is headed by exception, that is, this class is used for exceptional conditions that the user program should catch. For example, NullPointerException, RuntimeException etc. and other branch Error are used by the Java runtime system to indicate the errors that have to do with the runtime environment itself, that is JRE. For example, VirtualMachine error or StackOverflow error etc. Figure 2 given below show the Throwable class and its subclasses.

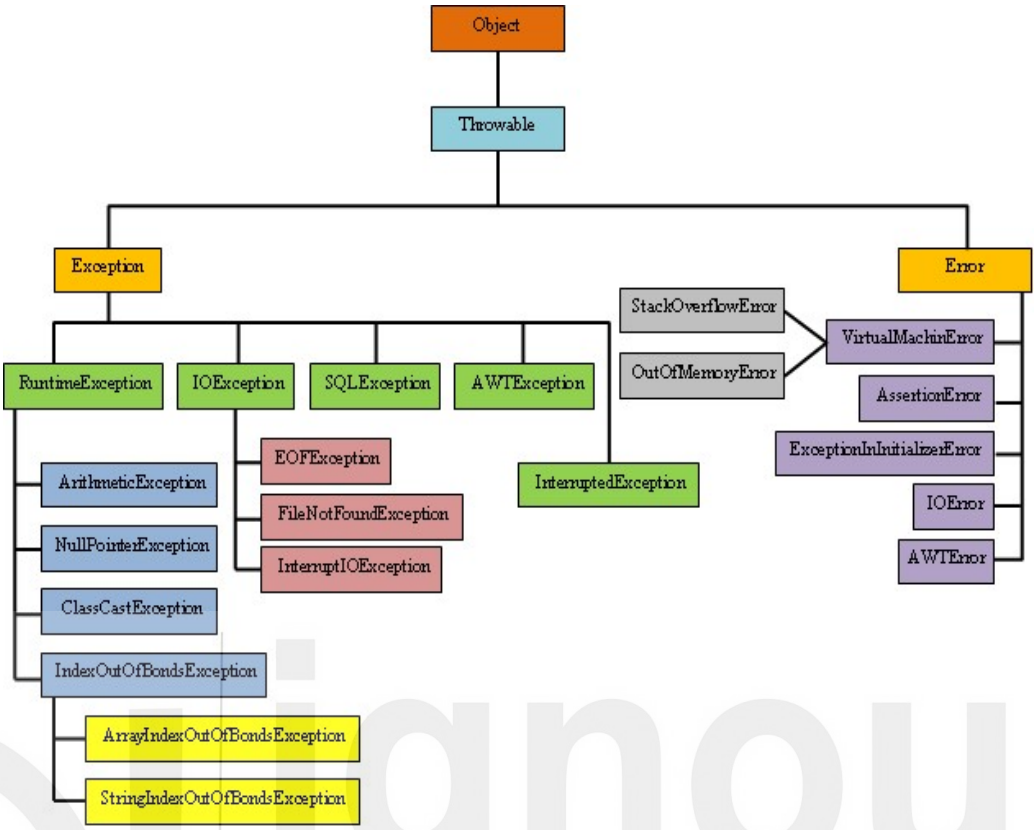


Figure 2: Throwable Class Hierarchy in Java

Exception class and its Subclasses in Throwable class Hierarchy

Here one important and interesting question arises that how Java virtual machine handles exceptions whenever an exception has occurred inside a method? The answer to this question is that the method creates an object known as **exception object** and hands it off to the runtime system. This exception object contains name and description of the exception and also the current state of the program in which the exception has occurred. The process of creating the exception object and handling it to the runtime system is called **'throw'ing** an exception, then by using **try, catch and finally clauses** these exceptions can be handled in Java. This is how Java virtual machine handles exceptions internally.

In figure 3 Exception class and its sub classes are shown. For more details and current updates you can see the current documentation of Java. Having a good understanding of sub classes of exception class is essential for better programming in respect of exceptions handling in Java.

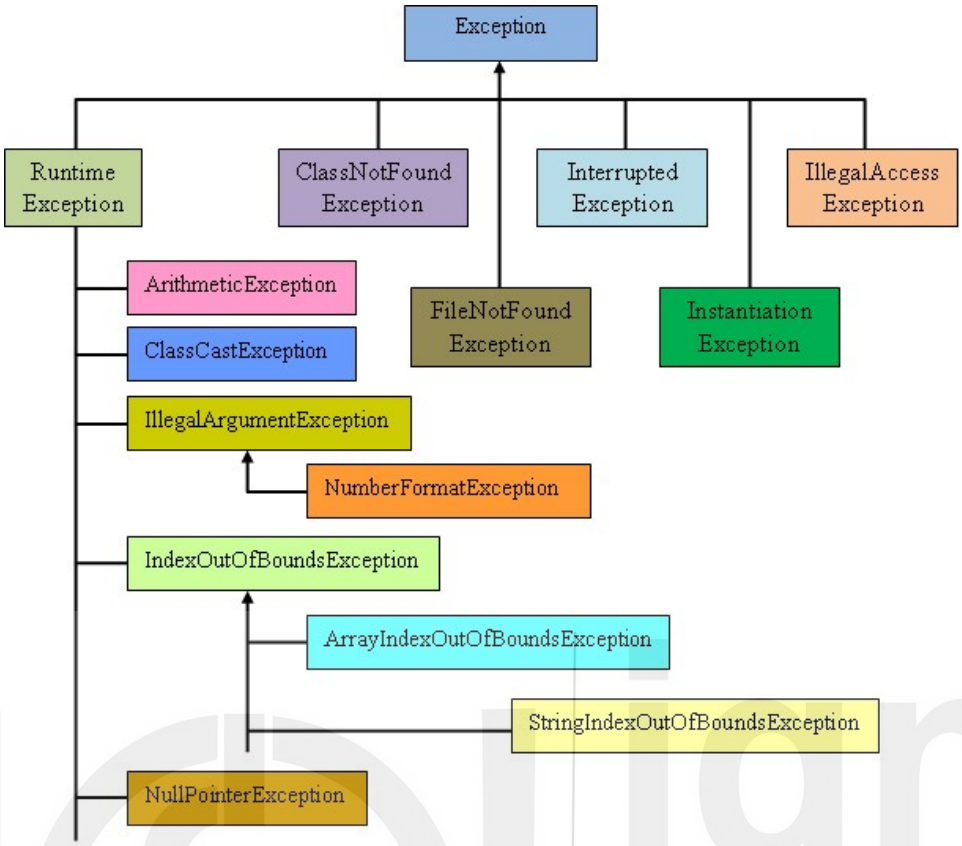


Figure 3: Exception Class and its subclasses

3.4.1 Checked and Unchecked Exceptions

In simple terms, exceptions that happen at compile time are **Checked Exceptions**. That is, the compiler checks such exceptions at the time of compilation. The checked exceptions cannot be ignored; the programmer should handle these exceptions.In other words a checked exception must be either caught or declared in the method in which it is thrown.If we do not write the code to handle them, the compiler gives an error. On the other hand, the exceptions that occur at the time of execution, like IOExceptions etc, are ignored at the time of compilation are called **Unchecked Exceptions**. These Unchecked exceptions are also called Runtime exceptions. Examples of unchecked exceptions are: divide by zero, array out of bounds, null pointer exception etc. The unchecked exceptions are built-in exceptions in Java. By good programming practice, many a time, unchecked exceptions can be avoided.

For exception handling in Java a basic format of programming is used. Now let us see it through a program code.

```
class Exception
{
  public static void main(String args[])
  {
    try
    {
      //code that may raise an exception
    }
```

```
        catch(Exception e)
        {

        }
        //Rest of the program
    }
}
```

Understanding the basic structural construct example of exception. Here we have defined the class and inside the main method we have a **try block**, in this try block we are going to write code that will raise exception or it contains the code that may raise exception and then the raised exception will be handled in the **catch block**. Now through a small program, let us see how the exception is handled once it occurs.

First we will create a new package called ExceptionPGD, and then we will create a new class called PGDException. Inside the class the first thing that we are going to do is that to write the main method, so here we'll create string str and make it "NULL". After this we will try to retrieve the length of the string. Notice that we are setting string to "NULL" intentionally, so when we execute this program, it will throw an exception, why, because the string is null and we are trying to retrieve the length of the string. Here you will find that it throws null pointer exception because the string is not there or it does not contain any values. Well, this is what we wanted it to do, throw an exception.

```
package ExceptionPGD;
public class PGDException
{
        public static void main(String args[])
        {
                string str = null;
                system.out.println(str.length( ));
        }
}
```

Output

As we said it throws NullPointerException because the string is null, it does not contain any values. With this understanding of exception, let us begin our journey of understanding how to handle exceptions, with the help of another program.

3.4.2 Using try , catch and finally method to handle exception

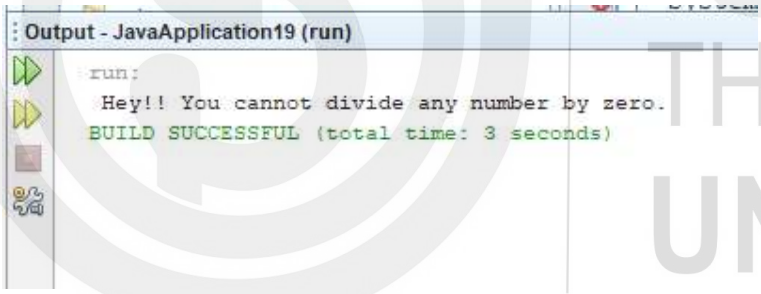
We have already seen the construct of an exception handling program earlier in this chapter. Now we will put it to practice, and to do that here in try block we are going to write the code that will raise exception, this time an ArithmeticException. Just to get the basic rule of maths right. Do you remember if we try to divide a number by zero, assuming both a and c are integer variables and we try, c=a/0 , what happens here ? If

we try to divide a number by zero then it says we cannot divide the number by zero because it raises arithmetic exception a base arithmetic anomaly.

Now as it throws an exception and we saw in our last example that the program exited after throwing an exception. But we don't want to exit if an exception occurs, So what should we do to maintain the normal flow of execution ? Well, this is where we write the catch block to handle this exception. Let's run the program and see the output.

```
package ExceptionPGD;

public class PGDException
{
  public static void main(String args[])
  {
    try
    {
      int a=90, b=0;
      int c=a/b;
      system.out.println("Resultant =" +c)
    }
    catch(ArithmeticException e)
    {
      System.out.println(" Hey!! You cannot divide any number by zero.");
    }
  }
}
```

Output:



As explained above the exception has been 'caught' by catch block so here first when the execution is started an exception is thrown and then the thrown exception is caught by the code of catch block which handles arithmetic exception and prints "Hey !! You cannot divide a number by zero".

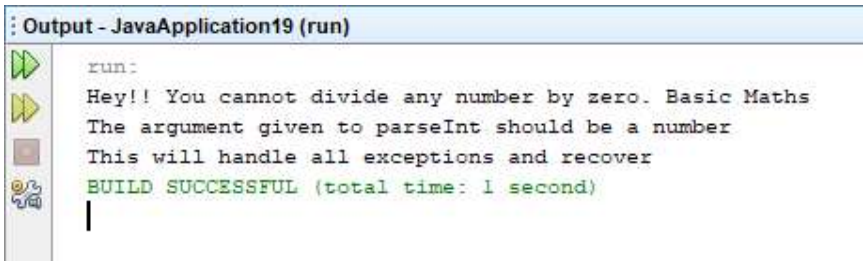
This is how basic exception handling works. Now let us learn various types of exceptions in Java. In Java there are two types of exceptions built-in exceptions and user defined exceptions. First let us see what are built-in exceptions.
Built-in exceptions are the exceptions which are available in Java libraries and they are suitable to explain certain error situations like arithmetic exception, array index out of bound and class not found exception, runtime exception, number format exception etc.

Various methods of exceptions handling are try, catch, finally, throw and throws. Two commonly used exception handling concepts are throw and throws. Only single exception is **thrown** by using **throw method**. Multiple exceptions can be **thrown** by using **throws** method. We have already seen try block which is used to enclose the code that may throw exception and catch block which will handle the thrown

exception. Syntax as you have already seen is very simple. We write a code that throws exception and we can handle using the catch block next. Now let us try nested try block, which is nothing but a try within a try block.

```
package ExceptionPGD;
public class PGDException
{
  public static void main(String args[])
  {
    try
    {
      int a=90, b=0;
      int c=a/b;
      System.out.println("Resultant =" +c)
    }
    catch(ArithmeticException e)
    {
      System.out.println(" Hey!! You cannot divide any number by zero. Basic Maths");
    }
    try
    {
      try
      {
        int num = Integer.parseInt("PGDegree");
        System.out.println(num);
      }
      catch(NumberFormatException e)
      {
        System.out.println(" The argument given to parseInt should be a number");
      }
      try
      {
        int a[] = new int[5];
        a[7]=25;
      }
      catch(ArrayIndexOutofBoundsException e)
      {
        System.out.println("Oops!!! Don't think your array extends to that place,
        array index out of bounds exception");
      }
      System.out.println("Marks culmination of Nested Try block; print some other
      stmt");
    }
    catch(Exception e)
    {
      System.out.println("This will handle all exceptions and recover");
    }
  }
}
```

Output:



Now, let us see example of use of multiple ***catch*** clause in a program.

```
try{                    }
catch (Exception e1)
{// Catch Block}
catch (Exception e2)
{// Catch Block}
catch (Exception e3)
{// Catch Block}
```

To understand the utility of having multiple catch blocks, think of a situation where code may throw multiple exceptions and there is a need to handle each one of those separately and differently. Put simply, if you want to handle each exception differently then use multiple catches.

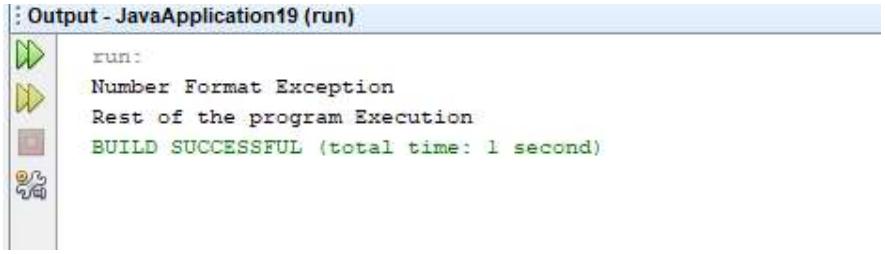
So here we have one try block and have different catch blocks. Now what we should practice while there is a need to handle such different types of exceptions while writing our code is to handle them from sub class exception to super class exception (specialized to generalized). Lets understand this with the help of an example. We have already seen the Exception hierarchy, from this we know that class FileNotFoundException extends IOException i.e. while writing multiple catch blocks we should first handle FileNotFoundException and then IOException. If we do the reverse i.e. handle IOException first, this will result in an unreachable code as FileNotFoundException will never be reached because IOException would have already handled that situation too. So you need to remember to place subclass exceptions higher in the list of catches.

Lets see an example of catching multiple exceptions:

```
Package ExceptionPGD;
public class PGDException
{
  public static void main(String args[])
  {
     try
     {
       int c  =  Integer.parseInt("Checking Virus");
       System.out.println("Resultant:", +c);
     }
     catch (NumberFormatException e)
     {
       System.out.println("Number Format Exception");
     }
     catch (Exception e)
     {
       System.out.println("This will handle all Exceptions");
     }
```

```
    System.out.println("Rest of the program Execution");
  }
}
```

Output:



Java has provided union catch feature since Java 7, where we may handle multiple exceptions using same catch block by clubbing them together as shown in example code below.

```
try {……}
catch (IOException | NumberFormatException e)
{
        System.out.println("This has failed to load", e);
}
```

Use of finally block in Exception Handling

The finally block in java is very important in exceptions handling as it is used to put important codes such as clean up code like closing the file, closing the database connection etc. The finally block is always executed whether exception generated or not and whether generated exceptions are handled or not. Therefore in a finally block all the crucial statements are kept regardless of the exception occurs or not.

```
try{              }
catch (Exception e1)
{// Catch Block}
catch (Exception e2)
{// Catch Block}
catch (Exception e3)
{// Catch Block}
finally { important code}
```

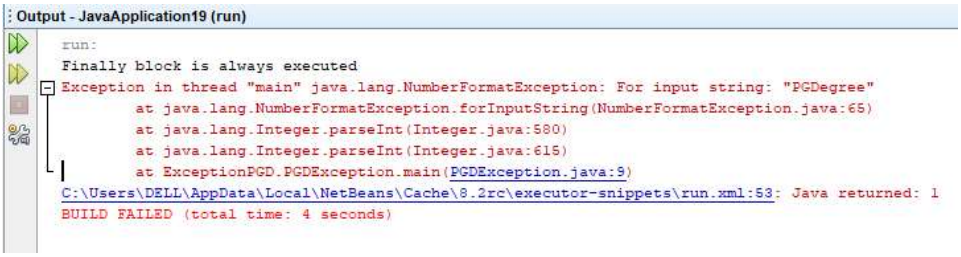
Let us see an example of use of finally block. In this program the code throws an exception however the catch block cannot handle it. Despite this, the finally block is executed after the try block and print the given message then the program terminates abnormally.

```
package ExceptionPGD;
public class PGDException
{
  public static void main(String args[])
  {
    try
    {
      int num = Integer.parseInt("PGDegree");
      system.out.println(num);
    }
    finally
```

```
    {
      System.out.println(" Finally block  is always executed");
  }
}
```

In the above code since there is no catch block the exception thrown will not be caught and will show NumberFormatException, however the flow of program will not be broken and finally block will be executed.

Output:



3.5 THROW AND THROWS

1. throw is explicitly used to throw an exception but throws is used to declare an exception.
2. Checked exceptions cannot be propagated using only throw but it can be propagated using throws.
3. throw is followed by an instance and throws is followed by a class
4. throw is always used within a method and throws is used with the method signature.
5. throw clause can throw only one exception and not multiple exceptions and throws can declare multiple exceptions.

Syntax of throw :

```
void PGD()
{
  throw new ArithmeticException("Not working");
}
```

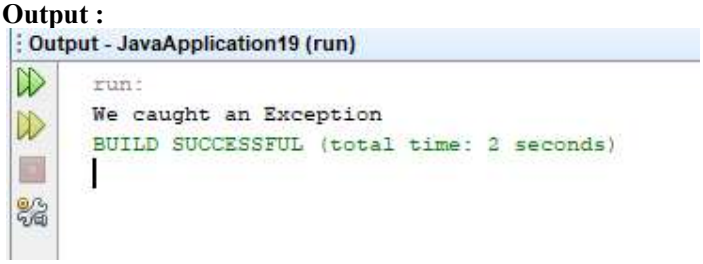
An example which demonstrate use of through clause in programming is given below.

```
package ExceptionPGD;
public class PGDException
{
    static void avrg()
    {
        try
        {
            throw new ArithmeticException("Demonstrating");
        }
        catch(ArithmeticException e)
        {
            System.out.println("We caught an Exception");
        }
    }
    public static void main(String args[])
```

```
    {
        avrg();
    }
}
```

When we execute the program, following output will be received.

Output :



Throws

Throws is also a keyword which is used to declare the exceptions it does not throw any exception but it specifies that there may occur an exception in the method and it is always used with the method signature. Let us see how.

Syntax:

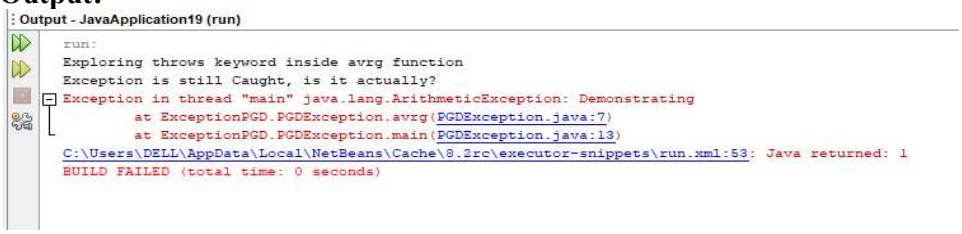
```
void a_method() throws ArithmeticException { }
```

An example to demonstrate use of throws clause.

```
package ExceptionPGD;
public class PGDException
{
        static void avrg() throws ArithmeticException
        {
            System.out.println("Exploring throws keyword inside avrg function");
            throw new ArithmeticException("Demonstrating");
        }
public static void main(String args[])
{
    try
    {
        avrg();
    }
    finally
    {
        System.out.println("Exception is still Caught, is it actually?");
    }
}
}
```

It will notify that there is an exception in the main method (can you guess what type of Exception is it : Hint: Arithmetic) as we are not handling it via catch in our code but still print the following.

Output:



3.6 FINAL VS FINALLY VS FINALIZE

What will happen in the situation when you have to execute some statement irrespective of whether it is caught as an exception or not? To handle such situation of finalize () method is used.
Let us understand the three - final, finally and finalize by looking at the differences between these three constructs; so final is a keyword, finally is a block, and finalize is a method. The final is used to apply restrictions on class methods and variables. The finally block is used to place an important code and finalize method is used to perform cleanup processing just before the object is garbage collected. Remember that a class declared as final cannot be inherited and the final method cannot be overridden and the final variable cannot be changed. As we have already learned it, the finally block will be executed whether the exception is handled or not.

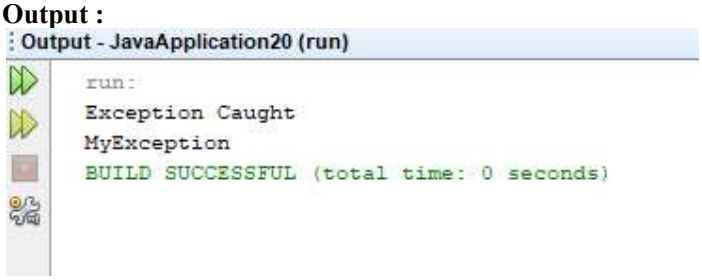
3.7 USER DEFINED EXCEPTION

Sometimes the built-in exceptions in Java are not able to describe a certain situation. In such cases a user can also create exceptions and that are called user defined exceptions . There are two important points which are used while creating/defining user defined exception.

- A user defined exception must extend the exception class
- exception is thrown using a throw keyword

```
package javaapplication20;
class MyException extends Exception
{
   public MyException(String s)
   {
      // Call constructor of parent Exception
      super(s);
   }
}
public class PGDException
 {
     public static void main(String args[])
   {
      try
      {
         // Throw an object of user defined exception
         throw new MyException("MyException");
      }
      catch (MyException e)
      {
         System.out.println("Exception Caught");
          // Print the message from MyException object
         System.out.println(e.getMessage());
      }
   }
}
```

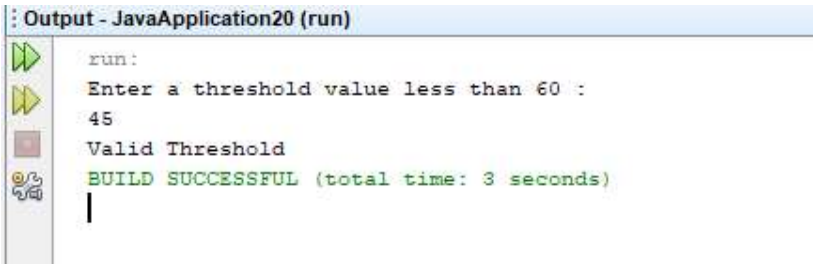
Output :



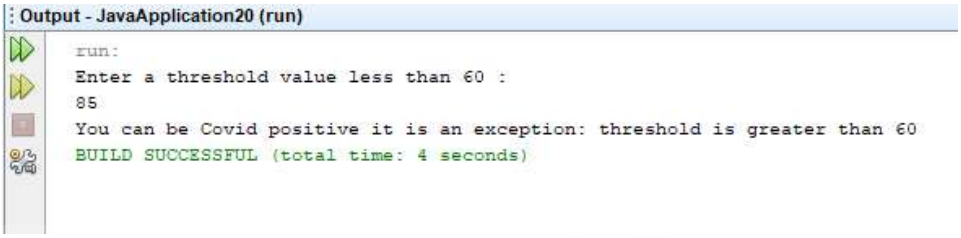
Another Example Program for Custom Exception

```
package javaapplication20;
import java.util.*;

class PGDException
{
static void ValidateThreshold(int threshold) throws InvalidInputException
{
if(threshold > 60)
{
throw new InvalidInputException("Invalid Threshold");
}
System.out.println("Valid Threshold");
}

public static void main( String args[])
{
Scanner scanner = new Scanner(System.in);
System.out.println("Enter a threshold value less than 60 :");
int threshold = scanner.nextInt();
try
{
ValidateThreshold(threshold);
}
catch(InvalidInputException e)
{
System.out.println("You can be Covid positive it is an exception: threshold is greater
than 60");
}
}
}
class InvalidInputException extends Exception
{
InvalidInputException(String exceptionText)
{
super(exceptionText);
}
}
```

Output-1 (for input value 45)



Output-2 (for input value 85)



Custom and User defined Exceptions come to our rescue in the rare situations where java already dose not has an exception class defined to handle an exception. These are handy tools which maintain the flow and readability of our code.

☞ Check Your Progress-3

1. Can there be a try block without a catch block ?

.....

.....

.....

.....

2. Which is the highest level of Exception Handling classes ?

.....

.....

.....

.....

3. Show the pre-defined exception hierarchy in Java

.....

.....

.....

.....

4. How to create a Custom Exception Classes?

.....

.....

5. What is the difference between java- ClassNotFoundException and NoClassDefFoundError.

6. What will happen when you compile and run the following code?

```
package javaapplication20;

public class PGDException extends Exception
{
String className;
public static void main(String[] args)
{
try
{
PGDException v = new PGDException();
if(v.className.equals("My Exception"
System.out.print("My Exception");
else
System.out.print("Other Exception");
}
catch(Exception e)
{
System.out.print("Exception Caught");
}
catch(NullPointerException ne)
{
System.out.print("Null");
}
}
}
```

7. Can an Exception be rethrown ? Is it required for the caller method to catch or re-throw the checked exception.

3.8 SUMMARY

In this unit we began with learning a unique feature of java language which is an statement *assertion* which is primarily used in non-production environment of code to prevent programmers making costly mistakes repeatedly and saw its advantages. Then we learnt about a relatively recent but important feature *annotations* which helps us document a program without writing lengthy comments. Finally we culminated the unit by learning how *Exception Handling* is done in java and also learnt how we create our own Exceptions.

3.9 SOLUTIONS/ ANSWER TO CHECK YOUR PROGRESS

☞ Check your progress-1

1. Assertions are **disabled** by default in java. We can use *java –ea or java -da* to enable or disable assertions respectively. If conditional statements used to check bug conditions reach production even after the bug has been removed they consume CPU cycles and may be cause of bottleneck or resource crunch. This problem does not exist with assertions as they are disabled by default.

2. Assert usually throws "AssertionError" when the assumption made is wrong. AssertionError extends from Error class (that ultimately extends from Throwable).

3. If assertions are enabled for the program in which the assertion fails, then it will throw AssertionError.

4. An assert statement declares a Boolean condition that is expected to occur in a program. If this boolean condition evaluates to false, then an AssertionError is given at runtime provided the assertion is enabled. If the assumption is correct, then the boolean condition will return true.

5. The AssertionError thrown by the assert statement is an unchecked exception that extends the Error class. The assertions are not required to declare them explicitly and also there is no need to try or catch them.

6. To assert an exception we declare an object of ExpectedException as follows:
   public ExpectedException exception = ExpectedException. none ();
   Then we use it's expected () and expectMessage() methods in the test method, to assert the exception, and give the exception message.

☞ Check your Progress-2

1. These are :
   - *@Override* – marks that a method is meant to override an element declared in a superclass.
   - *@Deprecated* – indicates that element is deprecated and should not be used.
   - *@SuppressWarnings* – tells the compiler to suppress specific warnings.
   - *@FunctionalInterface* – introduced in Java 8, indicates that the type declaration is a functional interface and whose implementation can be provided using a Lambda Expression

2. Annotations are there in the *java.lang* and *java.lang.annotation* packages.

3. Annotations are a form of an interface where the keyword *interface* is preceded by *@,* and the body contains *annotation type element* declarations that look just like methods:

```
public @interface SelfDefinedAnnotation
{
   String value();

   int[] types();
}
```

To start using it using your code :

```
@SelfDefinedAnnotation(value = "A String type element", types = 1)
public class MyAnnotationUser
{
   @SelfDefinedAnnotation(value = "any new relevant value", types = { 31, 48 })
   public Element nextElement;
```

4. On behalf of the annotation, java compiler or JVM performs some additional operations. By default, annotations are not inherited to subclasses. The @Inherited annotation marks the annotation to be inherited to subclasses.

5. Object

6. java.text

7.
@Retention annotation specifies how the marked annotation is stored.
@Documented annotation indicates that whenever the specified annotation is used those elements should be documented using the Javadoc tool. (By default, annotations are not included in Javadoc.)
@Target annotation marks another annotation to restrict what kind of Java elements the annotation can be applied to.
@Inherited annotation indicates that the annotation type can be inherited from the super class. (This is not true by default.) When the user queries the annotation type and the class has no annotation for this type, the class superclass is queried for the annotation type. This annotation applies only to class declarations.
@Repeatable annotation, introduced in Java SE 8, indicates that the marked annotation can be applied more than once to the same declaration or type use.

☞ Check Your Progress-3:

1. Yes. (java 7 onwards)

2. Throwable is the highest level of Error Handling classes.

3. //Pre-defined Java Classes

```
class Error extends Throwable{}
class Exception extends Throwable{}
class RuntimeException extends Exception{}
```

4. class MyOwnRTPCRTestException extends Exception{}

5. Both *ClassNotFoundException* and *NoClassDefFoundError* occur when the JVM can not find a requested class on the classpath. *ClassNotFoundException* is a checked exception which occurs when an application tries to load a class through its fully-qualified name and can not find its definition on the classpath *NoClassDefFoundError* is a fatal error. The error occurs when a compiler could successfully compile the class, but Java runtime could not locate the class file.

6. The catch block catching child class must come first in order before the catch block catching the parent class.
NullPointerException is child class of the Exception class (Exception > RuntimeException > NullPointerException), so the catch block with NullPointerException must come before the catch block with Exception class. The code will give compilation error "Unreachable catch block for NullPointerException. It is already handled by the catch block for Exception".

7. Yes , Exception can be rethrown. Yes, if the method is throwing any of the checked exceptions, the caller must either catch it using try catch block or re-throw it using throws clause. Otherwise, compilation fails.

3.10 REFFERENCES/FURTHER READING

- Herbert Schildt "Java The Complete Reference", McGraw-Hill, 2017.
- Savitch, Walter, " Java: An introduction to problem solving & programming", Pearson Education Limited, 2019.
- Muneer Ahmad Dar, "Java Programming Simplified: From Novice to Professional", BPB, 2020.
- Lulliana Cosmina, " Java for Absolute Beginners", Apress,2018.
- Yashvant Kanetkar, " Let Us Java", BPB,2019.
- https://www.buggybread.com
- https://www.baeldung.com
- https://www.javacodeexamples.com
- https://docs.oracle.com/en/java/