**Exception Handling**

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.

An exception is an unwanted or unexpected event, which occurs during the execution of a program i.e at run time, that disrupts the normal flow of the program’s instructions.

Exception is an abnormal condition.

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

If an exception occurs, which has not been handled by programmer then program execution gets terminated and a system generated error message is shown to the user.

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

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.

Suppose there are 10 statements in your program and there occurs an exception at statement 5, the rest of the code will not be 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](https://www.javatpoint.com/java-tutorial).

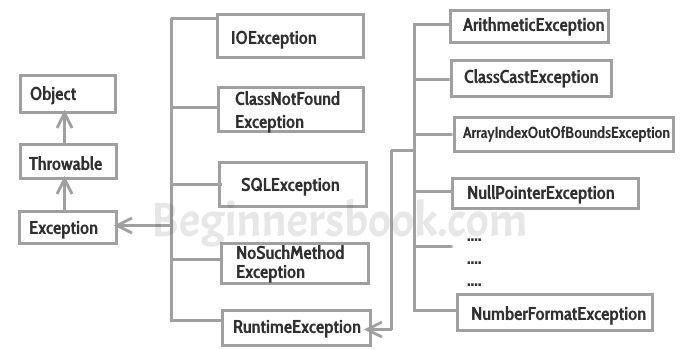
There can be several reasons that can cause a program to throw exception. For example: Opening a non-existing file in your program, Network connection problem, bad input data provided by user etc.

## **Difference between error and exception**

****Errors**** indicate that something severe enough has gone wrong, the application should crash rather than try to handle the error.An Error indicates serious problem that a reasonable application should not try to catch.

****Exceptions**** are events that occurs in the code. A programmer can handle such conditions and take necessary corrective actions.Exception indicates conditions that a reasonable application might try to catch. Few examples:  
**NullPointerException** – When you try to use a reference that points to null.  
**ArithmeticException** – When bad data is provided by user, for example, when you try to divide a number by zero this exception occurs because dividing a number by zero is undefined.  
**ArrayIndexOutOfBoundsException** – When you try to access the elements of an array out of its bounds, for example array size is 5 (which means it has five elements) and you are trying to access the 10th element.

## **Hierarchy of Java Exception classes**

The java.lang.Throwable class is the root class of Java Exception hierarchy which is inherited by two subclasses: Exception and Error. A hierarchy of Java Exception classes are given below:



All exception and errors types are sub classes of class Throwable, which is base class of hierarchy.One branch is headed by Exception. This class is used for exceptional conditions that user programs should catch. NullPointerException is an example of such an exception.Another branch,Error are used by the Java run-time system([JVM](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/)) to indicate errors having to do with the run-time environment itself(JRE). StackOverflowError is an example of such an error.



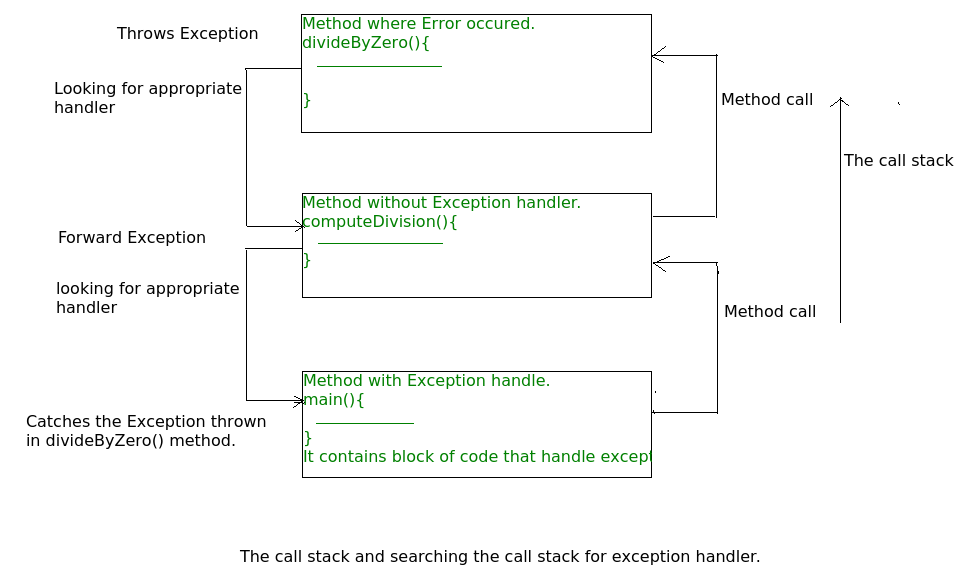
**How JVM handle an Exception?**

**Default Exception Handling :** Whenever inside a method, if an exception has occurred, the method creates an Object known as Exception Object and hands it off to the run-time system(JVM). The exception object contains name and description of the exception, and current state of the program where exception has occurred. Creating the Exception Object and handling it to the run-time system is called throwing an Exception.There might be the list of the methods that had been called to get to the method where exception was occurred. This ordered list of the methods is called Call Stack.Now the following procedure will happen.

* The run-time system searches the call stack to find the method that contains block of code that can handle the occurred exception. The block of the code is called Exception handler.
* The run-time system starts searching from the method in which exception occurred, proceeds through call stack in the reverse order in which methods were called.
* If it finds  appropriate handler then it passes the occurred exception to it. Appropriate handler means the type of the exception object thrown matches the type of the exception object it can handle.
* If run-time system searches all the methods on call stack and couldn’t have found the appropriate handler then run-time system handover the Exception Object to default exception handler , which is part of run-time system. This handler prints the exception information in the following format and terminates program abnormally.

Exception in thread "xxx" Name of Exception : Description

... ...... .. // Call Stack



****How Programmer handles an exception?****

****Customized Exception Handling :**Java exception handling is managed via five keywords: try, catch, [throw](https://www.geeksforgeeks.org/throw-throws-java/), [throws](https://www.geeksforgeeks.org/throw-throws-java/), and finally.**

**Program statements that you think can raise exceptions are contained within a try block. If an exception occurs within the try block, it is thrown. Your code can catch this exception (using catch block) and handle it in some rational manner.**

**System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword [throw](https://www.geeksforgeeks.org/throw-throws-java/).**

**Any exception that is thrown out of a method must be specified as such by a [throws](https://www.geeksforgeeks.org/throw-throws-java/) clause. Any code that absolutely must be executed after a try block completes is put in a finally block.**

****Explanation** : In the example an array is defined with size i.e. you can access elements only from index 0 to 3. But you trying to access the elements at index 4(by mistake) that’s why it is throwing an exception.In this case, JVM terminates the program abnormally. The statement System.out.println(“Hi, I want to execute”); will never execute. To execute it, we must handled the exception using try-catch. Hence to continue normal flow of the program, we need **try-catch** clause.**

****How to use try-catch clause****

**try** {

// block of code to monitor for errors

// the code you think can raise an exception

}

**catch** (ExceptionType1 exOb) {

// exception handler for ExceptionType1

}

**catch** (ExceptionType2 exOb) {

// exception handler for ExceptionType2

}

// optional

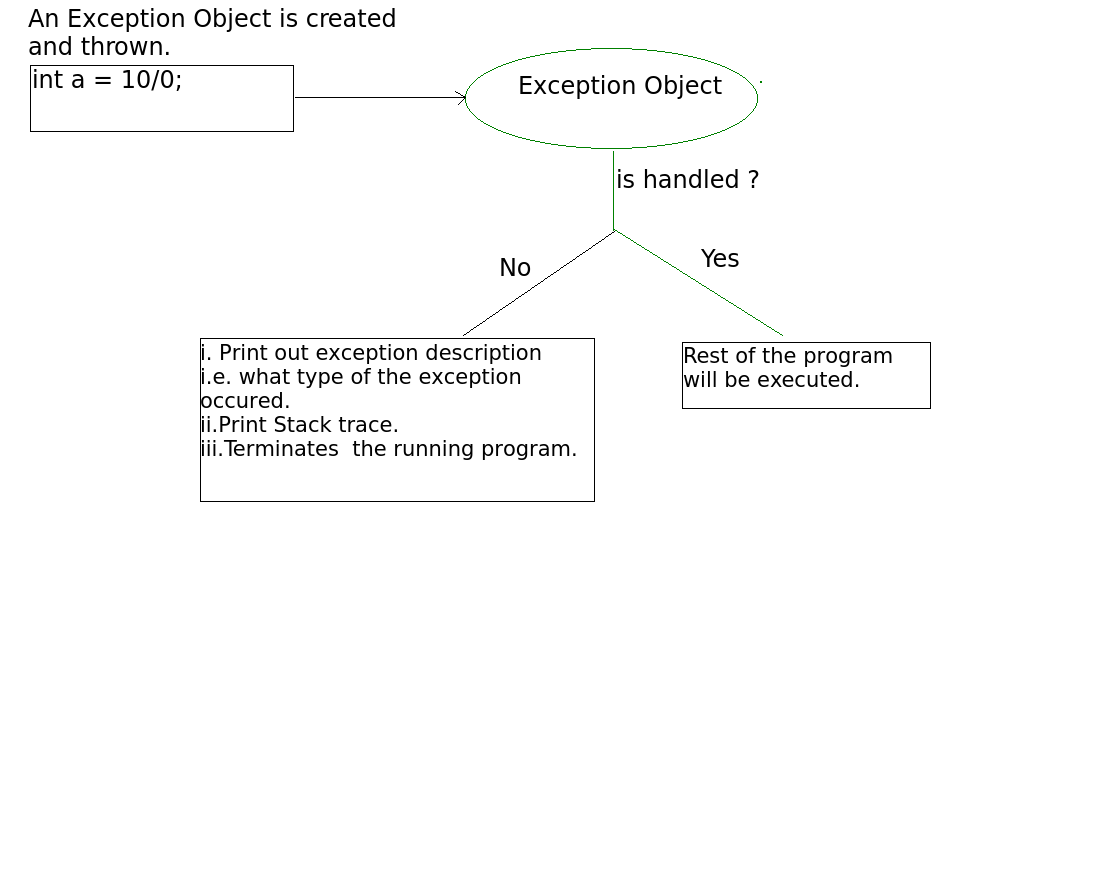
**finally** {

// block of code to be executed after try block ends

}

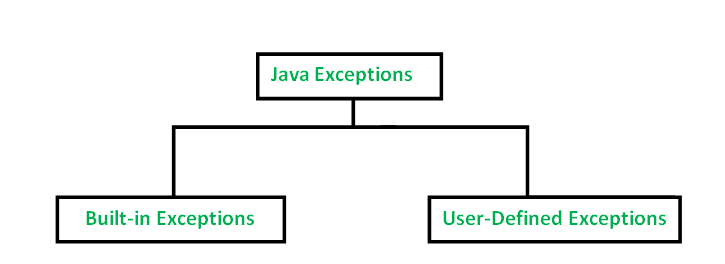
****Points to remember :****

* **In a method, there can be more than one statements that might throw exception, So put all these statements within its own try block and provide separate exception handler within own catch block for each of them.**
* **If an exception occurs within the try block, that exception is handled by the exception handler associated with it. To associate exception handler, we must put catch block after it. There can be more than one exception handlers. Each catch block is a exception handler that handles the exception of the type indicated by its argument. The argument, ExceptionType declares the type of the exception that it can handle and must be the name of the class that inherits from Throwable class.**
* **For each try block there can be zero or more catch blocks, but only one finally block.**
* **The finally block is optional.It always gets executed whether an exception occurred in try block or not . If exception occurs, then it will be executed after try and catch blocks. And if exception does not occur then it will be executed after the try block. The finally block in java is used to put important codes such as clean up code e.g. closing the file or closing the connection.**



****Types of Exception in Java****

**Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.**



**Built-in exceptions are the exceptions which are available in Java libraries. These exceptions are suitable to explain certain error situations. Below is the list of important built-in exceptions in Java.**

1. ****ArithmeticException**  
   It is thrown when an exceptional condition has occurred in an arithmetic operation.**
2. ****ArrayIndexOutOfBoundsException**  
   It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.**
3. ****ClassNotFoundException**  
   This Exception is raised when we try to access a class whose definition is not found**
4. ****FileNotFoundException**  
   This Exception is raised when a file is not accessible or does not open.**
5. ****IOException**  
   It is thrown when an input-output operation failed or interrupted**
6. ****InterruptedException**  
   It is thrown when a thread is waiting , sleeping , or doing some processing , and it is interrupted.**
7. ****NoSuchFieldException**  
   It is thrown when a class does not contain the field (or variable) specified**
8. ****NoSuchMethodException**  
   It is thrown when accessing a method which is not found.**
9. ****NullPointerException**  
   This exception is raised when referring to the members of a null object. Null represents nothing**
10. ****NumberFormatException**  
    This exception is raised when a method could not convert a string into a numeric format.**
11. ****RuntimeException**  
    This represents any exception which occurs during runtime.**
12. ****StringIndexOutOfBoundsException**  
    It is thrown by String class methods to indicate that an index is either negative than the size of the string**

****User-Defined Exceptions****

**Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, user can also create exceptions which are called ‘user-defined Exceptions’.  
Following steps are followed for the creation of user-defined Exception.**

* **The user should create an exception class as a subclass of Exception class. Since all the exceptions are subclasses of Exception class, the user should also make his class a subclass of it. This is done as:**

****class MyException extends Exception****

* **We can write a default constructor in his own exception class.**

****MyException(){}****

* **We can also create a parameterized constructor with a string as a parameter.  
  We can use this to store exception details. We can call super class(Exception) constructor from this and send the string there.**

****MyException(String str)****

****{****

****super(str);****

****}****

* **To raise exception of user-defined type, we need to create an object to his exception class and throw it using throw clause, as:**

****MyException me = new MyException(“Exception details”);****

**throw me;**

* **The following program illustrates how to create own exception class MyException.**
* **Details of account numbers, customer names, and balance amounts are taken in the form of three arrays.**
* **In main() method, the details are displayed using a for-loop. At this time, check is done if in any account the balance amount is less than the minimum balance amount to be ept in the account.**
* **If it is so, then MyException is raised and a message is displayed “Balance amount is less”.**

// Java program to demonstrate user defined exception

// This program throws an exception whenever balance

// amount is below Rs 1000

**class** MyException **extends** Exception

{

    //store account information

**private** **static** **int** accno[] = {1001, 1002, 1003, 1004};

**private** **static** String name[] =

                 {"Nish", "Shubh", "Sush", "Abhi", "Akash"};

**private** **static** **double** bal[] =

         {10000.00, 12000.00, 5600.0, 999.00, 1100.55};

    // default constructor

    MyException() {    }

    // parametrized constructor

    MyException(String str) { **super**(str); }

    // write main()

**public** **static** **void** main(String[] args)

    {

**try**  {

            // display the heading for the table

            System.out.println("ACCNO" + "\t" + "CUSTOMER" +

                                           "\t" + "BALANCE");

            // display the actual account information

**for** (**int** i = 0; i < 5 ; i++)

            {

                System.out.println(accno[i] + "\t" + name[i] +

                                               "\t" + bal[i]);

                // display own exception if balance < 1000

**if** (bal[i] < 1000)

                {

                    MyException me =

**new** MyException("Balance is less than 1000");

**throw** me;

                }

            }

        } //end of try

**catch** (MyException e) {

            e.printStackTrace();

        }

    }

}

****Checked vs Unchecked Exceptions :****

**In Java, there are two types of exceptions:**

**1) **Checked**: are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using throws keyword.**

**For example, consider the following Java program that opens file at location “C:\test\a.txt” and prints the first three lines of it. The program doesn’t compile, because the function main() uses FileReader() and FileReader() throws a checked exception FileNotFoundException. It also uses readLine() and close() methods, and these methods also throw checked exception IOException**

**To fix the above program, we either need to specify list of exceptions using throws, or we need to use try-catch block.**

**Since FileNotFoundException is a subclass of IOException, we can just specify IOException in the throws list and make the above program compiler-error-free.**

1. ****Unchecked**are the exceptions that are not checked at compiled time. In C++, all exceptions are unchecked, so it is not forced by the compiler to either handle or specify the exception. It is up to the programmers to be civilized, and specify or catch the exceptions.  
   In Java exceptions under Error and RuntimeException classes are unchecked exceptions, everything else under throwable is checked.**

**Consider the following Java program. It compiles fine, but it throws ArithmeticException when run. The compiler allows it to compile, because ArithmeticException is an unchecked exception.**

### ****1) Checked Exception****

**The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.**

### ****2) Unchecked Exception****

**The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.**

### ****3) Error****

**Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.**

****Should we make our exceptions checked or unchecked?****

**If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception.**

## ****Java Exception Keywords****

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

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 try block**

Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.

If an exception occurs at the particular statement of try block, the rest of the block code will not execute. So, it is recommended not to keeping the code in try block that will not throw an exception.

Java try block must be followed by either catch or finally block.

1. **try**{
2. //code that may throw an exception
3. }**catch**(Exception\_class\_Name ref){}

## **Java catch block**

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception ( i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

The catch block must be used after the try block only. You can use multiple catch block with a single try block.

## **Internal working of java try-catch block**

The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

* Prints out exception description.
* Prints the stack trace (Hierarchy of methods where the exception occurred).
* Causes the program to terminate.

But if exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed.

## **Java Multi-catch block**

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

## **Points to remember**

* At a time only one exception occurs and at a time only one catch block is executed.
* All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.

try{

int a[]=new int[5];

a[5]=30/0;

}

catch(ArithmeticException e)

{

System.out.println("Arithmetic Exception occurs");

}

catch(ArrayIndexOutOfBoundsException e)

{

System.out.println("ArrayIndexOutOfBounds Exception occurs");

}

catch(Exception e)

{

System.out.println("Parent Exception occurs");

}

# **Java Nested try block**

The try block within a try block is known as nested try block in java.

# **Why use nested try block**

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

# **Java finally block**

**Java finally block** is a block that is used to execute important code such as closing connection, stream etc.

Java finally block is always executed whether exception is handled or not.

Java finally block follows try or catch block.



If you don't handle exception, before terminating the program, JVM executes finally block(if any).

# **Why use java finally**

* Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

For each try block there can be zero or more catch blocks, but only one finally block.

Note: The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).

**Java throw keyword**

The Java throw keyword is used to explicitly throw an exception.

We can throw either checked or uncheked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception. We will see custom exceptions later.

The syntax of java throw keyword is given below.

**throw exception;**

**throw new IOException("sorry device error);**

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

1. **public** **class** TestThrow1{
2. **static** **void** validate(**int** age){
3. **if**(age<18)
4. **throw** **new** ArithmeticException("not valid");
5. **else**
6. System.out.println("welcome to vote");
7. }
8. **public** **static** **void** main(String args[]){
9. validate(13);
10. System.out.println("rest of the code...");
11. }
12. }

**Java Exception propagation**

An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method,If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack.This is called exception propagation.

**Rule: By default Unchecked Exceptions are forwarded in calling chain (propagated).**

lass TestExceptionPropagation1{

void m(){

int data=50/0;

}

void n(){

m();

}

void p(){

try{

n();

}catch(Exception e){System.out.println("exception handled");}

}

public static void main(String args[]){

TestExceptionPropagation1 obj=new TestExceptionPropagation1();

obj.p();

System.out.println("normal flow...");

}

}

In the above example exception occurs in m() method where it is not handled,so it is propagated to previous n() method where it is not handled, again it is propagated to p() method where exception is handled.

Exception can be handled in any method in call stack either in main() method,p() method,n() method or m() method.

Rule: By default, Checked Exceptions are not forwarded in calling chain (propagated).

**Java throws keyword**

The Java **throws** keyword is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

**Syntax of java throws**

**return\_type method\_name() throws exception\_class\_name{**

**//method code }**

**Which exception should be declared**

Ans) checked exception only, because:unchecked Exception: under your control so correct your code.

error: beyond your control e.g. you are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

**throw and throws in Java:**

The **throw** keyword in Java is used to explicitly throw an exception from a method or any block of code. We can throw either checked or unchecked exception. The throw keyword is mainly used to throw custom exceptions.

Syntax:

**throw** Instance

Example:

**throw** new ArithmeticException("/ by zero");

But this exception i.e, Instance must be of type Throwable or a subclass of Throwable. For example Exception is a sub-class of Throwable and user defined exceptions typically extend Exception class. Unlike C++, data types such as int, char, floats or non-throwable classes cannot be used as exceptions.

The flow of execution of the program stops immediately after the throw statement is executed and the nearest enclosing try block is checked to see if it has a catch statement that matches the type of exception. If it finds a match, controlled is transferred to that statement otherwise next enclosing try block is checked and so on. If no matching catch is found then the default exception handler will halt the program.

**throws**

**throws** is a keyword in Java which is used in the signature of method to indicate that this method might throw one of the listed type exceptions. The caller to these methods has to handle the exception using a try-catch block.

Syntax:

type method\_name(parameters) **throws** exception\_list

exception\_list is a comma separated list of all the

exceptions which a method might throw.

In a program, if there is a chance of rising an exception then compiler always warn us about it and compulsorily we should handle that checked exception, Otherwise we will get compile time error saying unreported exception XXX must be caught or declared to be thrown. To prevent this compile time error we can handle the exception in two ways:

**By using try catch**

**By using throws keyword**

We can use throws keyword to delegate the responsibility of exception handling to the caller (It may be a method or JVM) then caller method is responsible to handle that exception.

Example :

public static void main(String[] args)throws InterruptedException

{

Thread.sleep(10000);

System.out.println("Hello Geeks");

}

Explanation : In the above program, by using throws keyword we handled the InterruptedException and we will get the output.

**Important points to remember about throws keyword:**

**throws** keyword is required only for checked exception and usage of throws keyword for unchecked exception is meaningless.

**throws** keyword is required only to convince compiler and usage of throws keyword does not prevent abnormal termination of program.

By the help of **throws** keyword we can provide information to the caller of the method about the exception.

**Difference between throw and throws in Java :**

**throw:**

> Java throw keyword is used to explicitly throw an exception.

> Checked exception cannot be propagated using throw only.

> Throw is followed by an instance.

> Throw is used within the method.

> You cannot throw multiple exceptions.

**throws:**

> Java throws keyword is used to declare an exception.

> Checked exception can be propagated with throws.

> Throws is followed by class.

> Throws is used with the method signature.

> You can declare multiple exceptions e.g.

public void method()throws IOException,SQLException.

**Difference between final, finally and finalize:**

**final** :

Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.

Final is a keyword.

**finally** :

Finally is used to place important code, it will be executed whether exception is handled or not.

Finally is a block.

**finalize**:

Finalize is used to perform clean up processing just before object is garbage collected.

Finalize is a method.

**ExceptionHandling with MethodOverriding in Java**

There are many rules if we talk about methodoverriding with exception handling. The Rules are as follows:

> **If the superclass method does not declare an exception**

If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception.

> **If the superclass method declares an exception**

If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

1. Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception.
2. Rule: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but can declare unchecked exception.
3. Rule: If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.
4. Example in case subclass overridden method declares same exception
5. Example in case subclass overridden method declares subclass exception
6. Example in case subclass overridden method declares no exception

**Java Custom Exception**

If you are creating your own Exception that is known as custom exception or user-defined exception. Java custom exceptions are used to customize the exception according to user need.

By the help of custom exception, you can have your own exception and message.

class InvalidAgeException extends Exception{

InvalidAgeException(String s){

super(s); }}

**Serialization and Deserialization in Java with Example**

Serialization is a mechanism of converting the state of an object into a byte stream.

Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object.



The byte stream created is platform independent. So, the object serialized on one platform can be deserialized on a different platform.

To make a Java object serializable we implement the java.io.**Serializable** interface.

The **ObjectOutputStream** class contains **writeObject**() method for serializing an Object.

public final void writeObject(Object obj)

throws IOException

The **ObjectInputStream** class contains **readObject**() method for deserializing an object.

public final Object readObject()

throws IOException,

ClassNotFoundException

**Advantages of Serialization :**1. To save/persist state of an object.  
2. To travel an object across a network.

Only the objects of those classes can be serialized which are implementing java.io.**Serializable**interface.  
**Serializable** is a marker interface (has no data member and method). It is used to “mark” java classes so that objects of these classes may get certain capability. Other examples of marker interfaces are:- Cloneable and Remote.

Only the objects of those classes can be serialized which are implementing java.io.Serializable interface.  
Serializable is a marker interface (has no data member and method). It is used to “mark” java classes so that objects of these classes may get certain capability. Other examples of marker interfaces are:- Cloneable and Remote.

**Points to remember :**1. If a parent class has implemented Serializable interface then child class doesn’t need to implement it but vice-versa is not true.  
2. Only non-static data members are saved via Serialization process.  
3. Static data members and transient data members are not saved via Serialization process.So, if you don’t want to save value of a non-static data member then make it transient.  
4. Constructor of object is never called when an object is deserialized.  
5. Associated objects must be implementing Serializable interface.

**SerialVersionUID**

The Serialization runtime associates a version number with each Serializable class called a SerialVersionUID, which is used during Deserialization to verify that sender and reciever of a serialized object have loaded classes for that object which are compatible with respect to serialization.

If the reciever has loaded a class for the object that has different UID than that of corresponding sender’s class, the Deserialization will result in an InvalidClassException. A Serializable class can declare its own UID explicitly by declaring a field name.

It must be static, final and of type long.

i.e- ANY-ACCESS-MODIFIER static final long serialVersionUID=42L;

If a serializable class doesn’t explicitly declare a serialVersionUID, then the serialization runtime will calculate a default one for that class based on various aspects of class, as described in Java Object Serialization Specification. However it is strongly recommended that all serializable classes explicitly declare serialVersionUID value, since its computation is highly sensitive to class details that may vary depending on compiler implementations, any change in class or using different id may affect the serialized data.

It is also recommended to use private modifier for UID since it is not useful as inherited member.

**serialver**

The serialver is a tool that comes with JDK. It is used to get serialVersionUID number for Java classes.

You can run the following command to get serialVersionUID

**Example** 1:

// Java code for serialization and deserialization

// of a Java object

import java.io.\*;

class Demo implements java.io.Serializable {

public int a;

public String b;

// Default constructor

public Demo(int a, String b) {

this.a = a;

this.b = b;

} }

class Test{

public static void main(String[] args) {

Demo object = new Demo(1, "geeksforgeeks");

String filename = "file.ser";

// Serialization

try {

//Saving of object in a file

FileOutputStream file = new FileOutputStream(filename);

ObjectOutputStream out = new ObjectOutputStream(file);

// Method for serialization of object

out.writeObject(object);

out.close();

file.close();

System.out.println("Object has been serialized");

} catch(IOException ex) {

System.out.println("IOException is caught");

}

Demo object1 = null;

// Deserialization

try{

// Reading the object from a file

FileInputStream file = new FileInputStream(filename);

ObjectInputStream in = new ObjectInputStream(file);

// Method for deserialization of object

object1 = (Demo)in.readObject();

in.close();

file.close();

System.out.println("Object has been deserialized ");

System.out.println("a = " + object1.a);

System.out.println("b = " + object1.b);

} catch(IOException ex) {

System.out.println("IOException is caught");

} catch(ClassNotFoundException ex) {

System.out.println("ClassNotFoundException is caught");

} }}

**Example** 2:

import java.io.\*;

class Emp implements Serializable {

private static final long serialversionUID =

129348938L;

transient int a; static int b; String name; int age;

// Default constructor

public Emp(String name, int age, int a, int b) {

this.name = name;

this.age = age;

this.a = a;

this.b = b;

} }

public class SerialExample {

public static void printdata(Emp object1) {

System.out.println("name = " + object1.name);

System.out.println("age = " + object1.age);

System.out.println("a = " + object1.a);

System.out.println("b = " + object1.b);

}

public static void main(String[] args) {

Emp object = new Emp("ab", 20, 2, 1000);

String filename = "shubham.txt";

// Serialization

try {

// Saving of object in a file

FileOutputStream file = new FileOutputStream

(filename);

ObjectOutputStream out = new ObjectOutputStream

(file);

// Method for serialization of object

out.writeObject(object);

out.close();

file.close();

System.out.println("Object has been serialized\n"

+ "Data before Deserialization.");

printdata(object);

// value of static variable changed

object.b = 2000;

} catch (IOException ex) {

System.out.println("IOException is caught");

}

object = null;

// Deserialization

try {

// Reading the object from a file

FileInputStream file = new FileInputStream

(filename);

ObjectInputStream in = new ObjectInputStream

(file);

// Method for deserialization of object

object = (Emp)in.readObject();

in.close();

file.close();

System.out.println("Object has been deserialized\n"

+ "Data after Deserialization.");

printdata(object);

// System.out.println("z = " + object1.z);

}catch (IOException ex) {

System.out.println("IOException is caught");

} catch (ClassNotFoundException ex) {

System.out.println("ClassNotFoundException" +

" is caught");

} }}

**Output**:

Object has been serialized

Data before Deserialization.

name = ab

age = 20

a = 2

b = 1000

Object has been deserialized

Data after Deserialization.

name = ab

age = 20

a = 0

b = 2000

**Description** for Output:

You have seen while deserializing the object the values of a and b has changed. The reason being a was marked as **transient** and b was **static**.

In case of **transient** variables:- A variable defined with transient keyword is not serialized during serialization process.This variable will be initialized with default value during deserialization. (e.g: for objects it is null, for int it is 0).

In case of **static** Variables:- A variable defined with static keyword is not serialized during serialization process.This variable will be loaded with current value defined in the class during deserialization.

**Customized Serialization and Deserialization In Java :**

**Why is custom serialization needed?**

During serialization, there may be data loss if we use the ‘transient’ keyword. ‘Transient’ keyword is used on the variables which we don’t want to serialize. But sometimes, it is needed to serialize them in a different manner than the default serialization (such as encrypting before serializing etc.), in that case, we have to use custom serialization and deserialization.

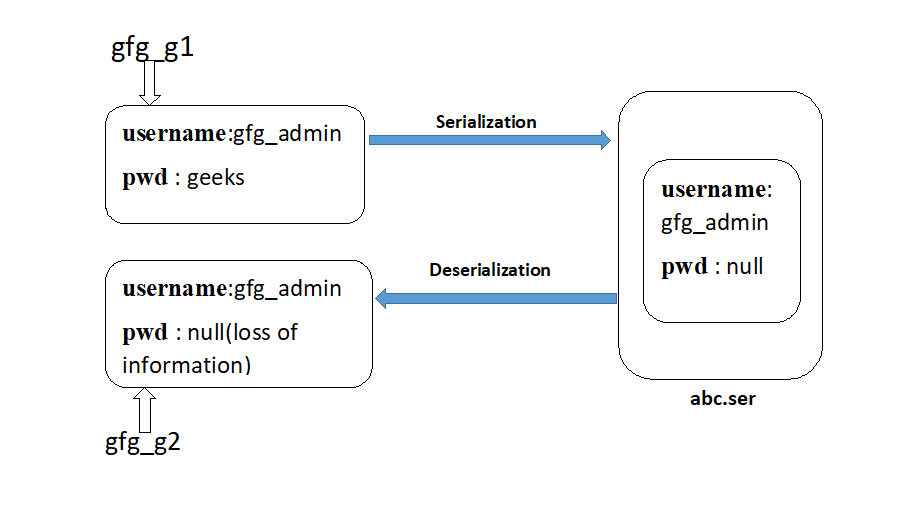
Below program illustrates the above situation of data loss:

|  |
| --- |
| // Java program to illustrate loss of information  // because of transient keyword.  **import** java.io.\*;    **class** GfgAccount **implements** Serializable {        String username = "gfg\_admin";    **transient** String pwd = "geeks";    }    **class** CustomizedSerializationDemo {  **public** **static** **void** main(String[] args) **throws** Exception      {          GfgAccount gfg\_g1 = **new** GfgAccount();            System.out.println("Username : " + gfg\_g1.username +                                   "    Password : " + gfg\_g1.pwd);            FileOutputStream fos = **new** FileOutputStream("abc.ser");            ObjectOutputStream oos = **new** ObjectOutputStream(fos);            // writeObject() method present in GfgAccount class          // will be automatically called by jvm          oos.writeObject(gfg\_g1);            FileInputStream fis = **new** FileInputStream("abc.ser");            ObjectInputStream ois = **new** ObjectInputStream(fis);            // readObject() method present GfgAccount class          // will be automatically called by jvm          GfgAccount gfg\_g2 = (GfgAccount)ois.readObject();            System.out.println("Username : " + gfg\_g2.username +                                 "      Password : " + gfg\_g2.pwd);      }  } |

**Output**:

Username : gfg\_admin Password : geeks

Username : gfg\_admin Password : null



In the above image example, before serialization, Account object can provide proper username and password but deserialization of Account object provides only username and not the password. This is due to declaring password variable as transient.

Hence during default serialization, there may be a chance of loss of information because of the transient keyword. To recover this loss, we will have to use **Customized Serialization.**

Customized serialization can be implemented using the following two methods:

1. **private void writeObject(ObjectOutputStream oos) throws Exception:** This method will be executed automatically by the jvm(also known as Callback Methods) at the time of serialization. Hence to perform any activity during serialization, it must be defined only in this method.
2. **private void readObject(ObjectInputStream ois) throws Exception:** This method will be executed automatically by the jvm(also known as Callback Methods) at the time of deserialization. Hence to perform any activity during deserialization, it must be defined only in this method.

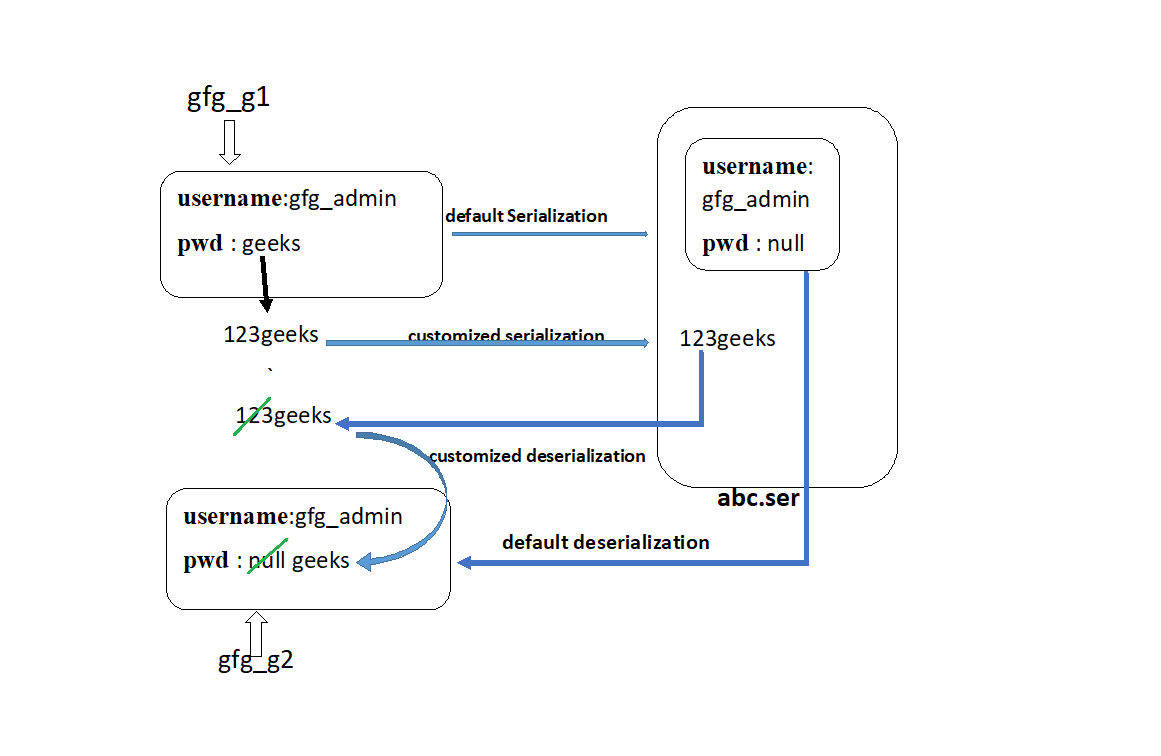
**Note**: While performing object serialization, we have to define the above two methods in that class.

|  |
| --- |
| // Java program to illustrate customized serialization  **import** java.io.\*;    **class** GfgAccount **implements** Serializable {        String username = "gfg\_admin";    **transient** String pwd = "geeks";        // Performing customized serialization using the below two methods:      // this method is executed by jvm when writeObject() on      // Account object reference in main method is      // executed by jvm.  **private** **void** writeObject(ObjectOutputStream oos) **throws** Exception      {          // to perform default serialization of Account object.          oos.defaultWriteObject();            // epwd (encrypted password)          String epwd = "123" + pwd;            // writing encrypted password to the file          oos.writeObject(epwd);      }        // this method is executed by jvm when readObject() on      // Account object reference in main method is executed by jvm.  **private** **void** readObject(ObjectInputStream ois) **throws** Exception      {          // performing default deserialization of Account object          ois.defaultReadObject();            // deserializing the encrypted password from the file          String epwd = (String)ois.readObject();            // decrypting it and saving it to the original password          // string starting from 3rd  index till the last index          pwd = epwd.substring(3);      }  }    **class** CustomizedSerializationDemo {  **public** **static** **void** main(String[] args) **throws** Exception      {          GfgAccount gfg\_g1 = **new** GfgAccount();            System.out.println("Username :" + gfg\_g1.username +                             "       Password :" + gfg\_g1.pwd);            FileOutputStream fos = **new** FileOutputStream("abc.ser");            ObjectOutputStream oos = **new** ObjectOutputStream(fos);            // writeObject() method on Account class will          // be automatically called by jvm          oos.writeObject(gfg\_g1);            FileInputStream fis = **new** FileInputStream("abc.ser");            ObjectInputStream ois = **new** ObjectInputStream(fis);            GfgAccount gfg\_g2 = (GfgAccount)ois.readObject();            System.out.println("Username :" + gfg\_g2.username +                               "       Password :" + gfg\_g2.pwd);      }  } |

**Output**:

Username :gfg\_admin Password :geeks

Username :gfg\_admin Password :geeks



**Object Serialization with Inheritance in Java :**

Serialization is a mechanism of converting the state of an object into a byte stream. Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object.

There are some cases of Serialization with respect to inheritance :

Case 1: **If superclass is serializable then subclass is automatically serializable :** If superclass is Serializable, then by default every subclass is serializable. Hence, even though subclass doesn’t implement Serializable interface( and if it’s superclass implements Serializable), then we can serialize subclass object.

Case 2:  **If a superclass is not serializable then subclass can still be serialized :**Even though superclass doesn’t implements Serializable interface, we can serialize subclass object if subclass itself implements Serializable interface. So we can say that to serialize subclass object, superclass need not to be serializable. But what happens with the instances of superclass during serialization in this case. The following procedure explain this.

**What happens when a class is serializable but its superclass is not ?**

**Serialization**: At the time of serialization, if any instance variable is inheriting from non-serializable superclass, then JVM ignores original value of that instance variable and save default value to the file.

**De- Serialization:** At the time of de-serialization, if any non-serializable superclass is present, then JVM will execute instance control flow in the superclass. To execute instance control flow in a class, JVM will always invoke default(no-arg) constructor of that class. So every non-serializable superclass must necessarily contain default constructor, otherwise we will get runtime-exception.

**Case 3:**

**If the superclass is serializable but we don’t want the subclass to be serialized :** There is no direct way to prevent subclass from serialization in java. One possible way by which a programmer can achieve this is by implementing the writeObject() and readObject() methods in the subclass and needs to throw **NotSerializableException** from these methods. These methods are executed during serialization and de-serialization respectively. By overriding these methods, we are just implementing our own custom serialization.

// By implementing writeObject method,

    // we can prevent

    // subclass from serialization

**private** **void** writeObject(ObjectOutputStream out) **throws** IOException

    {

**throw** **new** NotSerializableException();

    }

    // By implementing readObject method,

    // we can prevent

    // subclass from de-serialization

**private** **void** readObject(ObjectInputStream in) **throws** IOException

    {

**throw** **new** NotSerializableException();

}

**Externalizable interface in Java**

**Externalization** serves the purpose of custom Serialization, where we can decide what to store in stream.  
Externalizable interface present in java.io, is used for Externalization which extends Serializable interface. It consist of two methods which we have to override to write/read object into/from stream which are-   
 // to read object from stream

void **readExternal**(ObjectInput in)

// to write object into stream

void **writeExternal**(ObjectOutput out)

**Key differences between Serializable and Externalizable**

**Implementation** : Unlike Serializable interface which will serialize the variables in object with just by implementing interface, here we have to explicitly mention what fields or variables you want to serialize.

**Methods** : Serializable is marker interface without any methods. Externalizable interface contains two methods: writeExternal() and readExternal().

**Process**: Default Serialization process will take place for classes implementing Serializable interface. Programmer defined Serialization process for classes implementing Externalizable interface.

**Backward Compatibility and Control:** If you have to support multiple versions, you can have full control with Externalizable interface. You can support different versions of your object. If you implement Externalizable, it’s your responsibility to serialize super class.

**public No-arg constructor**: Serializable uses reflection to construct object and does not require no arg constructor. But Externalizable requires public no-arg constructor.

**Externalization Example - :**

|  |
| --- |
| // Java program to demonstrate working of Externalization  // interface  **import** java.io.\*;  **class** Car **implements** Externalizable {  **static** **int** age;      String name;  **int** year;    **public** Car(){          System.out.println("Default Constructor called");      }        Car(String n, **int** y){  **this**.name = n;  **this**.year = y;          age = 10;      }        @Override  **public** **void** writeExternal(ObjectOutput out)  **throws** IOException {          out.writeObject(name);          out.writeInt(age);          out.writeInt(year);      }        @Override  **public** **void** readExternal(ObjectInput in)  **throws** IOException, ClassNotFoundException{          name = (String)in.readObject();          year = in.readInt();          age = in.readInt();      }        @Override **public** String toString(){  **return** ("Name: " + name + "\n"                  + "Year: " + year + "\n"                  + "Age: " + age);      }  }    **public** **class** ExternExample {  **public** **static** **void** main(String[] args)    {          Car car = **new** Car("Shubham", 1995);          Car newcar = **null**;            // Serialize the car  **try** {              FileOutputStream fo                  = **new** FileOutputStream("gfg.txt");              ObjectOutputStream so                  = **new** ObjectOutputStream(fo);              so.writeObject(car);              so.flush();          }**catch** (Exception e) {              System.out.println(e);          }            // Deserializa the car  **try** {              FileInputStream fi                  = **new** FileInputStream("gfg.txt");              ObjectInputStream si                  = **new** ObjectInputStream(fi);              newcar = (Car)si.readObject();          }**catch** (Exception e) {              System.out.println(e);          }            System.out.println("The original car is:\n" + car);          System.out.println("The new car is:\n" + newcar);      }  } |

**Output:**   
Default Constructor called

The original car is:

Name: Shubham

Year: 1995

Age: 10

The new car is:

Name: Shubham

Year: 1995

Age: 10

In the example, the class Car has two methods- **writeExternal** and **readExternal**. So, when we write “Car” object to OutputStream, writeExternal method is called to persist the data. The same applies to the readExternal method.   
When an **Externalizable** object is reconstructed, an instance is created first using the public no-argument constructor, then the readExternal method is called. So, it is mandatory to provide a no-argument constructor.   
When an object implements Serializable interface, is serialized or deserialized, no constructor of object is called and hence any initialization which is implemented in constructor can’t be done.

**How to create Immutable class in Java?**

**Immutable** class means that once an object is created, we cannot change its content. In Java, all the **wrapper** classes (like **Integer**, **Boolean**, **Byte**, **Short**) and **String** class is immutable.

We can create our own immutable class as well.

**Following are the requirements:**

* The class must be declared as final (So that child classes can’t be created)
* Data members in the class must be declared as private (So that direct access is not allowed)
* Data members in the class must be declared as final (So that we can’t change the value of it after object creation)
* A parametrized constructor should initialize all the fields performing a deep copy (So that data members can’t be modified with object reference)
* Deep Copy of objects should be performed in the getter methods (To return a copy rather than returning the actual object reference)
* No setters (To not have the option to change the value of the instance variable)

we have created a final class named Student. It has three final data members, a parameterized constructor and getter methods. Please note that there is no setter method here. Also note that we don’t need to perform deep copy or cloning of data members of wrapper types as they are already immutable.

**import** java.util.HashMap;

**import** java.util.Map;

// An immutable class

**public** **final** **class** Student {

**private** **final** String name;

**private** **final** **int** regNo;

**private** **final** Map<String, String> metadata;

**public** Student(String name, **int** regNo,

                   Map<String, String> metadata)

    {

**this**.name = name;

**this**.regNo = regNo;

        Map<String, String> tempMap = **new** HashMap<>();

**for** (Map.Entry<String, String> entry :

             metadata.entrySet()) {

            tempMap.put(entry.getKey(), entry.getValue());

        }

**this**.metadata = tempMap;

    }

**public** String getName() { **return** name; }

**public** **int** getRegNo() { **return** regNo; }

**public** Map<String, String> getMetadata()

    {

        Map<String, String> tempMap = **new** HashMap<>();

**for** (Map.Entry<String, String> entry :

**this**.metadata.entrySet()) {

            tempMap.put(entry.getKey(), entry.getValue());

        }

**return** tempMap;

    }

}

// Driver class

**class** Test {

**public** **static** **void** main(String[] args)

    {

        Map<String, String> map = **new** HashMap<>();

        map.put("1", "first");

        map.put("2", "second");

        Student s = **new** Student("ABC", 101, map);

        System.out.println(s.getName());

        System.out.println(s.getRegNo());

        System.out.println(s.getMetadata());

        // Uncommenting below line causes error

        // s.regNo = 102;

        map.put("3", "third");

        System.out.println(s.getMetadata()); // Remains unchanged due to deep copy in constructor

        s.getMetadata().put("4", "fourth");

        System.out.println(s.getMetadata()); // Remains unchanged due to deep copy in getter

    }

}

**Java: String is Immutable. What exactly is the meaning?**

**String str = "knowledge";**

This, as usual, creates a string containing "knowledge" and assigns it a reference str.

// assigns a new reference to the

// same string "knowledge"

**String s = str;**

Let’s see how the below statement works:

**str = str.concat(" base");**

This appends a string " base" to str. But wait, how is this possible, since String objects are immutable? Well to your surprise, it is.

When the above statement is executed, the VM takes the value of String str, i.e. "knowledge" and appends " base", giving us the value "knowledge base". Now, since Strings are immutable, the VM can’t assign this value to str, so it creates a new String object, gives it a value "knowledge base", and gives it a reference str.

An important point to note here is that, while the String object is immutable, its reference variable is not. So that’s why, in the above example, the reference was made to refer to a newly formed String object.

At this point in the example above, we have two String objects: the first one we created with value "knowledge", pointed to by s, and the second one "knowledge base", pointed to by str. But, technically, we have three String objects, the third one being the literal "base" in the concat statement.

# **Important** Facts about String and Memory usage

What if we didn’t have another reference s to "knowledge"? We would have lost that String. However, it still would have existed, but would be considered lost due to having no references.

|  |
| --- |
| **import** java.io.\*;    **class** GFG {  **public** **static** **void** main(String[] args)      {          String s1 = "java";          s1.concat(" rules");            // Yes, s1 still refers to "java"          System.out.println("s1 refers to " + s1);      }  } |

**Output**:

s1 refers to java

**What’s happening:**

1. The first line is pretty straightforward: create a new String "java" and refer s1 to it.
2. Next, the VM creates another new String "java rules", but nothing refers to it. So, the second String is instantly lost. We can’t reach it.

The reference variable s1 still refers to the original String "java".

Almost every method, applied to a String object in order to modify it, creates new String object. So, where do these String objects go? Well, these exist in memory, and one of the key goals of any programming language is to make efficient use of memory.

As applications grow, it’s very common for String literals to occupy large area of memory, which can even cause redundancy. So, in order to make Java more efficient, the JVM sets aside a special area of memory called the “[String constant pool](https://www.geeksforgeeks.org/how-to-initialize-and-compare-strings-in-java/)“.

When the compiler sees a String literal, it looks for the String in the pool. If a match is found, the reference to the new literal is directed to the existing String and no new String object is created. The existing String simply has one more reference. Here comes the point of making String objects immutable:

In the String constant pool, a String object is likely to have one or many references. If several references point to same String without even knowing it, it would be bad if one of the references modified that String value. That’s why String objects are immutable.

**How to Initialize and Compare Strings in Java?**

**Initializing Strings in Java**

1. **Direct Initialization(String Constant) :** In this method, a String constant object will be created in String pooled area which is inside heap area in memory. As it is a constant, we can’t modify it, i.e. String class is immutable.  
   Examples:

String str = "GeeksForGeeks";

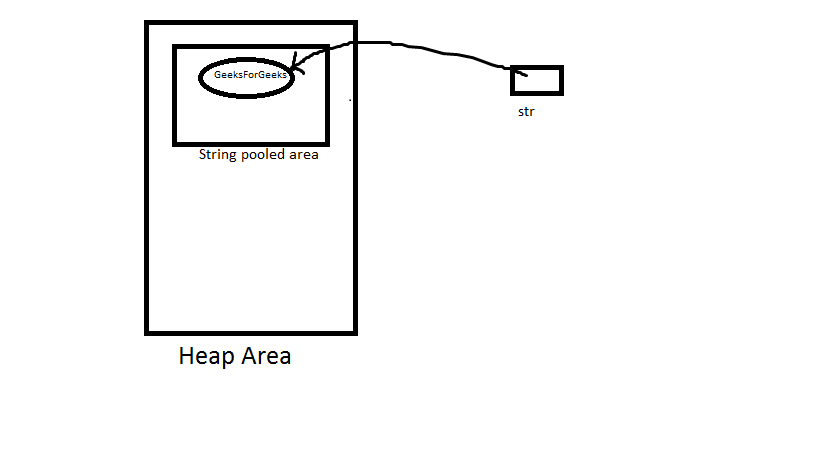
str = "geeks"; // This statement will make str

// point to new String constant("geeks")

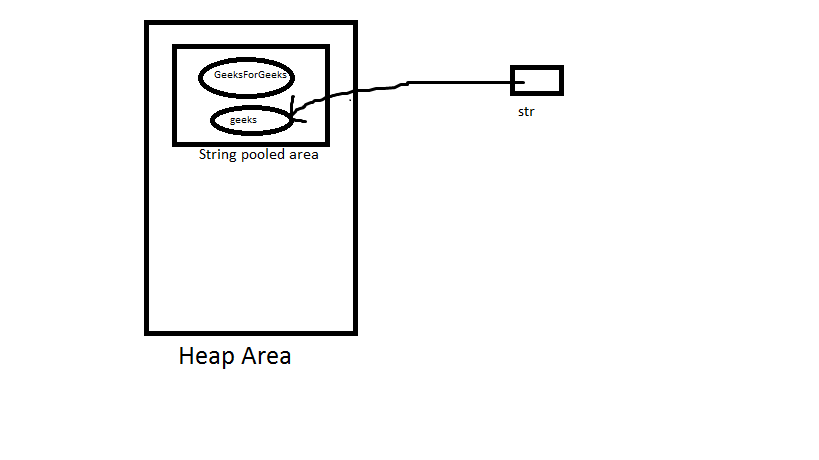
// rather than modifying the previous

// String constant.

**String str = “GeeksForGeeks”;**



**str = “geeks”;**



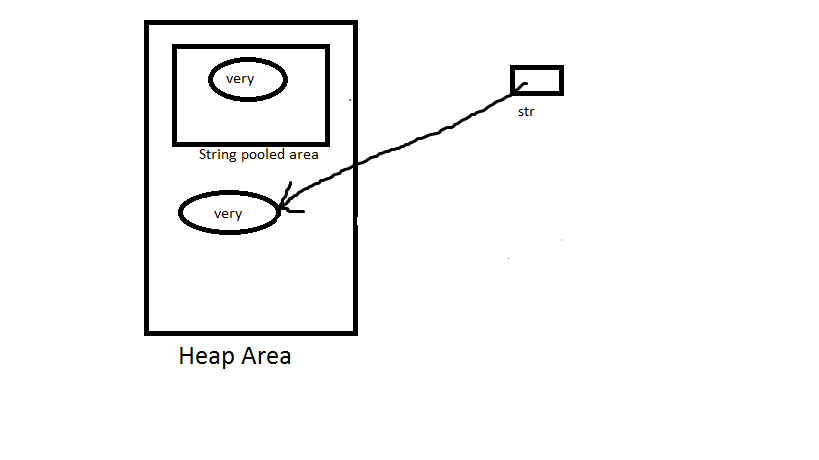
**Note**: If we again write str = “GeeksForGeeks” as next line, then it first check that if given String constant is present in String pooled area or not. If it present then str will point to it, otherwise creates a new String constant.

**2. Object Initialization (Dynamic):** In this method, a String object will be created in heap area (not inside String pooled area as in upper case). We can’t modify it(like in upper case). Also with same value, a String constant is also created in String pooled area, but the variable will point to String object in heap area only.  
Examples:

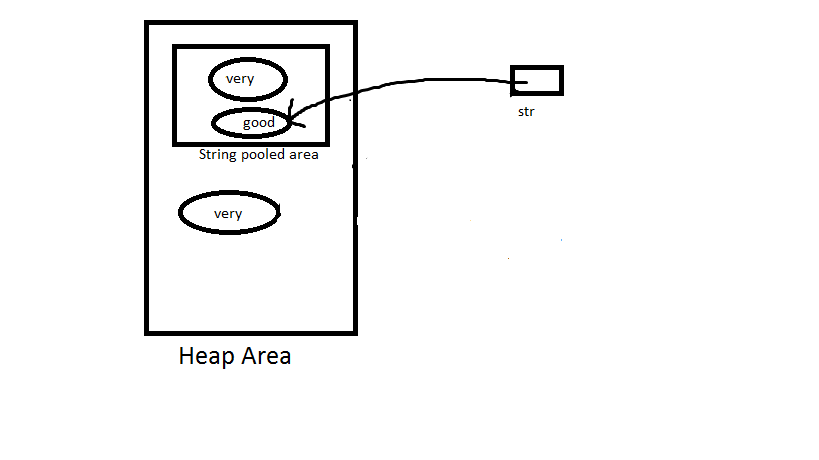
String str = new String("very");

str = "good";

**String str = new String(“very”)**



**str = “good”**  
Now this is a direct assignment, so String constant with value “good” is created in String pooled area and str will point to that.



**Note**: If we again write str = new String(“very”), then it will create a new object with value “very”, rather than pointing to the available objects in heap area with same value.But if we write str = “very”, then it will point to String constant object with value “very”, present in String pooled area.

**Comparing Strings and their References**

1. **equals()** method: It compares values of string for equality. Return type is boolean. In almost all the situation you can use useObjects.equals().

**2. == operator:** It compares references not values. Return type is boolean. == is used in rare situations where you know you’re dealing with interned strings.  
**3. compareTo()** method: It compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.For example, if str1 and str2 are two string variables then if   
str1 == str2 : return 0   
str1 > str2 : return a positive value   
str1 < str2 : return a negative value

**Note**: The positive and negative values returned by compareTo method is the difference of first unmatched character in the two strings.

**Compare two Strings in Java**

String is a sequence of characters. In Java, objects of String are immutable which means they are constant and cannot be changed once created.

**Below are 5 ways to compare two Strings in Java:**

**Using user-defined function :** Define a function to compare values with following conditions :

if (string1 > string2) it returns a positive value.

if both the strings are equal lexicographically

i.e.(string1 == string2) it returns 0.

if (string1 < string2) it returns a negative value.

The value is calculated as (int)str1.charAt(i) – (int)str2.charAt(i)

**Using String.equals() :**In Java, string equals() method compares the two given strings based on the data/content of the string. If all the contents of both the strings are same then it returns true. If any character does not match, then it returns false.

**Syntax:**

**str1.equals(str2);**

Here str1 and str2 both are the strings which are to be compared.

**Using String.equalsIgnoreCase() :** The String.equalsIgnoreCase() method compares two strings irrespective of the case (lower or upper) of the string. This method returns true if the argument is not null and the contents of both the Strings are same ignoring case, else false.

**Syntax:**

**str2.equalsIgnoreCase(str1);**

Here str1 and str2 both are the strings which are to be compared.

**Using Objects.equals() :** Object.equals(Object a, Object b) method returns true if the arguments are equal to each other and false otherwise. Consequently, if both arguments are null, true is returned and if exactly one argument is null, false is returned. Otherwise, equality is determined by using the equals() method of the first argument.

**Syntax:**

**public static boolean equals(Object a, Object b)**

Here a and b both are the string objects which are to be compared.

**Using String.compareTo() :**

Syntax:

int str1.compareTo(String str2)

Working:

It compares and returns the following values as follows:

if (string1 > string2) it returns a positive value.

if both the strings are equal lexicographically

i.e.(string1 == string2) it returns 0.

if (string1 < string2) it returns a negative value.

**Why not to use == for comparison of Strings?**

In general both equals() and “==” operator in Java are used to compare objects to check equality but here are some of the differences between the two:

Main difference between .equals() method and == operator is that one is method and other is operator.

One can use == operators for reference comparison (address comparison) and .equals() method for content comparison.

In simple words, == checks if both objects point to the same memory location whereas .equals() evaluates to the comparison of values in the objects.

Here two String objects are being created namely s1 and s2.

Both s1 and s2 refers to different objects.

When one uses == operator for s1 and s2 comparison then the result is false as both have different addresses in memory.

Using equals, the result is true because its only comparing the values given in s1 and s2.

public class Test {

public static void main(String[] args) {

String s1 = new String("HELLO");

String s2 = new String("HELLO");

System.out.println(s1 == s2); // FALSE

System.out.println(s1.equals(s2)); // TRUE

}

}

**enum in Java:**

Enumerations serve the purpose of representing a group of named constants in a programming language.

For example the 4 suits in a deck of playing cards may be 4 enumerators named Club, Diamond, Heart, and Spade, belonging to an enumerated type named Suit. Other examples include natural enumerated types (like the planets, days of the week, colors, directions, etc.).

Enums are used when we know all possible values at compile time, such as choices on a menu, rounding modes, command line flags, etc. It is not necessary that the set of constants in an enum type stay fixed for all time.

In Java (from 1.5), enums are represented using enum data type. Java enums are more powerful than C/C++ enums .

In Java, we can also add variables, methods and constructors to it.

The main objective of enum is to define our own data types(Enumerated Data Types).

**Declaration of enum in java :**

* Enum declaration can be done outside a Class or inside a Class but not inside a Method.

|  |
| --- |
| // A simple enum example where enum is declared  // outside any class (Note enum keyword instead of  // class keyword)  **enum** Color {      RED, GREEN, BLUE;  }    **public** **class** Test {      // Driver method  **public** **static** **void** main(String[] args)     {          Color c1 = Color.RED;          System.out.println(c1);      } } |

// enum declaration inside a class.

**public** **class** Test {

**enum** Color     {

        RED, GREEN, BLUE;

    }

    // Driver method

**public** **static** **void** main(String[] args)     {

        Color c1 = Color.RED;

        System.out.println(c1);     }}

* First line inside enum should be list of constants and then other things like methods, variables and constructor.
* According to [Java naming conventions](http://www.oracle.com/technetwork/java/codeconventions-135099.html), it is recommended that we name constant with all capital letters

**Important points of enum :**

* Every enum internally implemented by using Class.

/\* internally above enum Color is converted to

class Color{

public static final Color RED = new Color();

public static final Color BLUE = new Color();

public static final Color GREEN = new Color();

}\*/

* Every enum constant represents an object of type enum.
* enum type can be passed as an argument to switch statement.
* Every enum constant is always implicitly public static final. Since it is static, we can access it by using enum Name. Since it is final, we can’t create child enums.
* We can declare main() method inside enum. Hence we can invoke enum directly from the Command Prompt.

**Enum and Inheritance :**

* All enums implicitly extend java.lang.Enum class. As a class can only extend one parent in Java, so an enum cannot extend anything else.
* toString() method is overridden in java.lang.Enum class,which returns enum constant name.
* enum can implement many interfaces.

values(), ordinal() and valueOf() methods :

* These methods are present inside java.lang.Enum.
* values() method can be used to return all values present inside enum.
* Order is important in enums.By using ordinal() method, each enum constant index can be found, just like array index.

valueOf() method returns the enum constant of the specified string value, if exists.

|  |
| --- |
| // Java program to demonstrate working of values(),  // ordinal() and valueOf()  **enum** Color {      RED, GREEN, BLUE;  }    **public** **class** Test{  **public** **static** **void** main(String[] args) {          // Calling values()          Color arr[] = Color.values();            // enum with loop  **for** (Color col : arr) {              // Calling ordinal() to find index              // of color.              System.out.println(col + " at index "                               + col.ordinal());          }            // Using valueOf(). Returns an object of          // Color with given constant.          // Uncommenting second line causes exception          // IllegalArgumentException          System.out.println(Color.valueOf("RED"));          // System.out.println(Color.valueOf("WHITE"));      }  } |

**enum and constructor :**

* enum can contain constructor and it is executed separately for each enum constant at the time of enum class loading.
* We can’t create enum objects explicitly and hence we can’t invoke enum constructor directly.

**enum and methods :**

* enum can contain both concrete methods and abstract methods. If an enum class has an abstract method, then each instance of the enum class must implement it.

|  |
| --- |
| // Java program to demonstrate that enums can have constructor  // and concrete methods.    // An enum (Note enum keyword inplace of class keyword)  **enum** Color {      RED, GREEN, BLUE;        // enum constructor called separately for each      // constant  **private** Color()     {          System.out.println("Constructor called for : " +  **this**.toString());      }    **public** **void** colorInfo()     {          System.out.println("Universal Color");      }  }    **public** **class** Test {      // Driver method  **public** **static** **void** main(String[] args)     {          Color c1 = Color.RED;          System.out.println(c1);          c1.colorInfo();      } } |

**Enum with Customized Value in Java :**

By default enums have their own string values, we can also assign some custom values to enums. Consider below example for that.

Examples:

enum Fruits{

APPLE(“RED”), BANANA(“YELLOW”), GRAPES(“GREEN”);

}

In above example we can see that the Fruits enum have three members i.e APPLE, BANANA and GRAPES with have their own different custom values RED, YELLOW and GREEN respectively.

**Now to use this enum in code, there are some points we have to follow:-**

1. We have to create parameterized constructor for this enum class. Why? Because as we know that enum class’s object can’t be create explicitly so for initializing we use parameterized constructor. And the constructor cannot be the public or protected it must have private or default modifiers. Why? if we create public or protected, it will allow initializing more than one objects. This is totally against enum concept.
2. We have to create one getter method to get the value of enums.

|  |
| --- |
| // Java program to demonstrate how values can  // be assigned to enums.  **enum** TrafficSignal {      // This will call enum constructor with one      // String argument      RED("STOP"), GREEN("GO"), ORANGE("SLOW DOWN");        // declaring private variable for getting values  **private** String action;  **public** String getAction() {  **return** **this**.action;      }        // enum constructor - cannot be public or protected  **private** TrafficSignal(String action) {  **this**.action = action;      }  }    // Driver code  **public** **class** EnumConstructorExample {  **public** **static** **void** main(String args[])     {          // let's print name of each enum and there action          // - Enum values() examples          TrafficSignal[] signals = TrafficSignal.values();    **for** (TrafficSignal signal : signals)         {              // use getter method to get the value              System.out.println("name : " + signal.name() +                          " action: " + signal.getAction() );          }      }  } |

In Java, enum is a special Java type used to define collections of constants. More precisely, a Java enum type is a special kind of Java class. An enum can contain constants, methods etc. enum can be defined as a group of named constant.

There are two ways for making comparison of enum members :

By using == operator

By using equals() method

equals method uses == operator internally to check if two enum are equal. This means, You can compare Enum using both == and equals method.

Difference :

== operator never throws NullPointerException whereas .equals() method can throw NullPointerException.

== is responsible for type compatibility check at compile time whereas .equals() method will never worry about the types of both the arguments.