Unit 5

Q) Exception handling fundamentals, Exception types, Exception as objects, Exception hierarchy, Exception Keywords - Try, catch, finally, throw, throws

The Exception handling in Java is one of the powerful mechanism to handle the runtime errors so that normal flow of the application can be maintained.

In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLExcep-

tion, RemoteException etc.

Advantage of Exception Handling

The core advantage of exception handling is to maintain the normal flow of the application. An exception normally disrupts the normal flow of the application that is why we use exception handling.

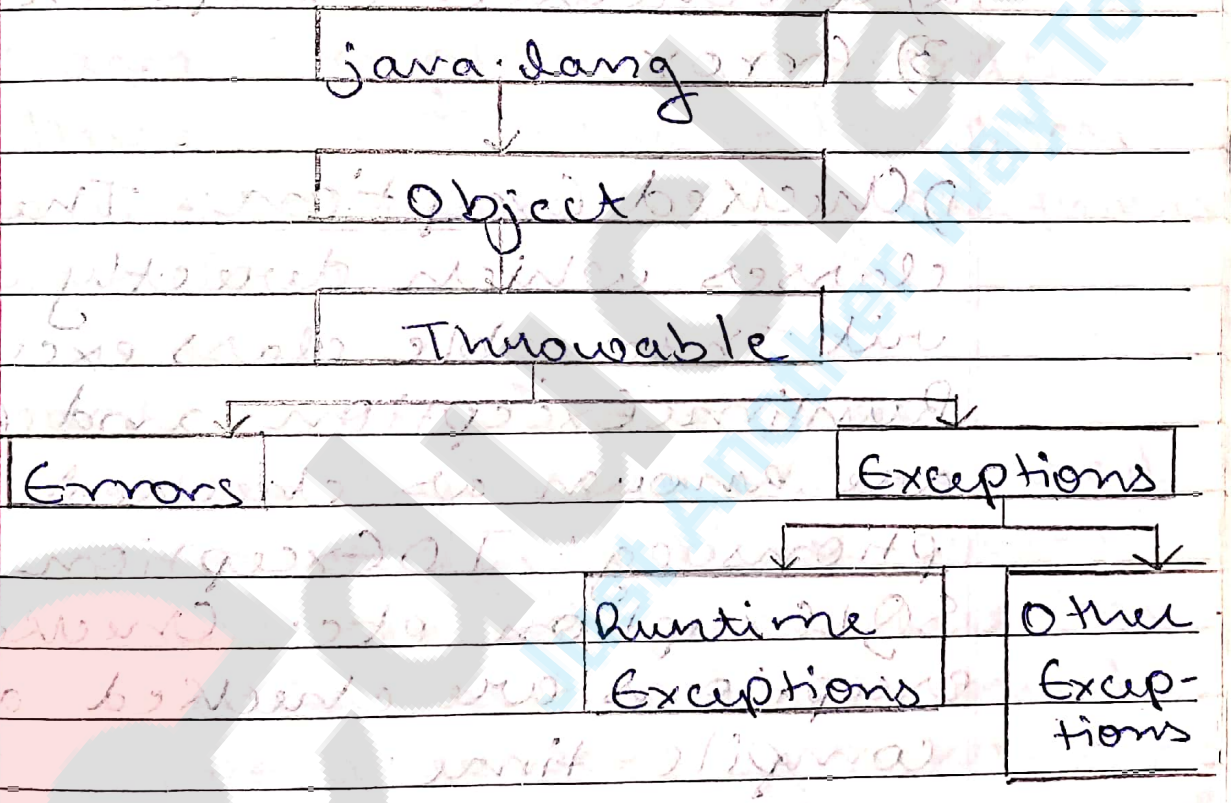
Let's take a scenario:

```
Statement 1;
Statement 2;
Statement 3;
Statement 4;
Statement 5; // exception occurs
Statement 6;
Statement 7;
Statement 8;
Statement 9;
Statement 10;
```

Suppose there are 10 statements in your program and there occurs an exception at statement 5, the rest of the code will not be exe executed i.e statement 6 to

10 will not be executed. If we perform exception handling, the rest of the statement will be executed. That is why we use exception handling in Java.

Hierarchy of Java Exception classes



The java.lang.Throwable class is the root class of Java Exception hierarchy which is inherited by two subclasses: Exceptions and Error.

Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception. According to Oracle, there are three types of exceptions:

1) Checked Exception
2) Unchecked Exception
3) Error

1) Checked Exception:- The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions eg: IOException, SQLException etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception:- The classes which inherit RuntimeException are known as unchecked exceptions eg ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.

Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

Error is irrecoverable eg:- OutOfMemoryError, VirtualMachineError, AssertionError etc.

Java Exception Keywords

There are 5 keywords which are used in handling exceptions in Java.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we

	can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.
& throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

Java Exception Handling Example :-

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

```
int data = 100/0;
} catch (ArithmeticException e)
{
System.out.println(e);
}
System.out.println("rest of the code...");
}
}
```

Output

Exception in thread main java.lang.ArithmeticException: / by zero

Common Scenarios of Java Exceptions

1) A scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException

```
int a=50/0;
```

2) A scenario where NullPointerException occurs

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException

```
String s = null;
System.out.println(s.length());
```

3) A scenario where NumberFormatException occurs

The wrong formatting of any value may occur NumberFormatException. Suppose I have a string variable that has characters, converting this variable into digit will occur NumberFormatException.

```
String s = "abc";
int i = Integer.parseInt(s);
//NumberFormatException
```

4) A scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would result in ArrayIndexOutOfBoundsException

as shown below:

```
int a[] = new int[5];
a[10] = 50;
```

Exception Objects.

```
try
{
// Statements, some of which might
throw an exception
}
catch (SomeExceptionType exp) //
may be omitted if there is a
//finally block
{
// Statements to handle
// SomeExceptionType exceptions
}
// additional catch blocks (optional)
// finally block (optional, unless
   there are no catch blocks)
```

When a catch{} block receives control it has a reference to an object of class Exception (or a subclass of Exception). The class of the object depen-

ds on what exception was thrown.
When an exception event occurs while a program is running, the Java run time system takes over and creates an Exception object to represent the event. Information about the event is put in the object. If the exception arose inside a try { } block, the Java run time system sends the Exception object to the appropriate catch { } block (if there is one).
The parameter of the catch { } block (exp in the above) refers to the Exception object.

Q) Creating User defined Exceptions
Java provides us facility to create our own exceptions which are basically derived classes of Exceptions for

PAGE NO.....
DATE ...../...../.....

example MyException in below code extends the Exception class.
We pass the string to the constructor of the super class - Exception which is obtained using "getMessage()" function on the object created.

```
class MyException extends Exception
{
public MyException(String s)
{
super(s);
}
}

public class Main
{
public static void main(String args[])
{
try
{
throw new MyException("
GeekGeeks");
}
catch (MyException ex)
{
```

```
        System.out.println("Caught");
        System.out.println(ex.getMess
        age());
    }
  }
}
```

Output:

```
Caught
GeeksGeeks
```

In the above code, constructor of MyException requires a string as its argument. The string is passed to parent class Exception's constructor using super(). The constructor of Exception class can also be called without a parameter and call to super is not mandatory.

```
class MyException extends Exception
{
}
public class setText
```

```
{
public static void main(String
args[])
{
try
{
throw new MyException();
}
catch(MyException ex)
{
System.out.println("Caught");
System.out.println(ex.getMess-
age());
}
}
}
```

Output:
Caught
null

Q] Assertions in Java
An assertion allows testing the correctness of any assumptions that have been made in the program.
Assertion is achieved using

The assert statement in Java. While executing assertion, it is believed to be true. If it fails, JVM throws an error named AssertionError. It is mainly used for testing purposes during development.

The assert statement is used with a Boolean expression and can be written in two different ways.

1) assert expression;
2) assert expression1 : expression2;

Example

```
import java.util.Scanner;
class Test
{
public static void main (String
  args [])
  {
  int value = 15;
  assert value >= 20 : "Under-
  weight";
  System.out.println("value
is" + value);
```

PAGE NO.....
DATE...../...../.....

}
}

Output
value is 15

After enabling assertions
Output
Exception in thread "main"
java.lang.AssertionError:
Underweight

Enabling Assertions
By default, assertions are disabled. We need to run the code as given. The syntax for enabling assertion statement in Java source code is:

java -ea Test
Or
java -enableassertions Test

Disabling Assertions
The syntax for disabling assertions in java are:

java -da Test or

java -disableassertions Test

Why use Assertions?
Wherever a programmer wants to see if his/her assumptions are wrong or not..
To make sure that an unreachable looking code is actually unreachable.
To make sure that assumptions written in comments are right.

```
if ((x & 1) == 1)
  { }
else // x must be even
  { assert (x % 2 == 0); }
```

To make sure default switch case is not reached.
To check object's state.
On the beginning of the method.
After method invocation.

Assertion vs normal Exception Handling

Assertions are mainly used to check logically impossible situations. For example, they can be used to check the state a code expects before it starts running or state after it finishes running. Unlike normal exception / error handling, assertions are generally disabled at run-time.

Where to use Assertions

- Arguments to private methods: Private arguments are provided by developers code only and developer may want to check his/her assumptions about arguments.
- Conditional cases.
- Conditions at the beginning of any method.

Where not to use Assertions

- Assertions should not be used to replace error messages.
- Assertions should not be used to check arguments

ins the public methods as they may be provided by user. Error handling should be used to handle errors provided by user.
- Assertions should not be used on command line arguments.

Java Annotations

Java Annotation is a tag that represents the metadata i.e attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.

Annotations in Java are used to provide additional information, so it is an alternative option for XML and Java marker interfaces.

Built-in Java Annotations

There are several built-in annotations in Java. Some annotations are applied to

Java code and some to other annotations.

Built-in Java Annotations used in Java code

1) @Override
2) @SuppressWarnings
3) @Deprecated

Built-in Java Annotations used in other annotations

1) @Target
2) @Retention
3) @Inherited
4) @Documented

Understanding Built-in annotations

@Override

@Override annotation assures that the subclass method is overriding the parent class method. If it is not so, compile time error occurs. Sometimes, we do silly mistake such as spelling mistakes etc. So, it is better to mark @Override annotation that provides assurity that method is overridden.

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

Output: Compile Time Error

@SuppressWarnings

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

```
import java.util.*;
class TestAnnotation2{
```

PAGE NO.....
DATE. ...../...../.....

```
@SuppressWarnings ("unchecked")
public static void main (String
 args[])
ArrayList = new ArrayList();
list.add ("Sonoo");
list.add ("vimal");
list.add ("ratan");
for (Object obj: list)
System.out.println(obj);
  }
 }
```

Now no warning at compile time.

If you remove the @SuppressWarnings ("unchecked") annotations it will show warning at compile time because we are using non-generic collection

@Deprecated

@Deprecated annotation marks that this method is deprecated so compiler prints warning. It informs user that it may be removed in the future versions. So, it is better not to use such meth-

ods:"

```
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();
}
}
```

At Compile Time:

Note: Test.java uses or overrides a deprecated API.

Note: Recompile with -Xlint:deprecation for details.

At Runtime:

hello n

PAGE NO.....
DATE...../...../.....

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:

```
@interface MyAnnotation{}
```

Points to remember for java custom annotation signature

There are few points that should be remembered by the programmer.

1) Method should not have any throws clauses
2) Method should return one of the following: primitive data types, String, Class, enum or array of these data types.
3) Method should not have any parameter.
4) We should attatch @just before interface keyword to define annotation
5) It may assign a dyault value to the method.

Types of Annotation

1) Marker Annotation
2) Single-Value Annotation
3) Multi-Value Annotation

1) Marker Annotation

An annotation that has no method, is called marker annotation. For example:

```
@interface MyAnnotation { }
```

The @Override and @Deprecated are marker annotations.

2) Single-Value Annotation

An annotation that has one method, is called single-value annotation. For example:

```
@interface MyAnnotation {
int value();
}
```

We can provide the default value also. For example:

```
@interface MyAnnotation {
int value() default 0;
}
```

The code to apply the single value annotation.

@MyAnnotation(value=10)
The value can be anything.

3) Multi-Value Annotation
An annotation that has more than one method, is called Multi-value annotation. For example:

```
@interface MyAnnotation{
int value1();
String value2();
String value3();
}
}
```

We can provide the default value also. For example

```
@interface MyAnnotation{
int value1() default 1;
String value2() default "";
String value3() default "xyz";
}
```

The code to apply the multi-value annotation

```
@MyAnnotation(value1=10, value2="Arun Kumar", value3="Ghaziabad")
```

Built-in Annotations used in custom annotations in java

1) @Target
2) @Retention
3) @Inherited
4) @Documented

@Target

@Target tag is used to specify at which type, the annotation is used.

The java.lang.annotation.ElementType enum declares many constants to specify the type of element where annotation is to be applied such as TYPE, METHOD, FIELD etc.

Element Types	Where the annotation can be applied

TYPE	class, interface or enumeration
FIELD	fields
METHOD	methods
CONSTRUCTOR	constructors
LOCAL_VARIABLE	local variables
ANNOTATION_TYPE	annotation type
PARAMETER	parameter

Example to specify annotation for a class

```
@Target(ElementType.TYPE)
@interface MyAnnotation{
int value1();
String value2();
}
```

Example to specify annotation for a class, methods or fields

```
@Target{ElementType.TYPE, ElementType.FIELD, ElementType.ME
@interface MyAnnotation{
int value1();
String value2();
}
```

@Retention

@Retention annotation is used to specify to what level annotation will be available

RetentionPolicy	Availability
RetentionPolicy.SOURCE	refers to the source code, discarded during compilation. It will not be available in the compiled class.
RetentionPolicy.CLASS	refers to the class file, available to java compiler but not to JVM. It is included in the class file.
RetentionPolicy.RUNTIME	refers to the runtime, available to java compiler and JVM.

Example to specify the RetentionPolicy

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@interface MyAnnotation{
```

```
int value1();
String value2();
}
```

Example of custom annotation: creating, applying and accessing annotation

File: Test.java

```
import java.lang.annotation.*;
import java.lang.reflect.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface MyAnnotation{
int value();
}
class Hello{
@MyAnnotation(value=10)
public void sayHello(){
System.out.println("hello annotation");
}
}
class TestCustomAnnotation1{
public static void main(String args
[])throws Exception{
Hello h=new Hello();
Method m=h.getClass().getMethod
("sayHello");
```

MyAnnotation manno=m.getAnnotation(MyAnnotation.class);
System.out.println("value is:"+manno.value());
}}

Output: value is: 10

How built-in annotations are used in real scenario?

In real scenario, java programmer only need to apply annotation. He/She doesn't need to create and access annotation. Creating and accessing annotation is performed by the implementation provider. On behalf of the annotation, java compiler or JVM performs some additional operations.

@Inherited

By default, annotations are not inherited to subclasses. The @Inherited annotation marks the annotation to be inherited to subclasses.

PAGE NO.....
DATE ...../...../.....

@Inherited
@interface ForEveryone{}
@interface ForEveryone{}
class Superclass{}
class ~~Does~~ Subclass extends Superclass{}

@Documented

The @Documented Marks the annotation for inclusion in the documentation.