+------------------------+

| Native Libraries |

+------------------------+

↑

+------------------------+

| Native Interface |

+------------------------+

↑

+----------------------------+

| Execution Engine |

| - Interpreter |

| - JIT Compiler |

| - Garbage Collector |

+----------------------------+

↑

+-------------------------------+

| Runtime Data Area |

| - Method Area |

| - Heap |

| - Stack |

| - PC Register |

| - Native Method Stack |

+-------------------------------+

↑

+---------------------------------+

| Class Loader Subsystem |

+---------------------------------+

↑

+--------------------------+

| Java Program |

+--------------------------+

**JVM Architecture with class loader sub System**

---------------------------------------------------------------

The entire JVM Architecture is divided into 3 sections -

**1) Class Loader sub system**

**2) Runtime Data areas (Memory Areas)**

**3) Execution Engine**

---------------------------------

**Class Loader Sub System**

**---------------------------------**

The main purpose of Class Loader sub system to load the required .class file into JVM Memory from different memory locations.

In order to load the .class file into JVM Memory, It uses an algorithm called "Delegation Hierarchy Algorithm".

Internally, Class Loader sub system performs the following Task

**1) LOADING**

**2) LINKING**

**3) INITIALIZATION**

**========**

**LOADING**

========

In order to load the required .class file, JVM makes a request to class loader sub system. The class loader sub system follows delegation hierarchy algorithm to load the required .class files from different areas.

*To load the required .class file we have 3 different kinds of class loaders.*

***Bootstrap/Primordial class Loader***

---------------------------------------------

It is responsible for loading all the predefined .class files that means all API(Application Programming Interface) level predefined classes are loaded by Bootstrap class loader.

It has the highest priority because Bootstrap class loader is the super class for Platform class loader.

It loads the classes from the following path

C -> Program files -> Java -> JDK -> lib -> jrt-fs.jar

***Platform/Extension class loader***

-------------------------------------------

It is responsible to load the required .class file which is given by some 3rd party in the form of jar file.

It is the sub class of Bootstrap class loader and super class of Application class loader so it has more priority than Application class loader.

It loads the required .class file from the following path.

C -> Program files -> Java -> JDK -> lib -> ext -> ThirdParty.jar

*Command to create the jar file:*

jar cf FileName.jar FileName.class [\*.class]

[If we want to compile more than one java file at a time then the command is:

javac \*.java]

***Application/System class loader***

-------------------------------------------

It is responsible to load all userdefined .class file into JVM memory.

It has the lowest priority because it is the sub class of Platform class loader.

It loads the .class file from class path level or environment variable.

***Note***

If all the class loaders are failed to load the .class file into JVM memory then we will get a Runtime exception i.e java.lang.ClassNotFoundException.

----------------------------------------------------------

***How Delegation Hierarchy algorithm works***

----------------------------------------------------------

Whenever JVM makes a request to class loader sub system to load the required .class file into JVM memory, first of all, class loader sub system makes a request to Application class loader, Application class loader will delegate(by pass) the request to the Extension class loader, Extension class loader will also delegate the request to Bootstrap class loader.

Bootstrap class loader will load the .class file from lib folder(jrt-fs.jar) and then by pass the request back to extension class loader, Extension class loader will load the .class file from ext folder(\*.jar) and by pass the request back to Application class loader, It will load the .class file from environment variable into JVM memory.

***Note***

java.lang.Object is the first class to be loaded into JVM Memory.

----------------------------------------------

**What is Method Chaining in java?**

----------------------------------------------

It is a technique through which we call multiple methods in a single statement.

In this method chaining, always for calling next method we depend upon last method return type.

The final return type of the method depends upon last method call as shown in the program.

MethodChainingDemo1.java

--------------------------------------

package com.ravi.method\_chaining;

public class MethodChainingDemo1 {

public static void main(String[] args)

{

String str = "india";

char ch = str.concat(" is great").toUpperCase().charAt(0);

System.out.println(ch);

}

}

MethodChaningDemo2.java

-----------------------

package com.ravi.method\_chaining;

public class MethodChaningDemo2 {

public static void main(String[] args)

{

String str = "Hyderabad";

int length = str.concat(" is IT City").toUpperCase().length();

System.out.println(length);

}

}

------------------------------------------------------------

***Role of java.lang.Class class in class loading***:

------------------------------------------------------------

There is a predefined class called Class available in java.lang pacakge.

In JVM memory whenever we load a class then it is loaded in special memory called Method Area and return type is java.lang.Class class object.

java.lang.Class cls = AnyClass.class

java.lang.Class class contains a predefined non static method called getName() through which we can get the fully qualified name [Package Name + class Name]

**public String getName()** : Provide fully qualified name of the class.

package com.ravi.class\_loading;

class Student{}

class Sample{}

class Employee{}

class Customer{}

public class ClassLoadingInMethodArea {

public static void main(String[] args)

{

Class cls = Student.class;

System.out.println(cls.getName()); //FQN(PackageName + Class Name)

cls = Sample.class;

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

cls = Employee.class;

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

cls = Customer.class;

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

}

}

-------------------------------------------------------------------------------------------------------------------------

*//WAP that describes Application class loader is responsible to load our user-defined .class*

java.lang.Class class has provided a predefined non static method called getClassLoader(), the return type of this method is ClassLoader class. [Factory Method]

This method will provide the class loader name which is responsible to load the .class file into JVM Memory.

public ClassLoader getClassLoader()

------------------------------------------------

package com.ravi.class\_loading;

class Test

{

}

public class ApplicationClassLoaderDemo {

public static void main(String[] args)

{

System.out.println("Test.class file is loaded by:");

System.out.println(Test.class.getClassLoader());

}

}

--------------------------------------------------------------

*//WAP to show Platform class loader is the super class for Application class loader.*

getClassLoader() method return type is ClassLoader so further we can call any method of ClassLoader class, ClassLoader class has provided a method called **getParent()** whose return type is again ClassLoader only.

public ClassLoader getParent();

-------------------------------

package com.ravi.class\_loading;

class Demo

{

}

public class PlatformClassLoader {

public static void main(String[] args)

{

System.out.println("Super class of Application class loader:");

System.out.println(Demo.class.getClassLoader().getParent());

}

}

-------------------------------------------------------------

package com.ravi.class\_loading;

class Foo

{

}

public class BootStrapClassLoaderDemo {

public static void main(String[] args)

{

System.out.println(Foo.class.getClassLoader().getParent().getParent());

}

}

***Note***

Here we will get the output as null because it is built in class loader for JVM which is used for internal purpose (loading only predefined .class file) so implementation is not provided hence we are getting null.

-------------------------------------------------------------------------------------------------------------------------

**=======**

**LINKING**

=======

***Verify***

***--------***

It ensures the correctness of the .class files, If any suspicious activity is there in the .class file then It will stop the execution immediately by throwing a runtime error i.e java.lang.VerifyError.

There is something called ByteCodeVerifier(Component of JVM), responsible to verify the loaded .class file i.e byte code. Due to this verify module JAVA is highly secure language.

java.lang.VerifyError is the sub class of java.lang.linkageError.

***Prepare***

----------

[Static variable memory allocation + static variable initialization with default value even the variable is final]

It will allocate the memory for all the static data members, here all the static data member will get the default values so if we have static int x = 100; then for variable x memory will be allocated (4 bytes) and now it will initialize with default value i.e 0, even the variable is final.

static Test t = new Test();

Here, t is a static reference variable so for t variable (reference variable) memory will be allocated as per JVM implementation i.e for 32 bit JVM (4 bytes of Memory) and for 64 bit (8 bytes of memory) and initialized with null.

***Resolve***

----------

All the symbolic references (#7) will be converted into direct references OR actual reference.

javap -verbose FileName.class

***Note***

By using above command we can read the internal details of .class file.

***Initialization***

-----------------

Here class initialization will take place. All the static data member will get their actual/original value and we can also use static block for static data member initialization.

Here, in this class initialization phase static variable and static block is having same priority so it will executed according to the order. (Top to bottom)

---------------------------------------------------------------------

*Can we write a Java Program without main method?*

---------------------------------------------------------------------

class WithoutMain

{

static

{

System.out.println("Hello User!!");

System.exit(0);

}

}

It was possible to write a java program without main method till JDK 1.6V. From JDK 1.7v onwards, at the time of loading the .class file JVM will verify the presence of main method in the .class file. If main method is not available then it will generate a runtime error that "main method not found in class".

----------------------------------------

**Static Block OR Static Initializer**

----------------------------------------

It is a special block in java which is automatically executed at the time of loading the .class file.

*Example:*

------------

static

{

}

Static blocks are executed only once because in java we can load the .class files only once.

If we have more than one static block in a class then it will be executed according to the order [Top to bottom]

The main purpose of static block to initialize the static data member of the class so it is also known as static initializer.

In java, a class is not loaded automatically, it is loaded based on the user request so static block will not be executed every time, It depends upon whether class is loaded or not.

static blocks are executed before the main or any static method.

A static blank final field must be initialized inside the static block only.

|  |
| --- |
| static final int A; *//static blank final field*  static  {  A = 100;  } |

A static blank final field also have default value.

We can't write any kind of return statement inside static block.

If we don't declare static variable before static block body execution then we can perform write operation (Initialization is possible due to prepare phase) but read operation is not possible directly otherwise we will get an error Illegal forward reference, it is possible with class name because now compiler knows that variable is coming from class area OR Method area.

*Example:*

|  |
| --- |
| static  {  x = 100; //Valid  System.out.println(x); //Invalid  System.out.println(ClassName.x); //valid  }  static int x; |

**If we directly perform direct read and write operation inside a static method then it is valid.**

|  |
| --- |
| public static void m1()  {  y = 200; //Valid  System.out.println(y); //Valid  }  static int y; |

------------------------------------------------------------

|  |
| --- |
| //static block  class Foo  {  Foo() {  System.out.println("No Argument constructor..");  }  {  System.out.println("Instance block..");  }  static  {  System.out.println("Static block...");  }  }  public class StaticBlockDemo  {  public static void main(String [] args)  {  System.out.println("Main Method Executed ");  new Foo();  new Foo();  new Foo();  }  }  =======================================================================  *Output*  Main Method Executed  Static Block 🡪 Only Once  Instance block  No Argument Constructor  Instance block  No Argument Constructor  Instance block  No Argument Constructor |

Here Foo.class file is not loaded into JVM Memory so static block of Foo class will not be executed.

|  |
| --- |
| class Test  {  static int x;  static  {  x = 100;  System.out.println("x value is :"+x);  }  static  {  x = 200;  System.out.println("x value is :"+x);  }  static  {  x = 300;  System.out.println("x value is :"+x);  }  }  public class StaticBlockDemo1  {  public static void main(String[] args)  {  System.out.println("Main Method");  System.out.println(Test.x);  }  }  ***Note***: *If a class contains more than 1 static block then it will be executed from top to bottom.* |

|  |
| --- |
| class Foo  {  static int x;  static  {  System.out.println("x value is :"+x);  }  }  public class StaticBlockDemo2  {  public static void main(String[] args)  {  new Foo();  }  }  =====================================================  *Output*  x value is: 0  ***Note:*** *static variables are also having default value.* |

|  |
| --- |
| class Demo  {  final static int a; *//Blank static final field*  static  {  a = 100; *//Initialization is compulsory here*  }  }  public class StaticBlockDemo3  {  public static void main(String[] args)  {  System.out.println("a value is :"+Demo.a);  }  } |

|  |
| --- |
| class A  {  static  {  System.out.println("A"); *//Static Block*  }  {  System.out.println("B"); *//Instance Block*  }  A()  {  System.out.println("C"); *//no parameterized Constructor*  }  }  class B extends A  {  static  {  System.out.println("D"); *//Static Block*  }  {  System.out.println("E"); *//Instance Block*  }  B()  {  System.out.println("F"); *//No parameterized Constructor*  }  }  public class StaticBlockDemo4  {  public static void main(String[] args)  {  new B(); *//class loading + Object Creation*  }  }  ***Output***  ADBCEF  ***Execution Flow***  Class Loading Phase:  A (static block) -> "A"  B (static block) -> "D"  Object Creation Phase:  A (instance block) -> "B"  A (constructor) -> "C"  B (instance block) -> "E"  B (constructor) -> "F" |

|  |
| --- |
| *//illegal forward reference*  class Demo  {  static  {  i = 100;  }  static int i;  }  public class StaticBlockDemo5  {  public static void main(String[] args)  {  System.out.println(Demo.i);  }  }  ======================================================================-=  ***Output:*** *0*  ***Execution Flow***  *Class Demoo is loaded*:  Static block executes: i = 100;  Static variable declaration overrides: i = 0;  *Main method executes:*  System.out.println(Demoo.i); prints the value of i, which is 0. |

|  |
| --- |
| class Demo  {  static  {  i = 100;  *//System.out.println(i);* *//Illegal forward reference*  System.out.println(Demo.i);  }  static int i;  }  public class StaticBlockDemo6  {  public static void main(String[] args)  {  System.out.println(Demo.i);  }  } |

|  |
| --- |
| class StaticBlockDemo7  {  static  {  System.out.println("Static Block");  return;  }  public static void main(String[] args)  {  System.out.println("Main Method");  }  }  **Static Block Rules:**   * A static block is a special block of code that is executed when a class is loaded. * It is not a method, so you **cannot use return in a static block** to terminate its execution explicitly.   **Static Blocks and Methods Are Different:**   * Static blocks are executed when the class is loaded, while static methods (like main) are executed explicitly by the JVM or user. |

|  |
| --- |
| public class StaticBlockDemo8  {  *// Declare a final static int variable, but it is not initialized yet.*  final static int x; *// Blank static final field*  *// Static block: runs once when the class is loaded by the JVM*  static  {  *// Initialize the static final field 'x' in the static block*  x = 15; *// Assigning a value to 'x'*  *// Calling the static method 'm1' after initializing 'x'*  m1(); *// This will print the value of 'x'*  }  *// Static method that prints the value of 'x'*  public static void m1()  {  *// Print the value of 'x'. At this point, 'x' is initialized to 15.*  System.out.println("Value of x is :" + x);  }  *// Main method: entry point of the program*  public static void main(String[] args)  {  *// Print the value of 'x' after it has been initialized*  System.out.println("After initialization: " + StaticBlockDemo8.x);  }  } |

|  |
| --- |
| class Test  {  // Static final field initialized when class is loaded  public static final Test t1 = new Test(); // t1 = new Test();  // Static block: runs once when class is loaded  static  {  System.out.println("static block");  }  // Non-static block: runs whenever an object is created  {  System.out.println("Non static block");  }  // Constructor: runs whenever an object is created  Test()  {  System.out.println("No Argument Constructor");  }  }  public class StaticBlockDemo9  {  public static void main(String[] args)  {  // Object creation triggers static/non-static blocks and constructor  new Test(); // 2 steps (class loading + object creation)  }  }  ***Output***  static block  Non static block  No Argument Constructor  Non static block  No Argument Constructor |

|  |
| --- |
| class Sample  {  *// Static block: runs once when the class is loaded*  static  {  System.out.println("Static Block");  x = m1(); *// Assigns the return value of m1() to x*  System.out.println(Sample.x); *// Prints the value of x after assignment*  }  *// Static method m1: prints message and returns 100*  public static int m1()  {  System.out.println("Static Method");  return 100;  }  static int x; *// Static variable x*  }  public class StaticBlockDemo10  {  public static void main(String[] args)  {  System.out.println(Sample.x); *// Prints the value of x from the Sample class*  }  } |

-------------------------------------------------------------

**Variable Memory Allocation and Initialization**

-------------------------------------------------------------

***1) static field OR Class variable***

-----------------------------------------

Memory allocation done at prepare phase of class loading and initialized with default value even variable is final.

It will be initialized with Original value (If provided by user at the time of declaration) at class initialization phase.

When JVM will shutdown then during the shutdown phase class will be un-loaded from JVM memory so static data members are destroyed. They have long life.

----------------------------------------------------

***2) Non static field OR Instance variable***

----------------------------------------------------

Memory allocation done at the time of object creation using new keyword (Instantiation) and initialized as a part of Constructor with default values even the variable is final. [Object class-> at the time of declaration -> instance block -> constructor]

When object is eligible for GC then object is destroyed and all the non static data memebers are also destroyed with corresponding object. It has less life in comparison to static data members becuase they belongs to object.

----------------------

***3) Local Variable***

----------------------

Memory allocation done at stack area (Stack Frame) and developer is responsible to initialize the variable before use. Once metod execution is over, It will be deleted from stack Frame henec it has shortest life.

----------------------------

***4) Parameter variable***

----------------------------

Memory allocation done at stack area (Stack Frame) and end user is responsible to pass the value at runtime. Once metod execution is over, It will be deleted from stack Frame henec it has shortest life.

***Note:*** *We can done validation only one parameter variables.*

--------------------------------------------------------------------------------

**How many ways we can load the .class file into JVM memory ?**

--------------------------------------------------------------------------------

There are so many ways to load the .class file into JVM memory but the following are the common examples:

**1) By using java command**

|  |
| --- |
| public class Test  {  }  javac Test.java  java Test |

Here we are making a request to class loader sub system to load Test.class file into JVM memory

**2) By using Constructor** (new keyword at the time of creating object).

**3) By accessing static data member of the class.**

**4) By using inheritance**

**5) By using Reflection API**

==========================================================================

|  |
| --- |
| *//Program that describes we can load a .class file by using new keyword (Object creation) OR by accessing static data member of the class.*  class Demo  {  static int x = 10;  static  {  System.out.println("Static Block of Demo class Executed!!! :"+x);  }  }  public class ClassLoading  {  public static void main(String[] args)  {  System.out.println("Main Method");  //new Demo();  System.out.println("Hello "+Demo.x);  }  } |

|  |
| --- |
| *//Program that describes whenever we try to load sub class, first of all super class will be loaded. [before parent, child can't exist]*  class Alpha  {  static  {  System.out.println("Static Block of super class Alpha!!");  }  }  class Beta extends Alpha  {  static  {  System.out.println("Static Block of Sub class Beta!!");  }  }  class InheritanceLoading  {  public static void main(String[] args)  {  new Beta();  }  } |

------------------------------------------------------------

**Loading the .class file by using Reflection API**

------------------------------------------------------------

java.lang.Class class has provided a predefined static factory method called forName(String className), It is mainly used to load the given .class file at runtime, The return type of this method is java.lang.Class

public static java.lang.Class forName(String className) throws

ClassNotFoundException

***Note:*** This method throws a checked execption i.e ClassNotFoundException

|  |
| --- |
| package com.ravi.dynamic\_loading;  class Bank  {  static  {  ifscCode = "Bankhyd98675";  System.out.println("IFSC code is :"+Bank.ifscCode);  }  static String ifscCode;  }  public class DynamicLoading  {  public static void main(String[] args) throws ClassNotFoundException  {  System.out.println("Main");  Class.forName("com.ravi.dynamic\_loading.Bank"); //FQN  }  }  ***Note:*** *From the above program it is clear that Class.forName(String className) is used to load the given .class file dynamically at runtime.* |

------------------------------------------------------------------------------------------------

***What is the difference between java.lang.ClassNotFoundException and java.lang.NoClassDefFoundError***

-----------------------------------------------

***java.lang.ClassNotFoundException***

-----------------------------------------------

It occurs when we try to load the required .class file at RUNTIME by using Class.forName(String className) statement or loadClass() static of ClassLoader class and if the required .class file is not available at runtime then we will get an exception i.e java.lang.ClassNotFoundException

***Note:*** *It does not have any concern at compilation time, at run time, JVM will simply verify whether the required .class file is available or not available.*

class Foo

{

static

{

System.out.println("static block of Foo class");

}

}

public class ClassNotFoundExceptionDemo

{

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

{

Class.forName("Ravi");

}

}

***Note:*** *At runtime, JVM will verify whether Ravi.class file is available Or not available, if not available then we will get java.lang.ClassNotFoundException.*

-------------------------------------------

***java.lang.NoClassDefFoundError***

-------------------------------------------

It occurs when the class was present at the time of COMPILATION but at runtime the required .class file is not available(manualy deleted by user ) Or it is not available in the current directory (Misplaced) then we will get a runtime error i.e java.lang.NoClassDefFoundError.

class Welcome

{

public void greet()

{

System.out.println("Hello Batch 40");

}

}

public class NoClassDeFFoundErrorDemo

{

public static void main(String[] args)

{

Welcome w = new Welcome();

w.greet();

}

}

***Note:*** *After compilation delete Welcome.class OR move this Welcome .class file from current folder to any other folder.*

*---------------------------------------------------------------------------------------*

\*\* **A static method does not act on instance variable directly why?**

---------------------------------------------------------------------------------------

All the static members (static variable, static block, static method, static nested inner class) are loaded/executed at the time of loading the .class file into JVM Memory.

At class loading phase object is not created because object is created in the 2nd phase i.e Runtime data area so at the TIME OF EXECUTION OF STATIC METHOD AT CLASS LOADING PAHSE, NON STATIC VARIABLE WILL NOT BE AVAILABLE BY DEFAULT henec we can't access non static variable from static context[static block, static method and static nested inner class] without creating the object.

------------------------------------------------------------ 08/09/2001

public class StaticDemo

{

int x = 100; //2nd Layer Data

public static void main(String[] args) //1st Layer Member

{

System.out.println("x value is :"+x);//error

}

}

--------------------------------------------------------------

class Test

{

private int x;

public Test(int x)

{

this.x = x;

}

public static void access()

{

System.out.println("x value is :"+x); //error

}

}

public class StaticDemo1

{

public static void main(String[] args)

{

Test t1 = new Test(10);

Test.access();

}

}

-------------------------------------------------------------

How to access super class and sub class variable using Variable hiding concept through static method

package com.ravi.execution;

class Alpha

{

protected int x = 100;

}

class Beta extends Alpha

{

protected int x = 200; //Variable Hiding

public static void access()

{

Beta b1 = new Beta();

System.out.println("Sub class x variable :"+b1.x);

Alpha a1 = b1;

System.out.println("Super class x variable is :"+a1.x);

}

}

public class IQ

{

public static void main(String[] args)

{

Beta.access();

}

}

------------------------------------------------------------

Runtime Data Areas :

---------------------

It is also known as Memory Area.

Once a class is loaded then based on variable type method type it is divided into different memory areas which are as follows :

1) Method Area

2) HEAP Area

3) Stack Area

4) PC Register

5) Native Method Stack

Method Area :

-------------

Whenever a class is loaded then the class is dumpped inside method area and returns java.lang.Class class object.

It provides all the information regarding the class, like name of the class, name of the package, static and non static fields available in the class, methods available in the class and so on.

We have only one method area per JVM that means for a single JVM we have only one Method area.

This Method Area OR Class Area is sharable by all the objects.

--------------------------------------------------------------

Program to Show From Method Area we can get complete information of the class. (Reflection API)

package com.ravi.class\_description;

import java.lang.reflect.Field;

import java.lang.reflect.Method;

public class ClassDescription {

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

{

Class cls = Class.forName(args[0]); //FQN

System.out.println("class name is :"+cls.getName());

System.out.println("Package Name is :"+cls.getPackageName());

System.out.println("Available methods are :");

Method[] methods = cls.getDeclaredMethods();

int count = 0;

for(Method method : methods)

{

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

count++;

}

System.out.println("Total number of methods are :"+count);

System.out.println("Available Fields are :");

Field[] fields = cls.getDeclaredFields();

count = 0;

for(Field field : fields)

{

count++;

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

}

System.out.println("Total number of Fields are :"+count);

}

}

Note :- getDeclaredMethods() is a predefined non static method available in java.lang.Class class , the return type of this method is Method array where Method is a predefined class available in java.lang.reflect sub package.

public Method[] getDeclaredMethods()

getDeclaredFields() is a predefined non static method available in java.lang.Class class , the return type of this method is Field array where Field is a predefined class available in java.lang.reflect sub package.

public Field[] getDeclaredFields()

Field and Method both the classes are providing getName() method to get the name of the field and Method.

--------------------------------------------------------------

HEAP AREA :

-----------

Whenever we create an object in java then the properties and behavior of the object are strored in a special memory area called HEAP AREA.

We have only one HEAP AREA per JVM.

--------------------------------------------------------------

STACK Area :

------------

All the methods are executed as a part of Stack Area.

Whenever we call a method in java then internally one stack Frame will be created to hold method related information.

Every Stack frame contains 3 parts :

1) Local Variable arrays

2) Frame Data

3) Operand Stack.

We have multiple stack area for a single JVM.

Everytime we create a thread in java then JVM will create a separate Runtime Stack.[Multithreading]

-------------------------------------------------------------

HEAP and STACK Diagram for Static data member and array variables [Beta.java]

--------------------

class Alpha

{

int val;

static int sval = 200;

static Beta b = new Beta();

public Alpha(int val)

{

this.val = val;

}

}

public class Beta

{

public static void main(String[] args)

{

Alpha am1 = new Alpha(9);

Alpha am2 = new Alpha(2);

Alpha []ar = fill(am1, am2);

ar[0] = am1;

System.out.println(ar[0].val);

System.out.println(ar[1].val);

}

public static Alpha[] fill(Alpha a1, Alpha a2)

{

a1.val = 15;

Alpha fa[] = new Alpha[]{a2, a1};

return fa;

}

}

--------------------------------------------------------------

PC Register :

-------------

It stands for Program counter Register.

In order to hold the current executing instruction of running thread we have separate PC register for each and every thread.

--------------------------------------------------------------

Native Method Stack :

---------------------

Native method means, the java methods which are written by using native languages like C and C++. In order to write native method we need native method library support.

Native method stack will hold the native method information in a separate stack.

---------------------------------------------------------------------

Execution Engine : [Interpreter + JIT Compiler]

Interpreter

------------

In java, JVM contains an interpreter which executes the program line by line. Interpreter is slow in nature because at the time of execution if we make a mistake at line number 9 then it will throw the execption at line number 9 and after solving the execption again it will start the execution from line number 1 so it is slow in execution that is the reason to boost up the execution java software people has provided JIT compiler.

JIT Compiler :

--------------

It stands for just in time compiler. The main purpose of JIT compiler to boost up the execution so the execution of the program will be completed as soon as possible.

JIT compiler holds the repeated instruction like method signature, variables, native method code and make it available to JVM at the time of execution so the overall execution becomes very fast.

--------------------------------------------------------------

HAS-A Relation :

----------------

class Order

{

private int orderId;

private String itemName;

private double itemPrice;

}

class Customer

{

private int customerId;

private String customerName;

private String customerAddree;

private Order order; //HAS-A relation

}

If we use a class as a property to another class then it is called HAS-A Relation.

The limitation of IS-A relation is, it is tightly coupled relation i.e IS-A type relation so if we modify the content of super class then it will automatically, It will reflect to all the sub classes.

While working with HAS-A relation we can access the property of another class so HAS-A relation provides accessibility

feature.

HAS-A relation we can achieve by using Association concept.

Association is divided into two types :

1) Composition (Strong Reference)

2) Aggregation (Weak Reference)

Association :

---------------

Association is a connection between two separate classes that can be built up through their Objects.

The association builds a relationship between the classes and describes how much a class knows about another class.

This relationship can be unidirectional or bi-directional. In Java, the association can have one-to-one, one-to-many, many-to-one and many-to-many relationships.

Example:-

One to One: A person can have only one PAN card

One to many: A Bank can have many Employees

Many to one: Many employees can work in single department

Many to Many: A Bank can have multiple customers and a customer can have multiple bank accounts.

3 files :

---------

Student.java

-------------

package com.ravi.has\_a\_reln;

public class Student

{

private int studentId;

private String studentName;

private int studentMarks;

public Student(int studentId, String studentName, int studentMarks) {

super();

this.studentId = studentId;

this.studentName = studentName;

this.studentMarks = studentMarks;

}

@Override

public String toString() {

return "Student [studentId=" + studentId + ", studentName=" + studentName + ", studentMarks=" + studentMarks

+ "]";

}

public int getStudentId() {

return studentId;

}

public void setStudentId(int studentId) {

this.studentId = studentId;

}

public String getStudentName() {

return studentName;

}

public void setStudentName(String studentName) {

this.studentName = studentName;

}

public int getStudentMarks() {

return studentMarks;

}

public void setStudentMarks(int studentMarks)

{

this.studentMarks = studentMarks;

}

}

--------------------------------------------------------------

Trainer.java

-------------

package com.ravi.has\_a\_reln;

import java.util.Scanner;

public class Trainer

{

public static void viewStudentProfile(Student obj)

{

Scanner sc = new Scanner(System.in);

System.out.print("Enter Student Id :");

int id = sc.nextInt();

if(id == obj.getStudentId())

{

System.out.println(obj);

}

else

{

System.err.println("Id is invalid !!!");

}

}

}

AssociationDemo.java

---------------------

package com.ravi.has\_a\_reln;

public class AssociationDemo {

public static void main(String[] args)

{

Student raj = new Student(111, "Raj", 455);

Student priya = new Student(222, "Priya", 460);

Trainer.viewStudentProfile(raj);

Trainer.viewStudentProfile(priya);

}

}

-------------------------------------------------------------

Composition (Strong reference) :

--------------------------------

Composition in Java is a way to design classes such that one class contains an object of another class. It is a way of establishing a "HAS-A" relationship between classes.

Composition represents a strong relationship between the containing class and the contained class.If the containing object (Car object) is destroyed, all the contained objects (Engine object) are also destroyed.

A car has an engine. Composition makes strong relationship between the objects. It means that if we destroy the owner object, its members will be also destroyed with it. For example, if the Car is destroyed the engine will also be destroyed as well.

Program Guidelines :

--------------------

1) One object can't exist without another object

2) We will not create two separate objects, during the creation of Car object, Engine object should be automatically created.

3) We can declare blank final field.

3 files :

----------

Engine.java

------------

package com.ravi.composition;

public class Engine

{

private String engineType;

private int horsePower;

public Engine(String engineType, int horsePower)

{

super();

this.engineType = engineType;

this.horsePower = horsePower;

}

@Override

public String toString() {

return "Engine [engineType=" + engineType + ", horsePower=" + horsePower + "]";

}

}

Car.java

--------

package com.ravi.composition;

public class Car

{

private String carName;

private int carModel;

private final Engine engine; // HAS-A Relation [Blank final field]

public Car(String carName, int carModel)

{

super();

this.carName = carName;

this.carModel = carModel;

this.engine = new Engine("Battery", 1200); //Composition

}

@Override

public String toString()

{

return "Car [carName=" + carName + ", carModel=" + carModel + ", engine=" + engine + "]";

}

}

CompositionDemo.java

---------------------

package com.ravi.composition;

public class CompositionDemo {

public static void main(String[] args)

{

Car car = new Car("Ford", 2024);

System.out.println(car);

}

}

--------------------------------------------------------------

05-12-2024

Aggregation (Weak Referance) :

------------------------------

Aggregation in Java is another form of association between classes that represents a "HAS-A" relationship, but with a weaker bond compared to composition.

In aggregation, one class contains an object of another class, but the contained object can exist independently of the container. If the container object is destroyed, the contained object can still exist.

3 files :

----------

College.java

-------------

package com.ravi.aggregation;

public class College

{

private String collegeName;

private String collegeLocation;

public College(String collegeName, String collegeLocation)

{

super();

this.collegeName = collegeName;

this.collegeLocation = collegeLocation;

}

@Override

public String toString() {

return "College [collegeName=" + collegeName + ", collegeLocation=" + collegeLocation + "]";

}

}

Student.java

------------

package com.ravi.aggregation;

public class Student

{

private int studentId;

private String studentName;

private int studentMarks;

private College college; // HAS-A Relation

public Student(int studentId, String studentName, int studentMarks, College college)

{

super();

this.studentId = studentId;

this.studentName = studentName;

this.studentMarks = studentMarks;

this.college = college;

}

@Override

public String toString() {

return "Student [studentId=" + studentId + ", studentName=" + studentName + ", studentMarks=" + studentMarks

+ ", college=" + college + "]";

}

}

AggregationDemo.java

---------------------

package com.ravi.aggregation;

public class AggregationDemo {

public static void main(String[] args)

{

College clg = new College("VIT", "Vellore");

Student s1 = new Student(1, "Scott", 450, clg);

System.out.println(s1);

Student s2 = new Student(2, "Smith", 452, clg);

System.out.println(s2);

}

}

Note :- IS-A relation is tightly coupled relation so if we modify the content of super class, sub class content will also modify but in HAS-A realtion we are accessing the properties of another class so we are not allowed to modify the content, we can access the content or Properties.

-------------------------------------------------------------

Assignemnts :

-------------

Order and Customer

Person and Address

Account and Customer

----------------------------------------------------------------

Description of System.out.println() :

-------------------------------------

public class System

{

public final static java.io.PrintStream out = null; //HAS-A Relation

}

System.out.println();

Internally System.out.println() creates HAS-A relation because System class contains a predefined class called java.io.PrintStream as shown in the above example.

The following program describes that how System.out.println() works internally :

Description.java

-----------------

package com.ravi.aggregation;

class Test

{

public static final String str = "Hyderabad";

}

public class Description {

public static void main(String[] args)

{

System.out.println(Test.str.length());

}

}

----------------------------------------------------------------

\*\*\*Polymorphism :

-----------------

Poly means "many" and morphism means "forms".

It is a Greek word whose meaning is "same object having different behavior".

In our real life a person or a human being can perform so many task, in the same way in our programming languages a method or a constructor can perform so many task.

Eg:-

void add(int a, int b)

void add(int a, int b, int c)

void add(float a, float b)

void add(int a, float b)

----------------------------------------------------------------

Types of Polymorphism :

-----------------------

Polymorphism can be divided into two types :

1) Static polymorphism OR Compile time polymorphism OR Early binding

2) Dynamic Polymorphism OR Runtime polymorphism OR Late binding

1) Static Polymorphism :

------------------------

The polymorphism which exist at the time of compilation is called Static OR compile time polymorphism.

In static polymorphism, compiler has very good idea that which method is invoked depending upon METHOD PARAMETER.

Here the binding of the method is done at compilation time so, it is known as early binding.

We can achieve static polymorphism by using Method Overloading concept.

Example of static polymorphism : Method Overloading.

2) Dynamic Polymorphism

-----------------------

The polymorphism which exist at runtime is called Dynamic polymorphim Or Runtime Polymorphism.

\*Here compiler does not have any idea about method calling, at runtime JVM will decide which method will be invoked depending upon CLASS TYPE OBJECT.

Here method binding is done at runtime so, it is also called Late Binding.

We can achieve dynamic polymorphism by using Method Overriding.

Example of Dynamic Polymorphism : Method Overriding

----------------------------------------------------------------

Method Overloading :

-------------------

Method Overloading :

--------------------

Writing two or more methods in the same class or even in the super and sub class in such a way that the method name must be same but the argument must be different.

While Overloading a method we can change the return type of the method.

If parameters are same but only method return type is different then it is not an overloaded method.

Method overloading is possible in the same class as well as super and sub class.

While overloading the method the argument must be different otherwise there will be ambiguity problem.

Method Overloading allows us to write two methods with same name but differ in:

1. Number of parameters

2. Data type of parameters

3. Sequence/Order of data type of parameters(int -long and long int)

IQ :

----

Can we overload the main method/static method ?

Yes, we can overload the main method OR static method but the execution of the program will start from main method which accept String [] array as a parameter.

Note :- The advantage of method overloading is same method name we can reuse for different functionality for refinement of the method.

Note :- In System.out.println() or System.out.print(), print()

and println() methods are best example for Method Overloading.

Example :

----------

public void makePayment(Cash c)

{

}

public void makePayment(UPI c)

{

}

public void makePayment(CreditCard c)

{

}

------------------------------------------------------------------

WAP to show Constructor Overloading :

--------------------------------------

package com.ravi.overload;

class Addition

{

public Addition(int x, int y)

{

this(2.3F, 7.8f);

System.out.println("Sum of two integer is :"+(x+y));

}

public Addition(float x, float y)

{

this("Data","base");

System.out.println("Sum of two float is :"+(x+y));

}

public Addition(String x, String y)

{

super();

System.out.println("Concatenation of two String is :"+(x+y));

}

}

public class OverloadDemo1

{

public static void main(String[] args)

{

new Addition(100,200);

}

}

------------------------------------------------------------------

WAP to show method overloading by changing the return type :

-------------------------------------------------------------

package com.ravi.overload;

class Sum

{

public int add(int x, int y)

{

return (x+y);

}

public double add(double x, double y)

{

return (x+y);

}

public String add(String x, String y)

{

return (x+y);

}

}

public class MethodOverloading

{

public static void main(String[] args)

{

Sum s1 = new Sum();

System.out.println(s1.add("Data", "base"));

System.out.println(s1.add(12.90, 45.89));

System.out.println(s1.add(10, 20));

}

}

----------------------------------------------------------------

Var-Args in Java :

--------------------

It was introduced from JDK 1.5 onwards.

It stands for variable argument. It is an array variable which can hold 0 to n number of parameters of same type or different type by using Object class.

It is represented by exactly 3 dots (...) so it can accept any number of argument (0 to nth) that means now we need not to define method body again and again, if there is change in method parameter value.

var-args must be only one and last argument.

We can use var-args as a method/Constructor parameter only.

-------------------------------------------------------------------

WAP to show that var-args can accept 0 to n number of parameters

package com.ravi.overload;

class Test

{

public void input(int ...x)

{

System.out.println("Var Args executed");

}

}

public class VarArgsDemo1

{

public static void main(String... args)

{

Test t1 = new Test();

t1.input();

t1.input(10);

t1.input(10,20);

t1.input(10,20,30);

t1.input(100,200,300,400);

}

}

------------------------------------------------------------------

WAP to show how to add parameter values using var args :

package com.ravi.overload;

class SumOfParameter

{

public void acceptParameter(int ...values)

{

int sum = 0;

for(int value : values)

{

sum = sum + value;

}

System.out.println("Sum of parameter is :"+sum);

}

}

public class VarArgsDemo2

{

public static void main(String[] args)

{

SumOfParameter s1 = new SumOfParameter();

s1.acceptParameter(10,20,30,40);

s1.acceptParameter(100,200,300,400);

}

}

------------------------------------------------------------------

WAP to show var args must be only one and last argument

package com.ravi.overload;

class Sample

{

// All commented codes are invalid

/\*

\* public void accept(float ...x, int ...y) { }

\*

\* public void accept(int ...x, int y) { }

\*

\* public void accept(int...x, int ...y) {}

\*/

public void accept(int x, int... y)

{

System.out.println("x value is :" + x);

for (int z : y)

{

System.out.println(z);

}

}

}

public class VarArgsDemo3

{

public static void main(String[] args)

{

Sample s1 = new Sample();

s1.accept(12, 10,20,30,40);

}

}

------------------------------------------------------------------

Program that show var args can accept hetrogeneous types of data.

-----------------------------------------------------------------

package com.ravi.overload;

class Accept

{

public void acceptHetrogeneous(Object ...obj)

{

for(Object o : obj)

{

System.out.println(o);

}

}

}

public class VarArgsDemo4 {

public static void main(String[] args)

{

Accept a1 = new Accept();

a1.acceptHetrogeneous(12,23.90,'A',"NIT", new String("Hyd"));

}

}

----------------------------------------------------------------

09-12-2024

----------

Wrapper classes in java :

-------------------------

We have 8 primitive data types in java i.e byte, short, int, long and so on.

Except these 8 primitive data types, everything in java is an object.

If we remove these 8 primitive data types then only java can become pure object oriented language.

On these primitive data types, we can't assign null or we can't invoke a method.

These primitive data types are unable to move in the network, only objects are moving in the network.

We can't perform serialization and object cloning on primitive data types.It is only possible with objects.

To avoid the above said problems, From JDK 1.5v, java software people has provided the following two concepts :

a) Autoboxing

b) Unboxing

Autoboxing

--------------

When we convert the primitive data types into corresponding wrapper object then it is called Autoboxing as shown below.

Primitive type Wrapper Object

---------------- ------------------

byte - Byte

short - Short

int - Integer

long - Long

float - Float

double - Double

char - Chracter

boolean - Boolean

Note : ALL THE WRAPPER CLASSES ARE IMMUTABLE(UN-CHANGED) AS WELL AS equals(Object obj) and hashCode() methods are overridden in all the Wrapper classes.

WAP to show that Wrapper classes are immutable :

-------------------------------------------------

package com.ravi.immutable\_demo;

class Product

{

protected double price;

public Product(double price)

{

super();

this.price = price;

}

public double getPrice() {

return price;

}

public void setPrice(double price) {

this.price = price;

}

@Override

public String toString()

{

return "Product [price=" + price + "]";

}

}

public class ImmutableDemo1

{

public static void main(String[] args)

{

Product laptop = new Product(67000);

System.out.println("Before Change :"+laptop.getPrice());

accept(laptop);

System.out.println("After Change :"+laptop.getPrice());

}

public static void accept(Product prod)

{

prod.setPrice(90000);

}

}

-------------------------------------------------------------

package com.ravi.immutable\_demo;

public class ImmutableDemo2

{

public static void main(String[] args)

{

Integer i = new Integer(90);

accept(i);

System.out.println(i); //90

}

public static void accept(Integer y)

{

y = 120;

}

}

---------------------------------------------------------------

package com.ravi.immutable\_demo;

public class ImmutableDemo3

{

public static void main(String[] args)

{

String str = "Data";

accept(str);

System.out.println(str); //Data

}

public static void accept(String s1)

{

s1 = "Data base";

}

}

----------------------------------------------------------------

--

Overloaded valueOf() method :

-----------------------------

We have 3 overloaded valueOf() method :

----------------------------------------

1) public static Integer valueOf(int x) : It will convert the given int value into Integer Object.

2) public static Integer valueOf(String str) : It will convert

the given String into Integer Object.

[valueOf() method will convert the String into Wrapper object where as parseInt() method will convet the String into primitive type]

3) public static Integer valueOf(String str, int radix/base) :

It will convert the given String number into Integer object

by using the specified radix or base.

Note :- We can pass base OR radix upto 36

i.e A to Z (26) + 0 to 9 (10) -> [26 + 10 = 36], It can be

calculated by using Character.MAX\_RADIX.

Output will be generated on the basis of radix

System.out.println(Character.MAX\_RADIX); //36

MAX\_RADIX is a final and static variable of Character class.

----------------------------------------------------------------

//Integer.valueOf(int);

public class AutoBoxing1

{

public static void main(String[] args)

{

int a = 12;

Integer x = Integer.valueOf(a); //Upto 1.4 version

System.out.println(x);

int y = 15;

Integer i = y; //From 1.5 onwards compiler takes care

System.out.println(i);

}

}

---------------------------------------------------------------

public class AutoBoxing2

{

public static void main(String args[])

{

byte b = 12;

Byte b1 = Byte.valueOf(b);

System.out.println("Byte Object :"+b1);

short s = 17;

Short s1 = Short.valueOf(s);

System.out.println("Short Object :"+s1);

int i = 90;

Integer i1 = Integer.valueOf(i);

System.out.println("Integer Object :"+i1);

long g = 12;

Long h = Long.valueOf(g);

System.out.println("Long Object :"+h);

float f1 = 2.4f;

Float f2 = Float.valueOf(f1);

System.out.println("Float Object :"+f2);

double k = 90.90;

Double l = Double.valueOf(k);

System.out.println("Double Object :"+l);

char ch = 'A';

Character ch1 = Character.valueOf(ch);

System.out.println("Character Object :"+ch1);

boolean x = true;

Boolean x1 = Boolean.valueOf(x);

System.out.println("Boolean Object :"+x1);

}

}

---------------------------------------------------------------

//Integer.valueOf(String str)

//Integer.valueOf(String str, int radix/base)

public class AutoBoxing3

{

public static void main(String[] args)

{

Integer a = Integer.valueOf(15);

Integer b = Integer.valueOf("25");

Integer c = Integer.valueOf("111",36); //Here Base we can take upto 36

System.out.println(a);

System.out.println(b);

System.out.println(c);

}

}

----------------------------------------------------------------

public class AutoBoxing4

{

public static void main(String[] args)

{

Integer i1 = new Integer(100);

Integer i2 = new Integer(100);

System.out.println(i1==i2); //false

Integer a1 = Integer.valueOf(15);

Integer a2 = Integer.valueOf(15);

System.out.println(a1==a2); //true

}

}

Note : It will generate Compilation warning because new Integer(int val) is marked for removal.

a1 and a2 reference variable, both are pointing to the same object (immutable) so == operator will provide

true.

----------------------------------------------------------------

How to convert int value into String :

--------------------------------------

Integer class has provided a predefined static method

toString() through which we can convert the int value into String Object

public static String toString()

//Converting integer value to String

public class AutoBoxing5

{

public static void main(String[] args)

{

int x = 12;

String str = Integer.toString(x);

System.out.println(str+2);

}

}

----------------------------------------------------------------

Unboxing :

----------------

Converting wrapper object to corresponding primitive type is called Unboxing.

Wrapper Primitive

Object type

---------- ----------

Byte - byte

Short - short

Integer - int

Long - long

Float - float

Double - double

Chracter - char

Boolean - boolean

----------------------------------------------------------------

We have total 8 Wrapper classes.

Among all these 8, 6 Wrapper classes (Byte, Short, Integer, Long, Float and Double) are the sub class of java.lang.Number class which represent numbers (either decimal OR non decimal)

so all the following six wrapper classes (Which are sub class of Number class) are providing the following common methods.

1) public byte byteValue()

2) public short shortValue()

3) public int intValue()

4) public long longValue()

5) public float floatValue()

6) public double doubleValue()

----------------------------------------------------------------

//Converting Wrapper object into primitive

public class AutoUnboxing1

{

public static void main(String args[])

{

Integer obj = 15; //Upto 1.4

int x = obj.intValue();

System.out.println(x);

}

}

--------------------------------------------------------------

public class AutoUnboxing2

{

public static void main(String[] args)

{

Integer x = 25;

int y = x; //JDK 1.5 onwards

System.out.println(y);

}

}

---------------------------------------------------------------

public class AutoUnboxing3

{

public static void main(String[] args)

{

Integer i = 15;

System.out.println(i.byteValue());

System.out.println(i.shortValue());

System.out.println(i.intValue());

System.out.println(i.longValue());

System.out.println(i.floatValue());

System.out.println(i.doubleValue());

}

}

---------------------------------------------------------------

public class AutoUnboxing4

{

public static void main(String[] args)

{

Character c1 = 'A';

char ch = c1.charValue();

System.out.println(ch);

}

}

---------------------------------------------------------------

public class AutoUnboxing5

{

public static void main(String[] args)

{

Boolean b1 = true;

boolean b = b1.booleanValue();

System.out.println(b);

}

}

---------------------------------------------------------------

class BufferTest

{

public static void main(String[] args)

{

Integer i = 127;

Integer j = 127;

System.out.println(i==j); //true

System.out.println(i.equals(j)); //true

Integer a = 128;

Integer b = 128;

System.out.println(a==b); //false

System.out.println(a.equals(b)); ///true

Integer p = 130;

Integer q = 130;

System.out.println(p.equals(q)); //true

}

}

Note : Integer i = 127; This statement will accept the range of

byte(-128 to 127) after this range it will craete another object in another memory location.

While comparing the Wrapper object, It will always better

to compare to Wrapper object by using equals(Object obj).

Each Wrapper class has overidden equals(Object obj) and hashCode() method.

---------------------------------------------------------------

Unlike primitive types we can't convert one wrapper type object to another wrapper object.

Example :

Long l = 12; //Invalid

Float f = 90; //Invalid

Double d = 123; //Invalid

package com.ravi.basic;

public class Conversion

{

public static void main(String[] args)

{

long l = 12; //Implicit OR Widening

byte b = (byte) 12L; //Explicit OR Narrowing

Long a = 12L;

Double d = 90D;

Double d1 = 90.78;

Float f = 12F;

}

}

---------------------------------------------------------------

Ambiguity issue while overloading a method :

---------------------------------------------

When we overload a method then compiler is selecting appropriate method among the available methods based on the following types.

1. Different number of parameters

2. Different data type of parameters

3. Different sequence(order) of data type of parameters

In case of ambiguity where compiler can select more than one method then compiler will provide the priority by using following rules :

1) Most Specific Type :

-----------------------

Compiler alwyas provide more priority to most specific data type or class type.

double > float [Here float is the most specific type]

float > long

long > int

int > char

int > short //[No relation between short and char]

short > byte

2) WAV [Widening -> Autoboxing -> Var Args]

Compiler gives the priority to select appropriate method by using the following sequence :

Widening ---> Autoboxing ----> Var args

3) Nearest Data type or Nearest class (sub class)

While selecting the appropriate method in ambiguity issue compiler provides priority to nearest data type or nearest class i.e sub class

----------------------------------------------------------------

class Test

{

public void accept(double d)

{

System.out.println("double");

}

public void accept(float d)

{

System.out.println("float");

}

}

public class AmbiguityIssue {

public static void main(String[] args)

{

Test t = new Test();

t.accept(6);

}

}

Note : Here float will be executed becuase float is the most specific type.

---------------------------------------------------------------

class Test

{

public void accept(int d)

{

System.out.println("int");

}

public void accept(char d)

{

System.out.println("char");

}

}

public class AmbiguityIssue {

public static void main(String[] args)

{

Test t = new Test();

t.accept(6);

}

}

Here 6 is int type so int will be executed.

---------------------------------------------------------------

class Test

{

public void accept(int ...d)

{

System.out.println("int");

}

public void accept(char ...d)

{

System.out.println("char");

}

}

public class AmbiguityIssue {

public static void main(String[] args)

{

Test t = new Test();

t.accept();

}

}

char will be executed becoz char is more specific type.

---------------------------------------------------------------

class Test

{

public void accept(short ...d)

{

System.out.println("short");

}

public void accept(char ...d)

{

System.out.println("char");

}

}

public class AmbiguityIssue {

public static void main(String[] args)

{

Test t = new Test();

t.accept();

}

}

Here we will get compilation error because there is no relation between char and short based on the specific type rule.

--------------------------------------------------------------

class Test

{

public void accept(short ...d)

{

System.out.println("short");

}

public void accept(byte ...d)

{

System.out.println("byte");

}

}

public class AmbiguityIssue {

public static void main(String[] args)

{

Test t = new Test();

t.accept();

}

}

Here byte will be executed because byte is the specific type.

--------------------------------------------------------------

class Test

{

public void accept(double ...d)

{

System.out.println("double");

}

public void accept(long ...d)

{

System.out.println("long");

}

}

public class AmbiguityIssue {

public static void main(String[] args)

{

Test t = new Test();

t.accept();

}

}

Here long will be executed because long is the most specific type.

--------------------------------------------------------------

class Test

{

public void accept(byte d)

{

System.out.println("byte");

}

public void accept(short s)

{

System.out.println("short");

}

}

public class AmbiguityIssue {

public static void main(String[] args)

{

Test t = new Test();

//t.accept(18); //compilation error

t.accept((short)9);

t.accept((byte)9);

}

}

Here value 18 is of type int so, we can't assign directly to byte and short, If we want, explicit type casting is reqd.

---------------------------------------------------------------

class Test

{

public void accept(int d)

{

System.out.println("int");

}

public void accept(long s)

{

System.out.println("long");

}

}

public class AmbiguityIssue {

public static void main(String[] args)

{

Test t = new Test();

t.accept(9);

}

}

Note : Here int will be executed because int is the nearest type

--------------------------------------------------------------

class Test

{

public void accept(Object s)

{

System.out.println("Object");

}

public void accept(String s)

{

System.out.println("String");

}

}

public class AmbiguityIssue {

public static void main(String[] args)

{

Test t = new Test();

t.accept(9);

}

}

Here Object will be executed.

--------------------------------------------------------------

class Test

{

public void accept(Object s)

{

System.out.println("Object");

}

public void accept(String s)

{

System.out.println("String");

}

}

public class AmbiguityIssue {

public static void main(String[] args)

{

Test t = new Test();

t.accept("NIT");

}

}

Here String will be executed

--------------------------------------------------------------

class Test

{

public void accept(Object s)

{

System.out.println("Object");

}

public void accept(String s)

{

System.out.println("String");

}

public void accept(Integer i)

{

System.out.println("Integer");

}

}

public class AmbiguityIssue {

public static void main(String[] args)

{

Test t = new Test();

t.accept(null);

}

}

Here We will get compilation error

---------------------------------------------------------------

class Alpha

{

}

class Beta extends Alpha

{

}

class Test

{

public void accept(Alpha s)

{

System.out.println("Alpha");

}

public void accept(Beta i)

{

System.out.println("Beta");

}

}

public class AmbiguityIssue {

public static void main(String[] args)

{

Test t = new Test();

t.accept(null);

}

}

Here Beta will be executed.

--------------------------------------------------------------

class Test

{

public void accept(Number s)

{

System.out.println("Number");

}

public void accept(Integer i)

{

System.out.println("Integer");

}

}

public class AmbiguityIssue {

public static void main(String[] args)

{

Test t = new Test();

t.accept(12);

}

}

Here Integer will be executed.

--------------------------------------------------------------

class Test

{

public void accept(long s)

{

System.out.println("Widening");

}

public void accept(Integer i)

{

System.out.println("Autoboxing");

}

}

public class AmbiguityIssue {

public static void main(String[] args)

{

Test t = new Test();

t.accept(12);

}

}

Here widening is having more priority

--------------------------------------------------------------

class Test

{

public void accept(int ...s)

{

System.out.println("Var args");

}

public void accept(Integer i)

{

System.out.println("Autoboxing");

}

}

public class AmbiguityIssue {

public static void main(String[] args)

{

Test t = new Test();

t.accept(12);

}

}

Here Autoboxing will be executed.

--------------------------------------------------------------

class Test

{

public void accept(Number n)

{

System.out.println("Number");

}

public void accept(Double d)

{

System.out.println("Double");

}

}

public class AmbiguityIssue {

public static void main(String[] args)

{

Test t = new Test();

t.accept(12);

}

}

Here Number will be executed.

==============================================================

\*\*\*\*Method Overriding :

------------------------

Writing two or more non static methods in super and sub class in such a way that method name along with method parameter (Method Signature) must be same as well as return type must be compaitable is called Method Overriding.

Method Overriding is not possible without inheritance.

Generally we can't change the return type of the method while overriding a method (compatibility issue) but from JDK 1.5v there is a concept called Co-variant (In same direction) through which we can change the return type of the method.

Example :

---------

class Super

{

public void m1()

{

}

}

class Sub extends Super

{

public void m1() //Overridden Method

{

}

}

Method overriding is mainly used to replacing the implementation of super class method by sub class method body.

Advantage of Method Overriding :

---------------------------------

The advantage of Method Overriding is, each sub class is specifying its own specific behavior.

--------------------------------------------------------------

Upcasting :-

------------

It is possble to assign sub class object to super class reference variable (up) using dynamic polymorphism. It is known as Upcasting.

Example:- Animal a = new Lion(); //valid [upcasting]

Downcasting :

-------------

By default we can't assign super class object to sub class reference variable.

Lion l = new Animal(); //Invalid

Even if we type cast Animal to Lion type then compiler will allow but at runtime JVM will not convert Animal object (Generic type) into Lion object (Specific type) and it will throw an exception java.lang.ClassCastException

Lion l = (Lion) new Animal(); //At runtime we will get

java.lang.ClassCastException

Note : To avoid this ClassCastException we should use instanceof opertor.

Downcasting is a technique to assign sub class object (Only reference is super type) to sub class reference variable as shown below.

Animal a1 = new Lion();

Lion l = (Lion) a1; //Downcasting

Downcasting is not possible without upcasting.

---------------------------------------------------------------

Program on Method Overriding :

------------------------------

package com.ravi.scenario\_based\_mo;

class Payment

{

public void makePayment()

{

System.out.println("Generic Payment");

}

}

class Cash extends Payment

{

public void makePayment()

{

System.out.println("Payment through Cash");

}

}

class CreditCard extends Payment

{

public void makePayment()

{

System.out.println("Payment through Credit Card");

}

}

class DebitCard extends Payment

{

public void makePayment()

{

System.out.println("Payment through Debit Card");

}

}

public class MethodOverridingDemo1

{

public static void main(String[] args)

{

Payment p = null;

p = new Cash(); p.makePayment();

p = new DebitCard(); p.makePayment();

p = new CreditCard(); p.makePayment();

}

}

---------------------------------------------------------------

14-12-2024

-----------

package com.nit.overriding;

class Animal

{

public void sleep()

{

System.out.println("Generaic Aninmal is sleeping here");

}

}

class Dog extends Animal

{

public void sleep()

{

System.out.println("Dog Aninmal is sleeping here");

}

}

class Puppy extends Dog

{

public void sleep()

{

System.out.println("Dog Aninmal is sleeping here");

}

}

public class MethodOverridingDemo2 {

public static void main(String[] args)

{

Animal a1 = new Puppy();

a1.sleep();

}

}

Note : In the above program, compiler willsearch the method from

Animal class but JVM will start executing from Puppy class followd by Dog, Animal and Object.

------------------------------------------------------------------

@Override Annotation :

--------------------------

In Java we have a concept called Annotation, introduced from JDK 1.5 onwards. All the annotations must be start with @ symbol.

@Override annotation is metadata (Giving information that method is overridden) and it is optional but it is always a good practice to write @Override annotation before the Overridden method so compiler as well as user will get the confirmation that the method is overridden method and it is available in the super class.

If we use @Override annotation before the name of the overridden method in the sub class and if the method is not available in the super class then it will generate a compilation error so it is different from comments because comment will not generate any kind of compilation error if method is not an overridden method, so this is how it is different from comment.

package com.nit.overriding;

class Bird

{

public void fly()

{

System.out.println("Generic Bird is flying");

}

}

class Parrot extends Bird

{

@Override

public void fly()

{

System.out.println("Parrot Bird is flying");

}

}

class Peacock extends Bird

{

public void fly()

{

System.out.println("Peacock Bird is flying");

}

}

public class OverridingDemo3

{

public static void main(String[] args)

{

Bird b = null;

b = new Parrot(); b.fly(); //Dynamic Method Dispatch

b = new Peacock(); b.fly(); //Dynamic Method Dispatch

}

}

-------------------------------------------------------------------

package com.ravi.annotation;

class Vehicle

{

public int fuelCapacity()

{

return 40;

}

public void printTankCapacity()

{

System.out.println(this.fuelCapacity());

}

}

class Car extends Vehicle

{

@Override

public int fuelCapacity()

{

return 18;

}

public void printTankCapacity()

{

System.out.println(super.fuelCapacity());

}

}

public class IQ {

public static void main(String[] args)

{

Vehicle v1 = new Car();

v1.printTankCapacity();

}

}

--------------------------------------------------------------------

package com.nit.overriding;

class Bird

{

public void fly()

{

System.out.println("Genric Bird is flying");

}

public void roam()

{

System.out.println("Generic Bird is roamig");

}

}

class Parrot extends Bird

{

//Method Overloading

public int fly(double height)

{

System.out.println("Parrot is flying with :"+height+" height");

return 0;

}

@Override

public void roam()

{

System.out.println("Parrot Bird is roamig");

}

}

public class MethodOverridingDemo1 {

public static void main(String[] args)

{

Parrot p = new Parrot();

p.fly(15.6);

p.roam();

}

}

Note : From the above program It is clear that Method overloading is also possible in super and sub class.

-----------------------------------------------------------

Variable Hiding concept in upcasting :

---------------------------------------

class Super

{

int x = 100;

}

class Sub exetnds Super

{

int x = 200; //Variable Hiding

}

Only non static methods are overridden in java but not the variables[variables are not overridden in java] because behavior will change but not the property(variable).

Note : In upcasting, variable will be always executed besed on the current reference class variable.

Note : static variable, non static variable and static methods

are always executed using current reference.

package com.nit.overriding;

class RBI

{

protected String ifscCode = "RBIHYD0914567";

public String loan()

{

return "Provide loan";

}

}

class SBI extends RBI

{

protected String ifscCode = "SBIAMEERPET15679";

@Override

public String loan()

{

return "9.2% ROI";

}

}

public class VariableHidingDemo1

{

public static void main(String[] args)

{

RBI r = new SBI();

System.out.println(r.ifscCode+" : "+r.loan());

}

}

-----------------------------------------------------------

16-12-2024

-----------

Can we override private method ?

--------------------------------

No, We can't override private method of super class because private methods are not visible (not available) to the sub class hence we can't override.

We can't use @Override annotation on private method of sub class because it is not overridden method, actually it is re-declared by sub class developer.

class Super

{

private void m1()

{

System.out.println("M1 method in super class");

}

}

class Sub extends Super

{

public void m1() //Re-Declarartion

{

System.out.println("M1 method in Sub class");

}

}

public class MethodAccessibility

{

public static void main(String[] args)

{

new Sub().m1();

}

}

Note :- private method of super class is not available or not inherited in the sub class so if the sub class declare the method with same signature then it is not overridden method, actually it is re-declared in the sub class.

----------------------------------------------------------------

Role of access modifier while overriding a method :

---------------------------------------------------

While overriding the method from super class, the access modifier of sub class method must be greater or equal in comparison to access modifier of super class method otherwise we will get compilation error.

In terms of accessibility, public is greater than protected, protected is greater than default (public > protected > default)

[default < protected < public]

\*\*So the conclusion is we can't reduce the visibility of the method while overriding a method.

Note :- private method is not availble (visible) in sub class so it is not the part of method overriding.

----------------------------------------------------------

class Vehicle

{

public void run()

{

System.out.println("Generic Vehicle is running");

}

}

class Car extends Vehicle

{

@Override

protected void run() //error

{

System.out.println("Car Vehicle is running");

}

}

public class AccessibilityLevel

{

public static void main(String[] args)

{

Vehicle v = new Car();

v.run();

}

}

-----------------------------------------------------------

Co-variant in java :

--------------------

As we know while method overriding, the method signature must be same as well as return type must be compaitable,

If return type is not compaitable we will get compilation error as shown in the program.

class Shape

{

public void draw()

{

System.out.println("Generic Draw");

}

}

class Square extends Shape

{

@Override

public int draw()

{

System.out.println("Drawing Square");

return 0;

}

}

public class Incompaitable

{

public static void main(String[] args)

{

Shape s1 = new Square();

s1.draw();

}

}

Note : error, return type int is not compaitable with void.

-----------------------------------------------------------

-----------------------------------------------------------

But from JDK 1.5 onwards we can change the return type of the method in only one case that the return type of both the METHODS(SUPER AND SUB CLASS METHODS) MUST BE IN INHERITANCE RELATIONSHIP (IS-A relationship so it is compatible) called Co-Variant as shown in the program below.

Note :- Co-variant will not work with primitive data type, it will work only with classes.

Co-variant represents only one direction that meanse sub class method return type object we can assign super class method return object i.e in one direction.

package com.nit.co\_variant;

class Alpha{}

class Beta extends Alpha

{}

class Bird

{

public Alpha fly()

{

System.out.println("Generic Bird is flying");

return new Alpha();

}

}

class Parrot extends Bird

{

@Override

public Beta fly()

{

System.out.println("Parrot Bird is flying");

return new Beta();

}

}

public class CoVariantDemo

{

public static void main(String[] args)

{

Bird b = new Parrot();

b.fly();

}

}

-----------------------------------------------------------package com.nit.co\_variant;

class RBI

{

public Object loan()

{

System.out.println("Bank should provide loan");

return this;

}

}

class SBI extends RBI

{

public Double loan()

{

System.out.println("Providing loan @ 9.2% ROI");

return null;

}

}

public class CoVariantDemo2

{

public static void main(String[] args)

{

RBI r = new SBI();

r.loan();

}

}

-----------------------------------------------------------

17-12-2024

----------

Polymorphic behavior of Method while using Method Overriding :

-----------

package com.ravi.polymorphic\_behavior;

class Payment

{

public double makePayment(double amount)

{

System.out.println("Payment is in process...");

return 0.0;

}

}

class UPI extends Payment

{

@Override

public double makePayment(double amount)

{

System.out.println("Making a payment of "+amount+" using UPI");

return amount;

}

}

class CreditCard extends Payment

{

@Override

public double makePayment(double amount)

{

System.out.println("Making a payment of "+amount+" using Credit Card");

return amount;

}

}

public class PolymorphicBehavior

{

public static void main(String[] args)

{

Payment p1 = null;

p1 = new UPI();

acceptPayment(p1);

p1 = new CreditCard();

acceptPayment(p1);

}

public static void acceptPayment(Payment payment)

{

payment.makePayment(12000);

}

}

-----------------------------------------------------------

How to call specific method of a particular class :

---------------------------------------------------

package com.ravi.poly\_behavior;

class Payment

{

public double makePayment(double amount)

{

System.out.println("Payment is in process...");

return 0.0;

}

}

class UPI extends Payment

{

@Override

public double makePayment(double amount)

{

System.out.println("making a payment of "+amount+ " by using UPI");

return 0.0;

}

public void offer()

{

System.out.println("2 days Holiday in GOA");

}

}

class CreditCard extends Payment

{

@Override

public double makePayment(double amount)

{

System.out.println("making a payment of "+amount+ " by using CreditCard");

return 0.0;

}

public void offer()

{

System.out.println("Get 1000 RS cash back offer");

}

}

public class PolymorphicBehavior {

public static void main(String[] args)

{

Payment p = null;

p = new UPI();

acceptPayment(p);

}

public static void acceptPayment(Payment payment)

{

UPI upi = (UPI)payment; //Down-casting

upi.makePayment(12000);

upi.offer();

}

}

Note : By using UPI reference variable we can call UPI class specific method like offer, In order to call specific method of a class we need that corresponding class reference.

-----------------------------------------------------------

package com.ravi.poly\_behavior;

class Payment

{

public double makePayment(double amount)

{

System.out.println("Payment is in process...");

return 0.0;

}

}

class UPI extends Payment

{

@Override

public double makePayment(double amount)

{

System.out.println("making a payment of "+amount+ " by using UPI");

return 0.0;

}

public void offer()

{

System.out.println("2 days Holiday in GOA");

}

}

class CreditCard extends Payment

{

@Override

public double makePayment(double amount)

{

System.out.println("making a payment of "+amount+ " by using CreditCard");

return 0.0;

}

public void offer()

{

System.out.println("Get 1000 RS cash back offer");

}

}

public class PolymorphicBehavior {

public static void main(String[] args)

{

Payment p = null;

p = new UPI();

acceptPayment(p);

p = new CreditCard();

acceptPayment(p);

}

public static void acceptPayment(Payment payment)

{

UPI upi = (UPI)payment; //Down-casting

upi.makePayment(12000);

upi.offer();

}

}

Note : When we try to assign CreditCard object into Payment parameter variable, It is ok but we can't convert this CreditCard object into UPI type we will get Runtime

Exception java.lang.ClassCastException i.e. CreditCard can't be converted into UPI.

-----------------------------------------------------------

instanceof Operator :

---------------------

It is a relational operator which returns true/false.

It is also a keyword.

It is used to verify whether a reference variable is pointing to a particular type of Object or not?

It is used to resolve the issue of ClassCastException.

While working with instanceof operator we must have IS-A

relation in between reference variable and class/interface

type otherwise we will get compilation error.

-----------------------------------------------------------

Program on instanceof operator :

---------------------------------

package com.ravi.instance\_of\_demo;

class Alpha

{

}

class Beta extends Alpha

{

}

class Gamma extends Beta

{

}

public class InstanceOfDemo1

{

public static void main(String[] args)

{

Gamma g = new Gamma();

if(g instanceof Gamma)

{

System.out.println("g is pointing to Gamma Object");

}

if(g instanceof Beta)

{

System.out.println("g is pointing to Beta Object");

}

if(g instanceof Alpha)

{

System.out.println("g is pointing to Alpha Object");

}

if(g instanceof Object)

{

System.out.println("g is pointing to Object Object");

}

}

}

-----------------------------------------------------------

package com.ravi.instance\_of\_demo;

class Bird

{

public void fly()

{

System.out.println("Can fly");

}

}

class Fish

{

public void swim()

{

System.out.println("Can swim");

}

}

public class InstanceOf {

public static void main(String[] args)

{

Fish f = new Fish();

if(f instanceof Bird) //Compilation error

{

}

}

}

Here we will get compilation error becoz there is no relation between Bird and Fish

-----------------------------------------------------------

package com.ravi.instance\_of\_demo;

public class InstanceOfDemo3

{

public static void main(String[] args)

{

String str = "India";

if(str instanceof String)

{

System.out.println("str is pointing to String object");

}

}

}

-----------------------------------------------------------

package com.ravi.instance\_of\_demo;

class Vehicle

{

public void run()

{

System.out.println("Vehicle is running");

}

}

class BMW extends Vehicle

{

}

class Audi extends Vehicle

{

}

public class InstanceOfDemo4 {

public static void main(String[] args)

{

//BMW b = new BMW();

//acceptCarType(b);

Audi a = new Audi();

acceptCarType(a);

}

public static void acceptCarType(Vehicle v)

{

if(v instanceof BMW)

{

System.out.println("BMW Car");

}

else

{

System.out.println("AUDI Car");

}

}

}

==========================================================

package com.ravi.poly\_behavior;

class Payment

{

public double makePayment(double amount)

{

System.out.println("Payment is in process...");

return 0.0;

}

}

class UPI extends Payment

{

@Override

public double makePayment(double amount)

{

System.out.println("making a payment of "+amount+ " by using UPI");

return 0.0;

}

public void offer()

{

System.out.println("2 days Holiday in GOA");

}

}

class CreditCard extends Payment

{

@Override

public double makePayment(double amount)

{

System.out.println("making a payment of "+amount+ " by using CreditCard");

return 0.0;

}

public void offer()

{

System.out.println("Get 1000 RS cash back offer");

}

}

class DebitCard extends Payment

{

@Override

public double makePayment(double amount)

{

System.out.println("making a payment of "+amount+ " by using DebitCard");

return 0.0;

}

}

public class PolymorphicBehavior {

public static void main(String[] args)

{

UPI u = new UPI();

CreditCard cc = new CreditCard();

DebitCard dc = new DebitCard();

acceptPayment(u,cc,dc);

}

public static void acceptPayment(Payment ...payment)

{

for(Payment p : payment)

{

if(p instanceof UPI)

{

UPI upi = (UPI)p;

upi.makePayment(15000);

upi.offer();

}

else if(p instanceof CreditCard)

{

CreditCard cc = (CreditCard) p;

cc.makePayment(22000);

cc.offer();

}

else if(p instanceof DebitCard)

{

DebitCard d = (DebitCard) p;

d.makePayment(12000);

}

else if(p instanceof Cash)

{

DebitCard d = (DebitCard) p;

d.makePayment(12000);

}

}

}

}

===========================================================

final keyword in java :

-----------------------

It is used to provide some kind of restriction in our program.

We can use final keyword in ways 3 ways in java.

1) To declare a class as a final. (Inheritance is not possible)

2) To declare a method as a final (Overriding is not possible)

3) To declare a variable (Field) as a final (Re-assignment is not possible)

18-12-2024

----------

1) To declare a class as a final :

-----------------------------------

Whenever we declare a class as a final class then we cann't extend or inherit that class otherwise we will get a compilation error.

We should declare a class as a final if the composition of the class (logic of the class) is very important and we don't want to share the feature of the class to some other developer to modify the original behavior of the existing class, In that situation we should declare a class as a final.

Declaring a class as a final does not mean that the variables and methods declared inside the class will also become as a final, only the class behavior is final that means we can modify the variables value as well as we can create the object for the final classes.

Note :- In java String and All wrapper classes are declared as final class.

--------------------------------------------------------------

final class A

{

private int x = 100;

public void setData()

{

x = 120;

System.out.println(x);

}

}

class B extends A

{

}

public class FinalClassEx

{

public static void main(String[] args)

{

B b1 = new B();

b1.setData();

}

}

Note : class A is final so we can't inherit hence we will get compilation error.

---------------------------------------------------------------

final class Test

{

private int data = 100;

public Test(int data)

{

this.data = data;

System.out.println("Data value is :"+data);

}

}

public class FinalClassEx1

{

public static void main(String[] args)

{

Test t1 = new Test(200);

}

}

Note : for final class we can create object as well as we can modify the data.

--------------------------------------------------------------

Whenever we declare a constructor as private then we should declare the class with final modifier. If constructor is private then we can't create a sub class because super class constructor is not visible from sub class constructor.

final class Sample

{

private Sample() //Constructor is private so declare

the class with final modifier

{

System.out.println("Private Constructor");

}

}

public class FinalClassEx2

{

public static void main(String[] args)

{

}

}

---------------------------------------------------------------

Sealed class in Java :

-----------------------

It is a new feature introduced from java 15v (preview version) and become the integral part of java from 17v.

It is an improvement over final keyword.

By using sealed keyword we can declare classes and interfaces as sealed.

It is one kind of restriction that describes which classes and interfaces can extend or implement from Sealed class Or interface.

It is similar to final keyword with less restriction because here we can permit the classes to extend from the original Sealed class.

The class which is inheriting from the sealed class must be final, sealed or non-sealed.

The sealed class must have atleast one sub class.

We can also create object for Sealed class.

It provides the following modifiers :

1) sealed : Can be extended only through permitted class.

2) non-sealed : Can be extended by any sub class, if a user wants to give permission to its sub classes.

3) permits : We can provide permission to the sub classes, which are inheriting through Sealed class OR sealed interface

4) final : we can declare permitted sub class as final so, it cannot be extended further.

package com.ravi.sealed;

sealed class Bird permits Parrot, Sparrow

{

public void roam()

{

System.out.println("Generic Bird is roaming");

}

}

non-sealed class Parrot extends Bird

{

@Override

public void roam()

{

System.out.println("Parrot Bird is roaming");

}

}

final class Sparrow extends Bird

{

@Override

public void roam()

{

System.out.println("Sparrow Bird is roaming");

}

}

public class SealedDemo1 {

public static void main(String[] args)

{

Bird b = null;

b = new Parrot(); b.roam(); //Dynamic Method Dispatch

b = new Sparrow(); b.roam(); //Dynamic Method Dispatch

}

}

-------------------------------------------------------------

package com.nit.sealed;

sealed class OnlineClass permits Laptop, Mobile

{

public void attendJavaOnline()

{

System.out.println("Sunday online class at 9: 30AM");

}

}

final class Laptop extends OnlineClass

{

@Override

public void attendJavaOnline()

{

System.out.println("Attending online class through Laptop");

}

}

final class Mobile extends OnlineClass

{

@Override

public void attendJavaOnline()

{

System.out.println("Attending online class through Mobile");

}

}

public class SealedDemo2 {

public static void main(String[] args)

{

OnlineClass onlineClass = null;

onlineClass = new Laptop(); onlineClass.attendJavaOnline();

onlineClass = new Mobile(); onlineClass.attendJavaOnline();

}

}

--------------------------------------------------------------

2) To declare a method as a final (Overriding is not possible)

---------------------------------------------------------------

Whenever we declare a method as a final then we can't override that method in the sub class otherwise there will be a compilation error.

We should declare a method as a final if the body of the method i.e the implementation of the method is very important and we don't want to override or change the super class method body by sub class method body then we should declare the super class method as final method.

---------------------------------------------------------------

class A

{

protected int a = 10;

protected int b = 20;

public final void calculate()

{

int sum = a+b;

System.out.println("Sum is :"+sum);

}

}

class B extends A

{

@Override

public void calculate() //error

{

int mul = a\*b;

System.out.println("Mul is :"+mul);

}

}

public class FinalMethodEx

{

public static void main(String [] args)

{

A a1 = new B();

a1.calculate();

}

}

---------------------------------------------------------------

class Alpha

{

private final void accept()

{

System.out.println("Alpha class accept method");

}

}

class Beta extends Alpha

{

public void accept()

{

System.out.println("Beta class accept method");

}

}

public class FinalMethodEx1

{

public static void main(String [] args)

{

new Beta().accept();

}

}

Note : Here Program will compile and execute because private method of super class is not available to sub class.

-------------------------------------------------------------

3) To declare a variable/Field as a final :

--------------------------------------------

In older langugaes like C and C++ we use "const" keyword to declare a constant variable but in java, const is a reserved word for future use so instead of const we should use "final" keyword.

If we declare a variable as a final then we can't perform re-assignment (i.e nothing but re-initialization) of that variable.

In java It is always a better practise to declare a final variable by uppercase letter according to the naming convention.

---------------------------------------------------------------

class A

{

final int A = 10; //re-assignment is not possible

public void setData()

{

A = 10; //error

System.out.println("A value is :"+A);

}

}

class FinalVarEx

{

public static void main(String[] args)

{

final A a1 = new A();

a1.setData();

a1 = new A(); //error

a1.setData();

}

}

---------------------------------------------------------------

19-12-2024

-----------

What is Method Hiding in java ?

OR

Can we override static Method ?

OR

Can we override main Method ?

In order to work with Method Hiding we have different Cases :

Case 1 :

---------

Any public static method of super class is by default available

to sub class so from sub class we can call super class static method as shown in the program below :

class Super

{

public static void m1()

{

System.out.println("m1 static method of super class");

}

}

class Sub extends Super

{

}

public class MethodHiding

{

public static void main(String[] args)

{

Super.m1();

Sub.m1();

Sub s1 = new Sub();

s1.m1();

}

}

---------------------------------------------------------------

Case 2 :

-------

We can't override a static method with non static method because static method belongs to class and non static method belongs to object, If we try to override static method with non static method then it will generate an error i.e overridden method is static as shown below.

class Super

{

public static void m1()

{

System.out.println("m1 static method of super class");

}

}

class Sub extends Super

{

public void m1() //error

{

System.out.println("m1 non static method of Sub class");

}

}

public class MethodHiding1

{

public static void main(String[] args)

{

}

}

---------------------------------------------------------------

Case 3 :

--------

We can't override any non static method with static method, If we try then it will generate an error, Overriding method is static.

class Super

{

public void m1()

{

System.out.println("m1 non static method of super class");

}

}

class Sub extends Super

{

public static void m1()

{

System.out.println("m1 static method of Sub class");

}

}

public class MethodHiding1

{

public static void main(String[] args)

{

}

}

So, the conclusion is we cannot overide static with non static method as well as non-static with static method because static method belongs to class and non-static method belongs to object.

---------------------------------------------------------------

Can we show that the following program is Method Hiding but not Overriding :

class Super

{

public static void m1()

{

System.out.println("m1 static method of super class");

}

}

class Sub extends Super

{

public static int m1()

{

System.out.println("m1 static method of Sub class");

return 0;

}

}

public class MethodHiding

{

public static void main(String[] args)

{

}

}

From the above program it is clear that :

Method Hiding belons to static Method

Method Overriding belons to non-static Method

Case 4 :

---------

We can't override static method because It belong to class but not object, If we write static method in the sub class with same signature and compaitable return type then It is Method Hiding but not Method Overriding here compiler will search the method of super class and JVM will also execute the method of super class because method is not overridden.[Single copy and belongs to class area, sharable by all the objects]

Note :- 1) We can't apply @Override annotation on static methods.

2) Static methods can't be overridden so behavior is

same for all the Objects hence it is Static Polymorphism.

package com.ravi.method\_hiding;

class Animal

{

public static void checkup()

{

System.out.println("Check Up rule for all the Animals :");

}

}

class Dog extends Animal

{

public static void checkup()

{

System.out.println("Check Up rule for Dog Animal");

}

}

class Lion extends Animal

{

public static void checkup()

{

System.out.println("Check Up rule for Lion Animal");

}

}

public class OverriidngStaicMethod {

public static void main(String[] args)

{

Animal a = null;

a = new Dog(); a.checkup();

a = new Lion(); a.checkup();

}

}

static method always executed by using current reference

---------------------------------------------------------------

Abstraction [Hiding the complexcity]

------------------------------------

Showing the essential details without showing the background details is called Abstraction.

In order to acheive abstraction we can use the following two concepts of Java :

1) Abstract class and abstract Method (It provides 0 - 100% abstraction so Partial Abstraction)

2) Interface (100% abstraction)

-------------------------------------------------------------

20-12-2024

------------

Abstract class and abstract methods :

-------------------------------------------

A class that does not provide complete implementation (partial implementation) is defined as an abstract class.

An abstract method is a common method which is used to provide easiness to the programmer because the programmer faces complexcity to remember the method name.

An abstract method observation is very simple because every abstract method contains abstract keyword, abstract method does not contain any method body and at the end there must be a terminator i.e ; (semicolon)

In java, whenever action is common but implementations are different then we should use abstract method, Generally we declare abstract method in the super class and its implementation must be provided in the sub classes.

if a class contains at least one method as an abstract method then we should compulsory declare that class as an abstract class.

Once a class is declared as an abstract class we can't create an object for that class.

\*All the abstract methods declared in the super class must be overridden in the sub classes otherwise the sub class will become as an abstract class hence object can't be created for the sub class as well.

In an abstract class we can write all abstract method or all concrete method or combination of both the method.

It is used to acheive partial abstraction that means by using abstract classes we can acheive partial abstraction(0-100%).

\*An abstract class may or may not have abstract method but an abstract method must have abstract class.

Note :- We can't declare an abstract method as final, private and static (illegal combination of modifiers)

We can't declare an abstract class as a final.

-------------------------------------------------------------

abstract class Shape

{

public abstract void draw();

}

class Square extends Shape

{

@Override

public void draw()

{

System.out.println("Drawing Square");

}

}

class Rectangle extends Shape

{

@Override

public void draw()

{

System.out.println("Drawing Rectangle");

}

}

public class AbstractDemo1

{

public static void main(String[] args)

{

Shape s = new Square();

s.draw();

Shape s1 = new Rectangle();

s1.draw();

}

}

-------------------------------------------------------------

package com.ravi.asbatrct\_demo;

abstract class Vehicle

{

protected int speed = 120;

public Vehicle()

{

System.out.println("Vehicle class Constructor");

}

public void getVehicleDetails()

{

System.out.println("Vehicle has 4 wheels...");

}

public abstract void run();

}

class Car extends Vehicle

{

@Override

public void run()

{

System.out.println("Car is Running");

}

}

public class IQ

{

public static void main(String[] args)

{

Vehicle v = new Car();

System.out.println("Speed is :"+v.speed);

v.getVehicleDetails();

v.run();

}

}

Note : abstracr class constructor will be executed through

sub class object by using super()

-----------------------------------------------------------

IQ :

-----

What is the advantage of writing instance variable(Object properties) and constructor inside an abstract class, If we can't create an object for abstract class ?

Yes, We can't create an object for abstract class but still we can write instance variable and constructor that means

abstract class also contains properties and behavior but these properties will initialize through sub class object using super() so we can write object properties as well as

constructor.

-------------------------------------------------------------

WAP to initailize super abstract class properties by using sub class object.

package com.ravi.asbatrct\_demo;

abstract class Shape

{

protected String shapeType;

public Shape(String shapeType)

{

super();

this.shapeType = shapeType;

}

public abstract void draw();

}

class Square extends Shape

{

public Square(String shape)

{

super(shape);

}

@Override

public void draw()

{

System.out.println("Drawing :"+shapeType);

}

}

class Rectangle extends Shape

{

public Rectangle(String shape)

{

super(shape);

}

@Override

public void draw()

{

System.out.println("Darwing :"+shapeType);

}

}

public class AbstractDemo1

{

public static void main(String[] args)

{

Square ss = new Square("Square");

ss.draw();

Rectangle rr = new Rectangle("Rectangle");

rr.draw();

}

}

-------------------------------------------------------------

WAP that describes overriding of abstract method is compulsory in the sub classes otherwise sub class will also

become as an abstract class.

package com.ravi.asbatrct\_demo;

abstract class Alpha

{

public abstract void show();

public abstract void demo();

}

abstract class Beta extends Alpha

{

@Override

public void show() //demo();

{

System.out.println("Show Method Overridden in Beta class");

}

}

class Gamma extends Beta

{

@Override

public void demo()

{

System.out.println("Demo Method Overridden in Gamma class");

}

}

public class AbstractDemo2

{

public static void main(String[] args)

{

Gamma g = new Gamma();

g.show();

g.demo();

}

}

------------------------------------------------------------

WAP to show that abstract method is working as Business Rule

so all the sub classes will follow this rule :

package com.ravi.asbatrct\_demo;

abstract class Animal

{

protected String name;

public Animal(String name)

{

super();

this.name = name;

}

public abstract void checkup();

}

class Lion extends Animal

{

public Lion(String name)

{

super(name);

}

@Override

public void checkup()

{

System.out.println(name+" is going for Check up");

}

}

class Elephant extends Animal

{

public Elephant(String name)

{

super(name);

}

@Override

public void checkup()

{

System.out.println(name+" is going for Check up");

}

}

class Dog extends Animal

{

public Dog(String name)

{

super(name);

}

@Override

public void checkup()

{

System.out.println(name+" is going for Check up");

}

}

public class AbstractDemo3

{

public static void main(String[] args)

{

Lion lions[] = {new Lion("Simba"), new Lion("Scar"),new Lion("Alex")};

Elephant [] elephants = {new Elephant("Airavata"), new Elephant("My ele")};

Dog [] dogs = {new Dog("Tommy"), new Dog("Tiger"), new Dog("My Dog")};

checkAnimal(lions);

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

checkAnimal(elephants);

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

checkAnimal(dogs);

}

public static void checkAnimal(Animal ...animals)

{

for(Animal animal : animals)

{

animal.checkup();

}

}

}

------------------------------------------------------------

21-12-2024

-----------

Anonymous inner class concept :

--------------------------------

If we declare a class inside a method without any name and with ; (terminator) then it is called Anonymous inner class.

All the .class files are represented by $ symbol.An anonymous inner class .class files are represented by numbers like $1, $2 and so on

The main purpose of Anonymous inner class to extend a class OR to implement an interface that means to create sub type.

Anonymous inner class, class declaration and object creation by using new keyword is possible in the same line at the time of declartion the anonymous inner class body.

-------------------------------------------------------------

//Program to implement Anonymous inner class by using Concrete class

AnonymousDemo1.java

---------------------

package com.ravi.anonymous;

class Super

{

public void show()

{

System.out.println("Super class Show Method.");

}

}

public class AnonymousDemo1 {

public static void main(String[] args)

{

//Anonymous inner class(Without any Name)

Super sub = new Super()

{

@Override

public void show()

{

System.out.println("Sub class Show Method.");

}

};

sub.show();

}

}

//Program to implement Anonymous inner class by using abstract class

package com.ravi.anonymous;

abstract class Vehicle

{

public abstract void run();

}

public class AnonymousDemo2 {

public static void main(String[] args)

{

Vehicle car = new Vehicle()

{

@Override

public void run()

{

System.out.println("car is Running");

}

};

Vehicle bike = new Vehicle()

{

@Override

public void run()

{

System.out.println("Bike is running");

}

};

car.run(); bike.run();

}

}

============================================================

interface :

-----------

interface upto java 1.7

------------------------

An interface is a keyword in java which is similar to a class which defines working functionality of a class.

Upto JDK 1.7 an interface contains only abstract methods that means there is a guarantee that inside an interfcae we don't have concrete or general or instance methods.

From java 8 onwards we have a facility to write default and static methods.

By using interface we can achieve 100% abstraction concept because it contains only abstract methods.

In order to implement the member of an interface, java software people has provided implements keyword.

All the methods declared inside an interface is by default public and abstract so at the time of overriding we can't reduce the accessibility level.

All the variables declared inside an interface is by default public, static and final.

We should override all the abstract methods of interface to the sub classes otherwise the sub class will become as an abstract class hence object can't be created.

We can't create an object for interface, but reference can be created.

By using interface we can acheive multiple inheritance in java.

We can achieve loose coupling using interface.

Note :- inside an interface we can't declare any blocks (instance, static), instance variables (No properties) as well as we can't write constructor inside an interface.

===========================================================

23-12-2024

-----------

//Programs using interface concept :

-----------------------------------

package com.ravi.interface\_demo;

sealed interface Moveable permits Car

{

int SPEED = 100; //public + static + final

void move(); //public + abstract

}

final class Car implements Moveable

{

@Override

public void move()

{

//SPEED = 120; //Invalid

System.out.println("Car speed is :"+SPEED);

}

}

public class InterfaceDemo1

{

public static void main(String[] args)

{

Moveable m = new Car();

m.move();

System.out.println("Speed of the Car is :"+Moveable.SPEED);

}

}

------------------------------------------------------------

package com.ravi.interface\_demo;

interface Bank

{

void deposit(double amount);

void withdraw(double amount);

}

class Customer implements Bank

{

protected double balance;

public Customer(double balance)

{

super();

this.balance = balance;

}

@Override

public void deposit(double amount)

{

if(amount <=0)

{

System.err.println("Amount can't be deposited");

System.exit(0);

}

else

{

this.balance = this.balance + amount;

System.out.println("Balance after deposit :"+this.balance);

}

}

@Override

public void withdraw(double amount)

{

if(amount > this.balance)

{

System.err.println("Insufficient Balance!!");

System.exit(0);

}

else

{

this.balance = this.balance - amount;

System.out.println("Amount after Witdraw is :"+this.balance);

}

}

public double getCurrentBalance()

{

return this.balance;

}

}

public class InterfaceDemo2 {

public static void main(String[] args)

{

Customer scott = new Customer(1000);

System.out.println("Current Balance is :"+scott.getCurrentBalance());

scott.deposit(1000);

scott.withdraw(500);

}

}

-------------------------------------------------------------

Program on loose coupling : (Industry Standard Program)

--------------------------------------------------------

Loose Coupling :- If the degree of dependency from one class object to another class is very low then it is called loose coupling. [interface is reqd]

Tightly coupled :- If the degree of dependency of one class to another class is very high then it is called Tightly coupled.

According to IT industry standard we should always prefer loose coupling so the maintenance of the project will become easy.

High Cohesion [Encapsulation]:

------------------------------

Our private data must be accessible via public methods (setter and getters) so, in between data and method we must have high cohesion.

(tight coupling) so, validation of outer data is possible.

6 files :

----------

HotDrink.java(I)

package com.ravi.loose\_coupling;

public interface HotDrink

{

void prepare();

}

Tea.java

--------

package com.ravi.loose\_coupling;

public class Tea implements HotDrink

{

@Override

public void prepare()

{

System.out.println("Preparing Tea");

}

}

Coffee.java

-------------

package com.ravi.loose\_coupling;

public class Coffee implements HotDrink

{

@Override

public void prepare()

{

System.out.println("Preparing Coffee");

}

}

Horlicks.java

--------------

package com.ravi.loose\_coupling;

public class Horlicks implements HotDrink {

@Override

public void prepare()

{

System.out.println("Preparing Horlicks");

}

}

Restaurant.java

----------------

package com.ravi.loose\_coupling;

public class Restaurant

{

public static void acceptObject(HotDrink hd) //hd = new Tea();

{

hd.prepare();

}

}

LooseCoupling.java

--------------------

package com.ravi.loose\_coupling;

public class LooseCoupling

{

public static void main(String[] args)

{

Restaurant.acceptObject(new Tea());

Restaurant.acceptObject(new Coffee());

Restaurant.acceptObject(new Horlicks());

}

}

Method retutn type as a interface :

-----------------------------------

It is always better to take method return type as interface so we can return any implementer class object as shown in the example below

public HotDrink accept()

{

return new Tea() OR new Coffee() OR new Horlicks() OR any future

implementer class object...........................

}

-------------------------------------------------------------

Compile time constant :

-----------------------

A compile time constant is a constant that is evaluated and replaced with its value at compile time rather than runtime.

It must be declared with static and final modifier as well as initialized with constant expression. (Must not be initialized by method call)

At compile time constant value will be converted by compiler at the time of compilation itself so, at runtime JVM can see the value but not the class name so class will not be loaded as shown in the program.

Example : public static final int A = 100; //Valid

public static final int A = m1(); //valid [Here

class will be loaded by JVM]

class Foo

{

public static final int A = 100; //compile time Constant

static

{

System.out.println("Static Block");

}

}

public class CompileTimeConstant

{

public static void main(String[] args)

{

System.out.println(Foo.A);

}

}

Note : SB will not be executed.

-------------------------------------------------------------

class Foo

{

public static final int A = returnValue();

static

{

System.out.println("Static Block");

}

public static int returnValue()

{

return 500;

}

}

public class CompileTimeConstant1

{

public static void main(String[] args)

{

System.out.println(Foo.A);

}

}

Note : SB will be executed

------------------------------------------------------------

2 files :

Test.java

---------

public class Test

{

public static final int MAX\_VALUE = 999;

}

Main.java

---------

public class Main

{

public static void main(String[] args)

{

System.out.println(Test.MAX\_VALUE);

}

}

Note : If we compile and execute both programs we will get 999 value, now change the 999 to 1500, Re- compile only Test.java so a new .class file will be acreated which will hold 1500 value. Now without re-compilation of main.java if we execute Main.java then we wil get 999 so it takes the value at compile time.

-----------------------------------------------------------

24-12-2024

-----------

Multiple Inheritance by using interface :

-----------------------------------------

In a class we have a constructor so, it is providing ambiguity issue but inside an interface we don't have constructor so multiple inheritance is possible using interface as shown in the program below.

The Implementer class constructor's super keyword will directly move to Object class constructor.[24-DEC]

package com.ravi.interface\_demo;

interface A

{

void m1();

}

interface B

{

void m1();

}

class Implementer implements A,B

{

@Override

public void m1()

{

System.out.println("Multiple Inheritance is Possible");

}

}

public class MultipleInheritance {

public static void main(String[] args)

{

new Implementer().m1();

}

}

-----------------------------------------------------------------

Extending an interface :

-------------------------

One interface can extend another interface but it can't implement an interface as shown in the program.

package com.ravi.interface\_demo;

abstract interface Alpha

{

void m1();

}

abstract interface Beta extends Alpha

{

void m2();

}

class Sub implements Beta

{

@Override

public void m1()

{

System.out.println("M1 method implemented");

}

@Override

public void m2()

{

System.out.println("M2 method implemented");

}

}

public class ExteningInterface {

public static void main(String[] args)

{

Sub s1 = new Sub();

s1.m1(); s1.m2();

}

}

------------------------------------------------------------------

java 8 fetaures :

-----------------

Interface from JDK 1.8V

------------------------

What is the limitation of abstract method in project maintenance :

-------------------------------------------------------------

Java 8 fetaures are introduced in java from March 2014.

intreface from JDK 1.8V [Java 8 = March 2014]

----------------------------------------------

Limitation of abstract method

OR

Maintenance problem with interface in an Industry upto JDK 1.7

The major maintenance problem with interface is, if we add any new abstract method at the later stage of development inside an existing interface then all the implementer classes have to override that abstract method otherwise the implementer class will become as an abstract class so it is one kind of boundation.

We need to provide implementation for all the abstract methods available inside an interface whether it is required or not?

To avoid this maintenance problem java software people introduced default method inside an interface.

-------------------------------------------------------------

What is default method :

-------------------------

We can write default method (method with body) inside an interface with default keyword from Java 8v.

This default method provides "default implementation" so the implementer class can override to provide specific implementation in the class.

Unlike abstract method, default method does not provide any kind of boundation to override this default method in the sub class.

By default the access modifier of default method is public.

We can't write default method inside a class, we can write

only inside an interface.

------------------------------------------------------------------

4 files :

---------

Vehicle.java(I)

-------------

package com.ravi.java8;

public interface Vehicle

{

void run();

void horn();

default void digitalMeter() //java 8

{

System.out.println("Digital Meter Facility is coming soon!!!");

}

}

Car.java

----------

package com.ravi.java8;

public class Car implements Vehicle

{

@Override

public void run()

{

System.out.println("Car is Running");

}

@Override

public void horn()

{

System.out.println("Car has Horn");

}

@Override

public void digitalMeter() //java 8

{

System.out.println("Car has Digital Meter Facility");

}

}

Bike.java

---------

package com.ravi.java8;

public class Bike implements Vehicle

{

@Override

public void run()

{

System.out.println("Bike is Running");

}

@Override

public void horn()

{

System.out.println("Bike has Horn");

}

}

DefaultMethod.java

------------------

package com.ravi.java8;

public class DefaultMethod

{

public static void main(String[] args)

{

Vehicle v = null;

v = new Car(); v.run(); v.horn(); v.digitalMeter();

v = new Bike(); v.run(); v.horn();v.digitalMeter();

}

}

-------------------------------------------------------------

What is static method of an interface :

----------------------------------------

We can write static method inside an interface from java 8 version.

Example :

---------

public interface Calculate

{

static double getCube(int num) //java 8 [AM is public]

{

}

}

by default this static method access modifier is public.

static method we can write inside an interafce to provide common functionality.

A static method of interface is not available to the implementer classes so static method of an interface can

be accessible through interface only.

-------------------------------------------------------------

Program that describes static method of an interface is by default public so we can access from another package.

Calculate.java(com.ravi.interface\_static\_method)

package com.ravi.interface\_static\_method;

public interface Calculate

{

static double getCube(int num)

{

return (num\*num\*num);

}

static double getSquare(int num)

{

return (num\*num);

}

}

package com.ravi.priority;

import com.ravi.interface\_static\_method.Calculate;

public class ELC //This class is in another package.

{

public static void main(String[] args)

{

double result = Calculate.getCube(8);

System.out.println("Cube of 8 is :"+result);

result = Calculate.getSquare(12);

System.out.println("Square of 12 is :"+result);

}

}

-------------------------------------------------------------

From java 8 version It is also possible to write main method

inside an interface and it will be executed.

package com.ravi.interface\_static\_method;

public interface Callable

{

public static void main(String[] args)

{

System.out.println("Main method inside interface");

}

}

-------------------------------------------------------------

Program that describes interface static methods are available to interafce only.

package com.ravi.interface\_static\_method;

interface Printable

{

static void m1()

{

System.out.println("M1 static Method");

}

}

public class ScopeOfStaticMethod implements Printable

{

public static void main(String[] args)

{

Printable.m1();

//ScopeOfStaticMethod.m1();

//new ScopeOfStaticMethod().m1();

}

}

------------------------------------------------------------

Interface Static Method:

------------------------

a) Accessible using the interface name.

b) Cannot be hidden by implementing classes.(Not Available)

c) Can be called using the interface name only.

Class Static Method:

--------------------

a) Accessible using the class name.

b) Can be hidden (not overridden) in subclasses by redeclaring a static method with the same signature and compaitable return type.

c) Can be called using the super class, sub class name as well as sub class object.

-------------------------------------------------------------

Priority of deafult and concrete method :

-----------------------------------------

While working with class and interface, default method is having low

priority than concrete method, In the same way class is more powerfult than interface.

interface A

{

}

class B

{

}

class C extends B implements A {} //Valid

class C implements A extends B {} //Invalid

-------------------------------------------------------------

package com.ravi.default\_method;

interface A

{

default void m1()

{

System.out.println("default method of interface A");

}

}

class B

{

public void m1()

{

System.out.println("Concrete method of class B");

}

}

class C extends B implements A

{

}

public class Priority

{

public static void main(String[] args)

{

C c1 = new C();

c1.m1();

}

}

-------------------------------------------------------------

Can we achieve multiple inheritance using default method :

----------------------------------------------------------

Multiple inheritance is possible in java by using default method inside an interface, here we need to use super keyword to differenciate the super interface methods.

Before java 1.8, we have abstract method inside an interface but now we can write method body(default method) so, to execute the default method inside an interface we need to take super keyword with interface name(Alpha.super.m1()).

package com.ravi.default\_method;

interface Alpha

{

default void m1()

{

System.out.println("m1 method of Alpha interface ");

}

}

interface Beta

{

default void m1()

{

System.out.println("m1 method of Beta interface ");

}

}

class Foo implements Alpha, Beta

{

@Override

public void m1()

{

Alpha.super.m1();

Beta.super.m1();

System.out.println("MI is possible using default method");

}

}

public class MultipleInheritance {

public static void main(String[] args)

{

new Foo().m1();

}

}

-------------------------------------------------------------

package com.ravi.default\_method;

interface Hello

{

int x = 100;

}

abstract class Test

{

int x = 200;

}

class MyClass extends Test implements Hello

{

int x = 300; //Variable Hiding to solve ambiguity issue

public void show()

{

System.out.println(x);

}

}

public class VariableHiding {

public static void main(String[] args)

{

new MyClass().show();

}

}

-------------------------------------------------------------

Introduction to Functional Programming ?

----------------------------------------

From JDK 8 onwards, Java also concentrated on function/method and introduced Functional Programming.

It is mainly used to write concise coding so the length of the method will be reduced.

What is a Functional interface in java ?

----------------------------------------

If an interface contains exactly one abstract method then

that interface is known as Functional interafce.

Example :

----------

public interface Drawable

{

void draw(); //SAM [Single abstract method]

}

It may contain 'n' number of default and static methods but It must contain only one abstract method.

In order to restrict developer to take more than one abstract method, Java software people has introduced @FunctionalInterface annotation.

Example :

---------

@FunctionalInterface

interface Vehicle

{

void run();

}

-------------------------------------------------------------

package com.ravi.default\_method;

@FunctionalInterface

interface Vehicle

{

void run();

}

public class MultipleInheritance

{

public static void main(String[] args)

{

Vehicle car = new Vehicle()

{

@Override

public void run()

{

System.out.println("Car is Running");

}

};

Vehicle bike = new Vehicle()

{

@Override

public void run()

{

System.out.println("Bike is Running");

}

};

car.run(); bike.run();

}

}

-------------------------------------------------------------

27-12-2024

-----------

What is Lambda Expression in java ?

------------------------------------

It is a new feature introduced in java from JDK 1.8 onwards.

It is an anonymous function i.e function without any name.

In java it is used to enable functional programming.

It is used to concise our code as well as we can remove boilerplate code.

Lambda will work only with functional interface.

If the body of the Lambda Expression contains only one statement then curly braces are optional.

We can also remove the variables type while defining the Lambda Expression parameter.

If the lambda expression method contains only one parameter then we can remove () symbol also.

In lambda expression return keyword is optional but if we use return keyword then {} are compulsory.

Independently Lamda Expression is not a statement.

It requires a target variable i.e functional interface reference only.

Lamda target can't be class or abstract class, it will work with functional interface only.

-------------------------------------------------------------

package com.ravi.lambda;

interface Vehicle

{

void run(); //SAM [Single Abstract Method]

}

public class LambdaDemo1 {

public static void main(String[] args)

{

Vehicle car = () -> System.out.println("Car is Running");

car.run();

Vehicle bike = () -> System.out.println("Bike is Running");

bike.run();

Vehicle bus = () -> System.out.println("Bus is Running");

bus.run();

}

}

-------------------------------------------------------------

package com.ravi.lambda;

@FunctionalInterface

interface Calculate

{

void doSum(int x , int y);

}

public class LambdaDemo2

{

public static void main(String[] args)

{

Calculate c1 = (c, d)-> System.out.println("Sum is :"+(c+d));

c1.doSum(12, 12);

}

}

-------------------------------------------------------------

package com.ravi.lambda;

import java.util.Scanner;

interface Length

{

public abstract int findLength(String str);

}

public class LambdaDemo3

{

public static void main(String[] args)

{

Length length = str -> str.length();

Scanner sc = new Scanner(System.in);

System.out.println("Please Enter your city Name :");

String cityName = sc.next();

System.out.println("Length of "+cityName+" is :"+length.findLength(cityName));

sc.close();

}

}

-------------------------------------------------------------

The following program explains that Lambda target must be

functional interface only.

@FunctionalInterface

interface Drawable

{

void draw();

}

public class Main

{

public static void main(String[] args)

{

Drawable d1 = ()-> System.out.println("Drawing");

d1.draw();

}

}

-------------------------------------------------------------

What is type parameter<T> in java ?

------------------------------------

It is a technique through which we can make our application indepenedent of data type. It is represented by <T>

In java we can pass Wrapper classes as well as User-defined classes to this type parameter(Only Reference type is reqd).

We cannot pass any primitive type to this type parameter.

package com.ravi.type\_parameter;

class Box<T>

{

private T data;

public Box(T data) //Student data =

{

super();

this.data = data;

}

public T getData()

{

return data;

}

}

public class TypeParameterDemo

{

public static void main(String[] args)

{

Box<Integer> intType = new Box<Integer>(12);

System.out.println("Integer Type :"+intType.getData());

Box<Double> doubleType = new Box<Double>(23.89);

System.out.println("Double Type :"+doubleType.getData());

Box<Boolean> booleanType = new Box<Boolean>(true);

System.out.println("Boolean Type :"+booleanType.getData());

Box<Student> studentType = new Box<Student>(new Student(111));

System.out.println("Student Type :"+studentType.getData());

}

}

class Student

{

private int id;

public Student(int id)

{

super();

this.id = id;

}

@Override

public String toString()

{

return "Student [id : " + id + "]";

}

}

-------------------------------------------------------------

28-12-2024

-----------

//Program by using Lambda to verify whether a number is even or

odd

import java.util.\*;

@FunctionalInterface

interface Predictable

{

boolean predict(int x);

}

public class Main

{

public static void main(String[] args)

{

Predictable p1 = num ->

{

return num % 2 == 0;

};

Scanner sc = new Scanner(System.in);

System.out.print("Enter a Number :");

int no = sc.nextInt();

boolean isEven = p1.predict(no);

System.out.println("Is "+no+ " even number "+isEven);

}

}

---------------------------------------------------------------

Working with predefined functional interfaces :

------------------------------------------------------

In order to help the java programmer to write concise java code in day to day programming java software people has provided the following predefined functional interfaces

1) Predicate<T> boolean test(T x);

2) Consumer<T> void accept(T x);

3) Function<T,R> R apply(T x);

4) Supplier<T> T get();

5) BiPredicate<T,U> boolean test(T x, U y);

6) BiConsumer<T, U> void accept(T x, U y);

7) BiFunction<T,U,R> R apply(T x, U y);

8) UnaryOperator<T> T apply(T x)

9) BinaryOperator<T> T apply(T x, T y)

Note :-

-------

All these predefined functional interfaces are provided as a part of java.util.function sub package.

Predicate<T> functional interface :

-------------------------------------------

It is a predefined functional interface available in java.util.function sub package.

It contains an abstract method test() which takes type parameter <T> and returns boolean value. The main purpose of this interface to test one argument boolean expression.

@FunctionalInterface

public interface Predicate<T>

{

boolean test(T x);

}

Note :- Here T is a "type parameter" and it can accept any type of User defined class as well as Wrapper class like Integer, Float, Double and so on.

We can't pass primitive type.

-----------------------------------------------------------

Programs on Predicate :

-----------------------

package com.ravi.predicate\_demo;

import java.util.Scanner;

import java.util.function.Predicate;

public class PredicateDemo1

{

public static void main(String[] args)

{

//Verify whether a person is eligible for vote or not

Predicate<Integer> p1 = age ->

{

return age >=18;

};

Scanner sc = new Scanner(System.in);

System.out.print("Enter your Age :");

int age = sc.nextInt();

boolean isEligible = p1.test(age);

if(isEligible)

{

System.out.println("You are eligible for Voting");

}

else

{

System.out.println("You are not eligible for Voting");

}

sc.close();

}

}

--------------------------------------------------------------

package com.ravi.predicate\_demo;

import java.util.Scanner;

import java.util.function.Predicate;

public class PredicateDemo2

{

public static void main(String[] args)

{

//Verify whether my name is Ravi or not

Predicate<String> p2 = str -> str.equalsIgnoreCase("Ravi");

Scanner sc = new Scanner(System.in);

System.out.print("Enter your Name :");

String name = sc.next();

System.out.println("Are you Ravi :? "+p2.test(name));

sc.close();

}

}

---------------------------------------------------------------

package com.ravi.predicate\_demo;

import java.util.Scanner;

import java.util.function.Predicate;

public class PredicateDemo3 {

public static void main(String[] args)

{

//Verify whether my name starts with Character 'A' or not

Predicate<String> p3 = str -> str.startsWith("A");

Scanner sc = new Scanner(System.in);

System.out.print("Enter your Name :");

String name = sc.next();

System.out.println("The name "+name+" starts with character A ?:"+p3.test(name));

sc.close();

}

}

---------------------------------------------------------------

Consumer<T> functional interface :

-----------------------------------------

It is a predefined functional interface available in java.util.function sub package.

It contains an abstract method accept() which takes T type parameter and returns nothing (void). It is used to accept the parameter value or consume the value.

@FunctionalInterface

public interface Consumer<T>

{

void accept(T x);

}

---------------------------------------------------------------

package com.ravi.consumer\_demo;

import java.util.function.Consumer;

public class ConsumerDemo {

public static void main(String[] args) {

Consumer<Integer> c1 = num -> System.out.println(num);

c1.accept(12);

Consumer<String> c2 = str -> System.out.println(str);

c2.accept("Java");

Consumer<Employee> c3 = emp -> System.out.println(emp);

c3.accept(new Employee(111, "Scott"));

}

}

class Employee

{

private int empId;

private String empName;

public Employee(int empId, String empName)

{

super();

this.empId = empId;

this.empName = empName;

}

@Override

public String toString()

{

return "Employee [empId=" + empId + ", empName=" + empName + "]";

}

}

---------------------------------------------------------------

Function<T,R> functional interface :

-----------------------------------------

Type Parameters:

T - the type of the input to the function.

R - the type of the result of the function.

It is a predefined functional interface available in java.util.function sub package.

It provides an abstract method apply that accepts one argument(T) and produces a result(R).

Note :- The type of T(input) and the type of R(Result) both will be decided by the user.

@FunctionalInterface

public interface Function<T,R>

{

public abstract R apply(T x);

}

--------------------------------------------------------------

package com.ravi.function;

import java.util.function.Function;

public class FunctionDemo1 {

public static void main(String[] args)

{

//Find the cube of a number

Function<Integer,Integer> fn1 = num -> num\*num\*num;

System.out.println(fn1.apply(5));

}

}

--------------------------------------------------------------

package com.ravi.function;

import java.util.Scanner;

import java.util.function.Function;

public class FunctionDemo2 {

public static void main(String[] args)

{

// I want to find the length of the given city

Function<String,Integer> fn2 = city -> city.length();

Scanner sc = new Scanner(System.in);

System.out.print("Enter your City Name :");

String city = sc.next();

System.out.println("The length of "+city+" is :"+fn2.apply(city));

sc.close();

}

}

---------------------------------------------------------------

30-12-2024

----------

package com.ravi.interface\_demo;

import java.sql.SQLSyntaxErrorException;

import java.util.Scanner;

import java.util.function.Function;

public class FunctionDemo3 {

public static void main(String[] args)

{

//Verify whether my name starts with "Ravi" or not

Function<String,Boolean> fn3 = str -> str.equals("Ravi");

Scanner sc = new Scanner(System.in);

System.out.print("Enter your Name :");

String name = sc.next();

Boolean isRavi = fn3.apply(name);

if(isRavi)

{

System.out.println("Yes He is Ravi");

}

else

{

System.err.println("No He is not Ravi");

}

sc.close();

}

}

---------------------------------------------------------------

Supplier<T> prdefined functional interface :

--------------------------------------------

It is a predefined functional interface available in java.util.function sub package.

It provides an abstract method get() which does not take any argument but produces/supply/return a value of type T.

@FunctionalInterface

public interface Supplier<T>

{

T get();

}

--------------------------------------------------------------

//Psssing String to the Supplier

package com.ravi.interface\_demo;

import java.util.function.Supplier;

public class SupplierDemo1

{

public static void main(String[] args)

{

Supplier<String> s1 = () -> 12 + 90+ " " + 34 ;

String data = s1.get();

System.out.println(data);

}

}

---------------------------------------------------------------

//Passing an Employee to the Supplier

package com.ravi.interface\_demo;

import java.util.Objects;

import java.util.function.Supplier;

class Employee

{

private Integer employeeId;

private String employeeName;

public Employee(Integer employeeId, String employeeName)

{

super();

this.employeeId = employeeId;

this.employeeName = employeeName;

}

@Override

public String toString()

{

return "Employee [employeeId=" + employeeId + ", employeeName=" + employeeName + "]";

}

public Integer getEmployeeId() {

return employeeId;

}

public void setEmployeeId(Integer employeeId) {

this.employeeId = employeeId;

}

public String getEmployeeName() {

return employeeName;

}

public void setEmployeeName(String employeeName) {

this.employeeName = employeeName;

}

@Override

public int hashCode()

{

return Objects.hash(employeeId, employeeName);

}

@Override

public boolean equals(Object obj) {

if (this == obj)

return true;

if (obj == null)

return false;

if (getClass() != obj.getClass())

return false;

Employee other = (Employee) obj;

return Objects.equals(employeeId, other.employeeId) && Objects.equals(employeeName, other.employeeName);

}

}

public class SupplierDemo2

{

public static void main(String[] args)

{

Supplier<Employee> s2 = () ->

{

Employee e1 = new Employee(111, "Scott");

return e1;

};

System.out.println(s2.get());

}

}

---------------------------------------------------------------

Object class equals(Object obj) :

----------------------------------

In java.lang.Object class, There is a predefined non static method called equals(Object obj) which is used to compere two objects by using == operator i.e memory reference. This method

behavior is same as == operator behavior.

If we want to compare two objects based on the content but not by using == operator then we should override this equals(Object obj) from Object class.

Eclipse IDE auto generate hashCode() and equals(Object obj) method, both are overridden method from the Object class as

shown below.

EqualsObjectComparison.java

-----------------------------

package com.ravi.interface\_demo;

import java.util.Objects;

class Student

{

private Integer studentId;

private String studentName;

public Student(Integer studentId, String studentName)

{

super();

this.studentId = studentId;

this.studentName = studentName;

}

public Integer getStudentId() {

return studentId;

}

public void setStudentId(Integer studentId) {

this.studentId = studentId;

}

public String getStudentName() {

return studentName;

}

public void setStudentName(String studentName) {

this.studentName = studentName;

}

@Override

public String toString() {

return "Student [studentId=" + studentId + ", studentName=" + studentName + "]";

}

//generate hashCode() and equals() method for content comparison

@Override

public int hashCode() {

return Objects.hash(studentId, studentName);

}

@Override

public boolean equals(Object obj) {

if (this == obj)

return true;

if (obj == null)

return false;

if (getClass() != obj.getClass())

return false;

Student other = (Student) obj;

return Objects.equals(studentId, other.studentId) && Objects.equals(studentName, other.studentName);

}

}

public class EqualsObjectComparison {

public static void main(String[] args)

{

Student s1 = new Student(111, "Raj");

Student s2 = new Student(111, "Raj");

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

System.out.println(s1.equals(s2)); //true (Overridden Method)

}

}

---------------------------------------------------------------

Record class in java :

-----------------------

public abstract class Record extends Object.

record Student(){} //final class Student extends Record [Compiler generated code]

It is a new feature introduced from java 17.(In java 14 preview version)

As we know only objects are moving in the network from one place to another place so we need to write BLC class with nessacery requirements to make BLC class as a Data carrier class.

Records are immutable data carrier so, now with the help of record we can send our immutable data (final data) from one application to another application.

It is also known as DTO (Data transfer object) OR POJO (Plain Old Java Object) classes.

It is mainly used to concise our code as well as remove the boiler plate code.

In record, automatically constructor will be generated which is known as canonical constructor and the variables which are known as components are by default final.

In order to validate the outer world data, we can write our own constructor which is known as compact constructor.

Record will automatically generate the implemenation of toString(), equals(Object obj) and hashCode() method.

We can define static and non static method as well as static variable and static block inside the record. We cannot define instance variable and instance block inside the record.

We cann't extend or inherit records because by default every record is implicilty final and It is extending from java.lang.Reocrd class, which is an abstract class.

We can implement an interface by using record.

We don't have setter facility in record because by default components are final.

---------------------------------------------------------------

3 files :

----------

ProductClass.java

------------------

package com.ravi.record;

import java.util.Objects;

public class ProductClass

{

private Integer productId;

private String productName;

public ProductClass(Integer productId, String productName)

{

super();

this.productId = productId;

this.productName = productName;

}

@Override

public String toString() {

return "ProductClass [productId=" + productId + ", productName=" + productName + "]";

}

public Integer getProductId() {

return productId;

}

public void setProductId(Integer productId) {

this.productId = productId;

}

public String getProductName() {

return productName;

}

public void setProductName(String productName) {

this.productName = productName;

}

@Override

public int hashCode() {

return Objects.hash(productId, productName);

}

@Override

public boolean equals(Object obj) {

if (this == obj)

return true;

if (obj == null)

return false;

if (getClass() != obj.getClass())

return false;

ProductClass other = (ProductClass) obj;

return Objects.equals(productId, other.productId) && Objects.equals(productName, other.productName);

}

}

ProductRecord.java

-------------------

package com.ravi.record;

public record ProductRecord(Integer productId, String productName)

{

//Compact Constructor

public ProductRecord

{

if(productId<=0)

{

System.err.println("Id is Invalid");

}

}

}

ClassAndRecordComparison.java

--------------------------------

package com.ravi.record;

public class ClassAndRecordComparison {

public static void main(String[] args)

{

ProductClass pc1 = new ProductClass(111, "Camra");

System.out.println(pc1);

ProductClass pc2 = new ProductClass(111, "Camra");

System.out.println(pc1.equals(pc2));

System.out.println(pc1.getProductName());

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

ProductRecord pr1 = new ProductRecord(-999, "Java Book");

System.out.println(pr1);

ProductRecord pr2 = new ProductRecord(999, "Java Book");

System.out.println(pr1.equals(pr2));

System.out.println(pr1.productName());

}

}

--------------------------------------------------------------

//supply manager object by using Supplier

package com.ravi.interface\_demo;

import java.util.function.Supplier;

record Manager(Integer managerId, String managerName)

{

}

public class SupplierDemo3

{

public static void main(String[] args)

{

Supplier<Manager> s3 = () -> new Manager(111, "John");

System.out.println(s3.get());

}

}

---------------------------------------------------------------

31-12-2024

----------

Can we create our own functional interface with Type parameter:

---------------------------------------------------------------

Yes, We can create our own functional interface with type parameter as shown below.

package com.nit.custom\_functional\_interface;

@FunctionalInterface

interface TriFunction<T,U,V,R>

{

public abstract R apply(T a, U b, V c);

}

public class CustomFunctionalInterface {

public static void main(String[] args)

{

TriFunction<Integer, Integer, Integer, String> tn = (x,y,z) -> ""+ x + y + z;

System.out.println(tn.apply(100, 200, 300));

}

}

---------------------------------------------------------------

BiPredicate<T,U> functional interface :

-----------------------------------

It is a predefined functional interface available in java.util.function sub package.

It is a functional interface in Java that represents a predicate (a boolean-valued function) OF TWO ARGUMENTS.

The BiPredicate interface has method named test, which takes two parameters and returns a boolean value, basically this BiPredicate is same with the Predicate, instead, it takes 2 arguments for the metod test.

@FunctionalInterface

public interface BiPredicate<T, U>

{

boolean test(T t, U u);

}

Type Parameters:

T - the type of the first argument to the predicate

U - the type of the second argument the predicate

Note : return type is boolean.

--------------------------------------------------------------

import java.util.function.\*;

public class Lambda11

{

public static void main(String[] args)

{

BiPredicate<String, Integer> filter = (x, y) ->

{

return x.length() == y;

};

boolean result = filter.test("Ravi", 4);

System.out.println(result);

result = filter.test("Hyderabad", 10);

System.out.println(result);

}

}

--------------------------------------------------------------

import java.util.function.BiPredicate;

public class Lambda12

{

public static void main(String[] args)

{

// BiPredicate to check if the sum of two integers is even

BiPredicate<Integer, Integer> isSumEven = (a, b) -> (a + b) % 2 == 0;

System.out.println(isSumEven.test(2, 3));

System.out.println(isSumEven.test(5, 7));

}

}

--------------------------------------------------------------

BiConsumer<T, U> functional interface :

---------------------------------------

It is a predefined functional interface available in java.util.function sub package.

It is a functional interface in Java that represents an operation that accepts two input arguments and returns no result.

It takes a method named accept, which takes two parameters and performs an action without returning any result.

@FunctionalInterface

public interface BiConsumer<T, U>

{

void accept(T t, U u);

}

--------------------------------------------------------------

import java.util.function.BiConsumer;

public class Lambda13

{

public static void main(String[] args)

{

BiConsumer<Integer, String> updateVariables = (num, str) ->

{

num = num \* 2;

str = str.toUpperCase();

System.out.println("Updated values: " + num + ", " + str);

};

int number = 15;

String text = "nit";

updateVariables.accept(number, text);

// Values after the update (note that the original values are unchanged)

System.out.println("Original values: " + number + ", " + text);

}

}

--------------------------------------------------------------

BiFunction<T, U, R> Functional interface :

---------------------------------

It is a predefined functional interface available in java.util.function sub package.

It is a functional interface in Java that represents a function that accepts two arguments and produces a result R.

The BiFunction interface has a method named apply that takes two arguments and returns a result of type R.

@FunctionalInterface

public interface BiFunction<T, U, R>

{

R apply(T t, U u);

}

--------------------------------------------------------------

import java.util.function.BiFunction;

public class Lambda14

{

public static void main(String[] args)

{

// BiFunction to concatenate two strings

BiFunction<String, String, String> concatenateStrings = (str1, str2) -> str1 + str2;

String result = concatenateStrings.apply("Hello", " Java");

System.out.println(result);

// BiFunction to find the length two strings

BiFunction<String, String, Integer> concatenateLength = (str1, str2) -> str1.length() + str2.length();

Integer result1 = concatenateLength.apply("Hello", "Java");

System.out.println(result1);

}

}

--------------------------------------------------------------

UnaryOperator<T> :

------------------

It is a predefined functional interface available in java.util.function sub package.

It is a functional interface in Java that represents an operation on a single operand that produces a result of the same type as its operand. This is a specialization of Function for the case where the operand and result are of the same type.

It has a single type parameter, T, which represents both the operand type and the result type.

@FunctionalInterface

public interface UnaryOperator<T> extends Function<T,T>

{

public abstract T apply(T x);

}

--------------------------------------------------------------

import java.util.function.\*;

public class Lambda15

{

public static void main(String[] args)

{

UnaryOperator<Integer> square = x -> x\*x;

System.out.println(square.apply(5));

UnaryOperator<String> concat = str ->

str.concat("base");

System.out.println(concat.apply("Data"));

}

}

--------------------------------------------------------------

BinaryOperator<T>

-----------------

It is a predefined functional interface available in java.util.function sub package.

It is a functional interface in Java that represents an operation upon two operands of the same type, producing a result of the same type as the operands.

This is a specialization of BiFunction for the case where the operands and the result are all of the same type.

It has two parameters of same type, T, which represents both the operand types and the result type.

@FunctionalInterface

public interface BinaryOperator<T> extends BiFunction<T,T,T>

{

public abstract T apply(T x, T y);

}

import java.util.function.\*;

public class Lambda16

{

public static void main(String[] args)

{

BinaryOperator<Integer> add = (a, b) -> a + b;

System.out.println(add.apply(3, 5));

}

}

============================================================

Can we use all different types of variables in the lambda body ?

-----------------------------------------------------------------

Yes, we can use all different types of variables like static field, non static field and local variable in the lambda body.

The local variable must be final or effectively final otherwise we will get compilation error.

@FunctionalInterface

interface Drawable

{

void draw();

}

public class Main

{

public static void main(String[] args)

{

String shapeType = "Rectangle"; //local variable

// shapeType = "Square"; //[It is Invalid]

Drawable d1 = ()->

{

System.out.println("Drawing : "+shapeType);

};

d1.draw();

}

}

==================================================================

01-01-2025

----------

Does an interface extend a class :

----------------------------------

No, an interface can't extend a class, An interface can extend another interface.

In order to support upcasting concept in java, all the public and non final methods of Object class are re-declared inside an interface as an abstract method if that interface does not extend from another interface.

InterfaceMemberDemo1.java

-----------------------------

package com.ravi.interface\_member;

interface Printable

{

//public String toString(); //added by compiler

//public int hashCode(); //added by compiler

//public boolean equals(Object obj); //added by compiler

}

class Print extends Object implements Printable

{

/\*public String toString()

\* {

\* //Inherited from Object class

\* }

\*

\* public int hashCode()

\* {

\* //Inherited from Object class

\* }

\*

\* public boolean equals(Object obj)

\* {

\* //Inherited from Object class

\* }

\*

\*

\*/

}

public class InterfaceMemberDemo1

{

public static void main(String[] args)

{

Printable p = new Print();

p.toString()

p.equals(null);

p.hashCode();

}

}

Note : java compiler will not add any final method of Object class inside an interface because abstract method we can't declare as final.

---------------------------------------------------------------

package com.ravi.interface\_member;

@FunctionalInterface

interface Drawable

{

void draw();

public String toString();

public int hashCode();

public boolean equals(Object obj);

}

public class InterfaceMemberDemo2

{

public static void main(String[] args)

{

}

}

-----------------------------------------------------------------

Can a default method of interface override/write the public method of Object class with same signature and return type.

No, a default method of an interface can't override/write the public method of Object class due to the following two reasons :

(01-JAN)

1) Ambiguity issue : Object class already contain the method which

interface wants to define as a default method hence it will provide Ambiguity to sub class

(Implementer class)

2) Priority : Concrete method is having more priority than

default method that means Object class method is having more priority than default method of interface so compiler will not allow to write any default method as a public method of Object class.

package com.ravi.interface\_member;

interface Moveable

{

default String toString() //error becoz it is Object class {

return "NIT";

}

}

public class ELC

{

public static void main(String[] args)

{

// TODO Auto-generated method stub

}

}

==================================================================

Interface from JAVA 9V

----------------------

We can write private static and private non static (not public)

methods inside an interface from java 9 version.

The main purpose of providing these two methods inside an

interface are as follows :

1) Code Reusability

--------------------

If two or more than two default methods want to share a common code (Helper Method code) then we can write these common code in private methods so it will enhance code reusability.

2) Hide the Logic from Outer World

----------------------------------

By writing these code in private static and private non static methods the actual logic is not visible to the outer world so, It is way to make our interface as a fully abstract class so 100% abstraction is possible.

Note : By default interface is not Fully abstract but we can make it full abstract from java 9V by writing the logic inside private method.

Note : from default method we can call private static as well as private non static methods but from public static method of interface we can call only private static method.

-----------------------------------------------------------------

package com.ravi.interface\_member;

interface Moveable

{

void move(); //JDK 1.0

default void m1() //JDK 8V

{

m3();

m4();

}

static void m2() //JDK 8V

{

m4();

}

private void m3() //JDK 9 [private non static method]

{

System.out.println("Private non static method of interface");

}

private static void m4() //JDK 9 [private static method]

{

System.out.println("Private static method of interface");

}

}

class Move implements Moveable

{

@Override

public void move()

{

System.out.println("It is moving");

}

}

public class ELC

{

public static void main(String[] args)

{

Moveable m = new Move();

m.move();

m.m1();

Moveable.m2();

}

}

-----------------------------------------------------------------

What is a Marker interface ?

-----------------------------

If an interface does not contain any field or methods, Basically an empty interface is known as Marker interface.

Example :

-----------

interface Drawable //Marker interface

{

//Empty interface

}

In java, we have few predefined marker interfaces are available which are as follows :

1) java.lang.Cloneable

2) java.io.Serializable

3) java.util.RandomAccess

------------------------------------------------------------------

\*\*\*\*What is difference between abstract class and interface ?

----------------------------------------------------------------

The following are the differences between abstract class and interface.

1) An abstract class can contain instance variables but interface variables are by default public , static and final (no instance variable).

2) An abstract class can have state (properties) of an object but interface can't have state of an object.

3) An abstract class can contain constructor but inside an interface we can't define constructor.

4) An abstract class can contain instance and static blocks but inside an interface we can't define any blocks.

5) Abstract class can't refer Lambda expression but using Functional interface we can refer Lambda Expression.

6) By using abstract class multiple inheritance is not possible but by using interface we can achieve multiple inheritance.

--------------OOPs Completed.................................

Offline : (38 - 40)

---------

1) Exception Handling

2) Array and String (Logical Session)

3) Multithreading

4) Collections Framework (Generic, Concurrent Collection, STREAM API)

Online :

---------

Inner class

Object class and its Method

File handling and Input and Output in java

==============================================================

02-01-2025

**--------------------------**

**Exception Handling**

**--------------------------**

An exception is an abnormal situation OR unexpected situation in a normal execution flow.

Due to an exception, the execution of the program will be disturbed first and then terminated permanently.

Exception always encounters at runtime only.

***Exception encounter due to the following reasons:***

1) The Wrong input is given by the user.

2) Due to dependency, When one part of the program is dependent to another part to complete the task then there might be a chance of getting an exception.

-------------------------------------------------------

**Different Criteria for Exception Generation**

-------------------------------------------------------

**1) java.lang.ArithmeticException**

Whenever we divide a number by 0 (int value) then we will get an exception i.e java.lang.ArithmeticException

int x = 100;

int y = 0;

int z = x /y; *//java.lang.ArithmeticException*

----------------------------------------------------------------------

**2) java.lang.ArrayIndexOutOfBoundsException**

Whenever we try to access the index of the array where element is not available then we will get java.lang.ArrayIndexOutOfBoundsException

int []arr = {10,20,30};

System.out.println(arr[3]);

----------------------------------------------------------------

**3) java.lang.NegativeArraySizeException**

The size of an array must be positive integer, if we pass -ve size then we will get an exception java.lang.NegativeArraySizeException

int [] arr = new int[-10];

--------------------------------------------------------

**4) java.lang.NumberFormatException**

If we try to convert any String value into corresponding primitive type OR Warpper type and if the String data is not available in numeric format then we will get an exception java.lang.NumberFormatException

String str = "h";

int x = Integer.parseInt(str);

System.out.println(x+2);

String s1 = "NIT";

Integer y = Integer.valueOf(s1);

System.out.println(y+2);

5) java.lang.NullPointerException :

------------------------------------

If a reference variable is pointing to null and if we try to invoke any non static method or field by using reference variable then we will get java.lang.NullPointerException

Case 1 :

--------

String str = null;

System.out.println(str.length()); //Exception

Case 2 :

--------

String str = "null";

System.out.println(str.length()+" : "+str.toUpperCase());

Case 3 :

--------

Scanner sc = new Scanner(System.in);

System.out.println("Enter your name :");

String name = sc.nextLine(); //"null"

System.out.println(name.length()+" : "+name.toUpperCase());

6) java.util.InputMismatchException :

--------------------------------------

If we try to read the data by using Scanner class method and if the data is not in a proper format/input then we will get exception java.util.InputMismatchException

Scanner sc = new Scanner(System.in);

System.out.println("Enter your Salary :");

double salary = sc.nextDouble(); //Eleven [Exception]

System.out.println("Your Salary is :"+salary);

7) java.lang.StringIndexOutOfBoundsException :

-----------------------------------------------

If we try to access the index of String (character Array) which index is not available then we will get java.lang.StringIndexOutOfBoundsException

String str = "hyderabad";

System.out.println(str.substring(-2,5));

Note : ArrayIndexOutOfBoundsException class StringIndexOutOfBoundsException class both are sub class of java.lang.IndexOutOfBoundsException.

--------------------------------------------------------------

06-01-2025

-----------

Exception Hierarchy :

----------------------

Hierarchy is available in paint diagram[06th-JAN-24]

Note :- As a developer we are responsibe to handle the Exception. System admin is responsibe to handle the error because we cannot recover from error.

--------------------------------------------------------------

WAP to show that Exception is the super class of all types of

Exception whether it is checked Or unchecked

package com.ravi.excp;

public class ExceptionSuper {

public static void main(String[] args)

{

Exception e1 = new ArithmeticException("Ravi divides by zero");

System.out.println(e1.toString()); //java.lang.AE : Ravi divides by zero

Exception e2 = new InterruptedException();

System.out.println(e2.toString()); //java.lang.IE

}

}

-------------------------------------------------------------

Exception format :

------------------

The java software people has provided the format of exception so whenever we print exception object by using toString() then the format is

Fully Qualified Name : errorMesage

Package Name + Class Name : errorMessage

------------------------------------------------------------

WAP that desribes, whenever an exception is encounter in the

program then our program will be terminated abnormally (halt in the middle)

package com.ravi.excp;

import java.util.Scanner;

public class AbnormalTermination {

public static void main(String[] args)

{

System.out.println("Main method Started!!!");

Scanner sc = new Scanner(System.in);

System.out.print("Enter the value of x :");

int x = sc.nextInt();

System.out.print("Enter the value of y :");

int y = sc.nextInt();

int result = x/y;

System.out.println("Result is :"+result);

System.out.println("Main method Completed!!!");

sc.close();

}

}

Note :

In the above program, If we put the value of y as 0 then program will be terminated in the middle, IT IS CALLED ABNORMAL TERMINATION.

Actually JVM has a default exception handler which is responsible to handle the execption and terminate the program in the middle abnormaly.

--------------------------------------------------------------

In order to handle the exception and to work with exception. java software people has provided the following keywords :

1) try block

2) catch block

3) finally block (try with resourses)

4) throw

5) throws

-------------------------------------------------------------

07-01-2025

-----------

Key points to remember :

--------------------------------

-> With try block we can write either catch block or finally block or both.

-> In between try and catch we can't write any kind of statement.

-> try block will trace our program line by line.

-> If we have any exception inside the try block,With the help of JVM, try block will automatically create the appropriate Exception object and then throw the Exception Object to the nearest catch block.

-> In the try block whenever we get an exception the control will directly jump to the nearest catch block so the remaining code of try block will not be executed.

-> catch block is responsible to handle the exception.

-> catch block will only execute if there is an exception inside try block.

--------------------------------------------------------------

try block :

-----------

Whenever our statement is error suspecting statement OR Risky statement then we should write that statement inside the try block.

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

\*try block is responsible to trace our code line by line, if any execption is encountered then with the help of JVM, TRY BLOCK WILL CREATE APPROPRIATE EXECPTION OBJECT, AND THROW THIS EXCEPTION OBJECT to the nearest catch block.

After the execption in the try block, the remaining code of try block will not be executed because control will directly transfer to the catch block.

In between try and catch block we cannot write any kind of statement.

catch block :

--------------

The main purpose of catch block to handle the exception which is thrown by try block.

catch block will only executed if there is an exception in the try block.

-------------------------------------------------------------

package com.ravi.basic;

import java.util.Scanner;

public class TryDemo

{

public static void main(String[] args)

{

System.out.println("Main method started....");

Scanner sc = new Scanner(System.in);

try

{

System.out.print("Enter the value of x :");

int x = sc.nextInt();

System.out.print("Enter the value of y :");

int y = sc.nextInt();

int result = x /y;

System.out.println("Result is :"+result);

System.out.println("End of try block");

}

catch(Exception e)

{

System.out.println("Inside Catch");

System.err.println(e);

}

System.out.println("Main method ended....");

sc.close();

}

}

In the above program if we put the value of y as 0 but still program will be executed normally because we have used try-catch so it is a

normal termination even we have an exception in the program.

-------------------------------------------------------------

package com.ravi.basic;

public class ThrowException

{

public static void main(String[] args)

{

try

{

//System.out.println(10/0);

//OR

throw new ArithmeticException("Ravi is dividing a number by zero");

}

catch(Exception e)

{

System.out.println("Catch Block");

System.err.println(e);

}

System.out.println("Main completed!!!");

}

}

From the above program it is clear that try block implicitly creating the exception object with the help of JVM and throwing the execption object to the nearest catch block.

We can also create the exception object manually and can throw the exception object to the catch block.

After throw keyword we can't write any kind of statement, It

will become un-reachable code.

--------------------------------------------------------------

class Foo

{

}

public class ExceptionDemo

{

public static void main(String[] args)

{

try

{

throw new Foo();

}

catch (Exception e)

{

System.out.println(e);

}

}

}

Note : We will get compiletion error because Foo class does not belong to exception hierarchy so it is not a throwable object.

--------------------------------------------------------------

Note : The main purpose of exception handling to provide user-friendly message so client can enjoy the services of software/websites.

Exception handlinag = No Abnormal Termination + User-friendly message on wrong input given by the client.

package com.ravi.basic;

import java.util.Scanner;

public class CustomerDemo

{

public static void main(String[] args)

{

System.out.println("Hello Client, Welcome to my application :");

Scanner sc = new Scanner(System.in);

try

{

System.out.print("Enter the value of x :");

int x = sc.nextInt();

System.out.print("Enter the value of y :");

int y = sc.nextInt();

int result = x /y;

System.out.println("Result is :"+result);

}

catch(Exception e)

{

System.err.println("Please don't put zero here");

}

System.out.println("Thank you for Visiting my application !!");

sc.close();

}

}

-------------------------------------------------------------

Throwable class Method to print Exception :

--------------------------------------------

Throwable class has provided the following three methods :

1) public String getMessage() :- It will provide only error message.

2) public void printStackTrace() :- It will provide the complete details regarding exception like exception class name, exception error message, exception class location, exception method name and exception line number.

3) public String toString() :- It will convert the exception into String representation.

-------------------------------------------------------------

package com.ravi.basic;

public class PrintStackTrace

{

public static void main(String[] args)

{

System.out.println("Main Method started");

try

{

String str = null;

System.out.println(str.toUpperCase());

}

catch(Exception e)

{

System.out.println("Ref variable is pointing to null");

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

System.out.println(e.getMessage());

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

System.out.println(e.toString());

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

e.printStackTrace();

}

System.out.println("Main Method ended");

}

}

--------------------------------------------------------------

Working with Specific Exception :

---------------------------------

While working with exception, in the corresponding catch block we can take Exception (super class) which can handle any type of Exception.

On the other hand we can also take specific type of exception (ArithmetiException, InputMismatchException and so on) which will handle only one type i.e specific type of exception.

package com.ravi.basic;

public class PrintStackTrace

{

public static void main(String[] args)

{

System.out.println("Main Method started");

try

{

String str = null;

System.out.println(str.toUpperCase());

}

catch(NullPointerException e)

{

System.out.println("Ref variable is pointing to null");

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

System.out.println(e.getMessage());

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

System.out.println(e.toString());

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

e.printStackTrace();

}

System.out.println("Main Method ended");

}

}

-------------------------------------------------------------

public class ExceptionDemo

{

public static void main(String[] args)

{

try

{

System.out.println("Inside Try!!!");

throw new OutOfMemoryError();

}

catch (Exception e)

{

System.out.println("Inside Catch!!!");

System.out.println(e);

}

}

}

Note : OutOfMemoryError can't be handled by Exception class.We need to write Error OR Throwable class.

--------------------------------------------------------------

Working with Infinity and Not a number(NaN) :

---------------------------------------------

10/0 -> Infinity (Java.lang.ArithmeticException)

10/0.0 -> Infinity (POSITIVE\_INFINITY)

0/0 -> Undefined (Java.lang.ArithmeticException)

0/0.0 -> Undefined (NaN)

While dividing a number with Integral literal in both the cases i.e Infinity (10/0) and Undefined (0/0) we will get java.lang.ArithmeticException because java software people has not provided any final, static variable support to deal with Infinity and Undefined.

On the other hand while dividing a number with with floating point literal in the both cases i.e Infinity (10/0.0) and Undefined (0/0.0) we have final, static variable support so the program will not be terminated in the middle which are as follows

10/0.0 = POSITIVE\_INFINITY

-10/0.0 = NEGATIVE\_INFINITY

0/0.0 = NaN

java.lang.Float and java.lang.Double classes are provided the support for these final and static variable, the same OR same type of variables are not available in Integeral Literal classes.

package com.ravi.basic;

public class InfinityFloatingPoint

{

public static void main(String[] args)

{

System.out.println("Main Started");

System.out.println(10/0.0);

System.out.println(-10/0.0);

System.out.println(0/0.0);

System.out.println(10/0);

System.out.println("Main Ended");

}

}

--------------------------------------------------------------

08-01-2025

-----------

Working with multiple try catch :

---------------------------------

According to our application requirement we can provide multiple try-catch in my application to work with multiple execptions.

package com.ravi.basic;

public class MultipleTryCatch

{

public static void main(String[] args)

{

System.out.println("Main method started!!!!");

try

{

int arr[] = {10,20,30};

System.out.println(arr[3]);

}

catch(ArrayIndexOutOfBoundsException e)

{

System.err.println("Array index is out of limit!!!");

}

try

{

String str = null;

System.out.println(str.length());

}

catch(NullPointerException e)

{

System.err.println("ref variable is pointing to null");

}

System.out.println("Main method ended!!!!");

}

}

Note : In the above program the drawback is, Client will get all the error messages at once which is not a recommended way.

In order to avoid this drawback we introduced multiple catch blocks with single try.

--------------------------------------------------------------

try with multiple catch blocks :

---------------------------------

According to industry standard we should write try with multiple catch blocks so we can provide proper information for each and every exception to the end user.

While working with multiple catch block always the super class catch block must be last catch block.

From java 1.7v this multiple exceptions we can write in a single catch block by using | symbol.

If try block is having more than one exception then always try block will entertain only first exception because control will transfer to the nearest catch block.

--------------------------------------------------------------

package com.ravi.basic;

public class MultyCatch

{

public static void main(String[] args)

{

System.out.println("Main Started...");

try

{

int c = 10/2;

System.out.println("c value is :"+c);

int []x = {12,78,56};

System.out.println(x[4]);

}

catch(ArrayIndexOutOfBoundsException e1)

{

System.err.println("Array is out of limit...");

}

catch(ArithmeticException e1)

{

System.err.println("Divide By zero problem...");

}

catch(Exception e1)

{

System.out.println("General");

}

System.out.println("Main Ended...");

}

}

-------------------------------------------------------------

package com.ravi.basic;

public class MultyCatch1

{

public static void main(String[] args)

{

System.out.println("Main method started!!!");

try

{

String str1 = "null";

System.out.println(str1.toUpperCase()); //NULL

String str2 = "Ravi";

int x = Integer.parseInt(str2);

System.out.println("Number is :"+x);

}

catch(NumberFormatException | NullPointerException e)

{

if(e instanceof NumberFormatException)

{

System.err.println("Number is not in a proper format");

}

else if(e instanceof NullPointerException)

{

System.err.println("ref variable is pointing to null");

}

}

System.out.println("Main method ended!!");

}

}

--------------------------------------------------------------

finally is a block which is meant for Resource handling purposes.

According to Software Engineering, the resources are memory creation, buffer creation, opening of a database, working with files, working with network resourses and so on hence these resourses must be closed properly.

Whenever the control will enter inside the try block always the finally block would be executed.

We should write all the closing statements inside the finally block because irrespective of exception finally block will be executed every time.

If we use the combination of try and finally then only the resources will be handled but not the execption, on the other hand if we use try-catch and finally then execption and resourses both will be handled.

--------------------------------------------------------------

package com.ravi.basic;

public class FinallyBlock

{

public static void main(String[] args)

{

System.out.println("Main method started");

try

{

System.out.println(10/0);

}

finally

{

System.out.println("Finally Block");

}

System.out.println("Main method ended");

}

}

Note :- In the above program finally block will be executed, even we have an exception in the try block but here only the resourses will be handled but not the exception.

--------------------------------------------------------------

package com.ravi.basic;

public class FinallyWithCatch

{

public static void main(String[] args)

{

try

{

int []x = new int[-2];

x[0] = 12;

x[1] = 15;

System.out.println(x[0]+" : "+x[1]);

}

catch(NegativeArraySizeException e)

{

System.err.println("Array Size is in negative value...");

}

finally

{

System.out.println("Resources will be handled here!!");

}

System.out.println("Main method ended!!!");

}

}

In the above program exception and resourses both are handled because we have a combination of try-catch and finally.

Note :- In the try block if we write System.exit(0) and if this line is executed then finally block will not be executed.

--------------------------------------------------------------

Limitation of finally block :

------------------------------

The following are the limitation of finally block :

1) In order to close the resourses, user is responsible to write finally block manually.

2) Due to finally block the length of the program will be increased.

3) In order to close the resourses inside the finally block, we need to declare the resourses outside of try block.

package com.ravi.basic;

import java.util.InputMismatchException;

import java.util.Scanner;

public class FinallyLimitation

{

public static void main(String[] args)

{

Scanner sc = null;

try

{

sc = new Scanner(System.in);

System.out.print("Enter your Age :");

int age = sc.nextInt();

System.out.println("Age is :"+age);

}

catch(InputMismatchException e)

{

System.err.println("Input mismatched..");

}

finally

{

System.out.println("Inside finally");

sc.close();

}

}

}

--------------------------------------------------------------

09-01-2025

----------

try with Resourses (Automatic Closing Facility)

------------------------------------------------

To avoid all the limitation of finally block, Java software people introduced a separate concept i.e try with resources from java 7 onwards.

Case 1:

-------

try(resource1 ; resource2) //Only the resources will be handled

{

}

Case 2 :

----------

//Resources and Exception both will be handled

try(resource1 ; resource2)

{

}

catch(Exception e)

{

}

Case 3 :

----------

try with resourses enhancement from java 9v

Resourse r1 = new Resourse();

Resourse r2 = new Resourse();

try(r1; r2)

{

}

catch(Exception e)

{

}

Note : The Resourse class must implements either from Closeable or AutoCloseable.

There is a predefined interface available in java.lang package called AutoCloseable from JDK 1.7 which contains predefined abstract method i.e close() which throws Exception.

There is another predefined interface available in java.io package called Closeable from JDK 1.5, this Closeable interface is the sub interface for AutoCloseable interface.

public interface java.lang.AutoCloseable

{

public abstract void close() throws Exception;

}

public interface java.io.Closeable extends java.lang.AutoCloseable

{

void close() throws IOException;

}

Whenever we pass any resourse class object as part of try with resources as a parameter then that class must implements either Closeable or AutoCloseable interface so, try with resourses will automatically call the respective class

close() method even an exception is encountered in the try block.

ResourceClass rc = new ResourceClass();

try(rc)

{

}

catch(Exception e)

{

}

This ResourceClass must implements either Closeable or AutoCloseable interface so, try block will automatically call the close() method as well as try block will get the guarantee of close() method support in the respective class.

The following program explains how try block is invoking the close() method available in DatabaseResource class and FileResourse class.

-------------------------------------------------------------

3 files :

----------

package com.ravi.try\_with\_resourses;

import java.io.Closeable;

import java.io.IOException;

public class FileResourse implements Closeable

{

@Override

public void close() throws IOException

{

System.out.println("File Resourse Closed!!!");

}

}

package com.ravi.try\_with\_resourses;

public class DatabaseResourse implements AutoCloseable

{

@Override

public void close() throws Exception

{

System.out.println("Database Resourse Closed!!!");

}

}

package com.ravi.try\_with\_resourses;

public class TryWithResourseDemo {

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

{

System.out.println("Main Started");

FileResourse fr = new FileResourse();

DatabaseResourse dr = new DatabaseResourse();

try(fr ; dr)

{

System.out.println(10/0);

}

catch(ArithmeticException e)

{

System.err.println("Divide by zero problem");

}

System.out.println("Main Ended");

}

}

--------------------------------------------------------------

package com.ravi.try\_with\_resourses;

import java.util.InputMismatchException;

import java.util.Scanner;

public class TryWithResourseDemo1

{

public static void main(String[] args)

{

Scanner sc = new Scanner(System.in);

try(sc)

{

System.out.println("Enter your Roll Number :");

int roll = sc.nextInt();

System.out.println("Your roll number is :"+roll);

}

catch(InputMismatchException e)

{

System.err.println("Input is mismatched!!!");

}

}

}

--------------------------------------------------------------

\*\*\*Difference between Checked Exception and Unchecked Exception :

------------------------------------------------------------

Checked Exception :

----------------------

A checked exception is a common exception that must be declared or handled by the application code where it is thrown, Here compiler takes very much care and wanted the clarity regarding the exception by saying that, by using this code you may face some problem at runtime and you did not report me how would you handle this situation at runtime are called Checked exception, so provide either try-catch or declare the method as throws.

Except RuntimeException, all the checked exceptions are directly sub class of java.lang.Exception OR Throwable.

Eg:

---

FileNotFoundException, IOException, InterruptedException,ClassNotFoundException, SQLException, CloneNotSupportedException, EOFException and so on

Unchecked Exception :-

--------------------------

An unchecked exception is rare and any exception that does not need to be declared or handled by the application code where it is thrown, here compiler does not take any care are called unchecked exception.

Unchecked exceptions are directly entertain by JVM because they are rarely occurred in java.

All the un-checked exceptions are sub class of RuntimeException as well as Error and all the sub classes of Error are considered as Unchecked Exception.

RuntimeException is also Unchecked Exception.

Eg:

---

ArithmeticException, ArrayIndexOutOfBoundsException, NullPointerException, NumberFormatException, ClassCastException, ArrayStoreException and so on.

-------------------------------------------------------------

Some Bullet points regarding Checked and Unchecked :

-----------------------------------------------------

Checked Exception :

------------------

1) Common Exception

2) Compiler takes care (Will not compile the code)

3) Handling is compulsory (try-catch OR throws)

4) Directly the sub class of java.lang.Exception OR Throwable

Unchecked Exception :

----------------------

1) Rare Exception

2) Comiler will not take any care

3) Handling is not Compulsory

4) Sub class of RuntimeException OR Error

\*Why compiler takes very much care regarding the checked Exception ?

---------------------------------------------------------------

As we know Checked Exceptions are very common exception so in case of checked exception "handling is compulsory" because checked Exception depends upon other resources as shown below.

IOException (we are depending upon System Keyboard OR Files )

FileNotFoundException(We are depending upon the file)

InterruptedException (Thread related problem)

ClassNotFoundException (class related problem)

SQLException (SQL related or database related problem)

CloneNotSupportedException (Object is the resourse)

EOFException(We are depending upon the file)

----------------------------------------------------------

When to provide try-catch or declare the method as throws for Checked Exception :-

try-catch

----------

We should provide try-catch if we want to handle the exception in the method where checked exception is encountered, as well as if we want to provide user-defined messages to the client.

throws :

--------

throws keyword describes that the method might throw an Exception, It also might not. It is used only at the end of a method declaration to indicate what exceptions it supports OR what type of Exception it might throw which will be handled by JVM or caller method.

Note :- It is always better to use try catch so we can provide appropriate user defined messages to our client.

------------------------------------------------------------

Exception propagation [Propagation of Exception from Callee to Caller]