**OOPs**

# 1. SIB – Static Initialization Block, Static Variables And Static Methods:

Static variables, Static Initialization Block and Static Methods – these all are static components or static members of a class. These static members are stored inside the Class Memory. To access static members, you need not to create objects. Directly you can access them with class name.

Static Initialization Block is used to initialize only static variables. It is a block without a name. It contains set of statements enclosed within { }. The syntax of SIB looks like this,

|  |  |
| --- | --- |
| 1  2  3  4 | static  {       //Set Of Statements  } |

Consider the following program.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | class StaticComponents  {       static int staticVariable;         static       {            System.out.println("StaticComponents SIB");            staticVariable = 10;       }         static void staticMethod()       {            System.out.println("From StaticMethod");            System.out.println(staticVariable);       }  }    public class MainClass  {       static       {            System.out.println("MainClass SIB");       }         public static void main(String[] args)       {           //Static Members directly accessed with Class Name            StaticComponents.staticVariable = 20;            StaticComponents.staticMethod();       }  } |

Let us discuss execution of above program step by step.

**Step 1:**

When you trigger >java MainClass, java command divides allocated memory into two parts – Stack and Heap. First, java command enters stack memory for execution. First, it checks whether **MainClass** is loaded into heap memory or not. If it is not loaded, loading operation of MainClass starts. Randomly some memory space is allocated to MainClass. It is called **Class memory**. All static members are loaded into this class memory. There is only one satic member in MainClass – main() method. It is loaded into class memory of MainClass.

**Step 2:**

After loading all static members, SIB – Static initialization Blocks are executed. Remember, **SIBs are not stored in the heap memory. They just come to stack, execute their tasks and leaves the memory**. So, after loading main() method, SIB of MainClass enters stack for execution. There is only one statement (Line 22) in SIB. it is executed. It prints “MainClass SIB” on console. After executing this statement, SIB leaves the stack memory.

**Step 3:**

Now, java command calls main() method for execution. main() method enters the stack. First statement (Line 28) is executed first. First, It checks whether class StaticComponents is loaded into memory. If it is not loaded, loading operation of StaticComponents takes place. Randomly, some memory is allocated to Class StaticComponents, then all static members of StaticComponents – ‘staticVariable’ and ‘staticMethod()’ are loaded into that class memory. ‘staticVariable’ is a global variable. So, first it is initialized with default value i.e 0.

**Step 4 :**

After loading all static members of StaticComponents, SIB blocks are executed. So, SIB of class StaticComponents enters the stack for execution. First Statement (Line 7) is executed. It prints “StaticComponents SIB” on the console. In the second statement, value 10 is assigned to ‘staticVariable’. There are no other statements left for execution, so it leaves stack memory.

**Step 5 :**

Now control comes back to main() method. The remaining part of first statement i.e value 20 is assigned to ‘staticVariable’ of class StaticComponents, is executed. In the second statement (Line 29), it calls staticMethod() of class StaticComponents for execution.

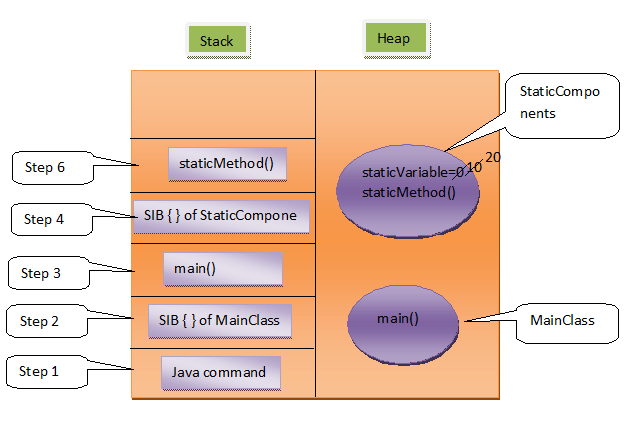
**Step 6:**

staticMethod() of StaticComponents enters stack for execution.  First statement (Line 13) is executed first. It prints “From staticMethod” on the console. In the second statement (Line 14), it prints the value of staticVariable i.e 20 on the console. There are no statements left. so, it leaves the stack.

**Step 7:**

Again, control comes back to main() method. There are no other statements left in main() method. so, it also leaves stack. java command also leaves the stack.

Diagramatic representation of memory allocation of above program looks like this.



**Output :**

Main Class SIB  
StaticComponents SIB  
From StaticMethod  
20

# 2. Non-Static Members And Their Memory Management In Java:

we will see non-static components of a class.

Let’s start with simple definitions of class and object.

Class : Class is the model/template/blueprint for the objects to be created of its type.

Object : It is an instance of a class. It is the real-time copy of class.

If you don’t understand with the definitions, read out this example. A class is like a blue print of a house. With this blueprint, you can build any number of houses. Each house build with this blueprint is an object or an instance of that blue print.

Non-Static variables and Non-Static methods are non-static components of a class. These are also called instance components of a class. Non-static components are stored inside the object memory. Each object will have their own copy of non-static components. But,  static components are common to all objects of that class.

Let’s have a look at this example.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | class A  {       int nonStaticVariable;       static int staticVariable;         static void staticMethod()       {            System.out.println(staticVariable);       //   System.out.println(nonStaticVariable);       }         void nonStaticMethod()       {            System.out.println(staticVariable);            System.out.println(nonStaticVariable);       }  }    class MainClass  {       public static void main(String[] args)       {            A.staticVariable = 10;       //   A.nonStaticVariable = 10;            A.staticMethod();      //    A.nonStaticMethod();              A a1 = new A();            A a2 = new A();              System.out.println(a1.nonStaticVariable);            System.out.println(a1.staticVariable);            a1.nonStaticMethod();            a1.staticMethod();              System.out.println(a2.staticVariable);            a1.staticVariable = 20;            System.out.println(a2.staticVariable);       }  } |

Let’s discuss memory allocation of above example step by step.

**Step 1 :**

When you trigger >java MainClass, java command divides allocated memory into two parts – stack and heap. First java command enters stack for execution. First it loads class **MainClass**into heap memory. Randomly some memory is allocated to MainClass. All static members are loaded into this class memory. There is only one static member in MainClass i.e main() method. It is loaded into class memory. After loading static members, SIBs are executed. But there is no SIBs in MainClass. So, directly java command calls main() method for execution.

**Step 2 :**

main() method enters stack for execution. First statement (Line 23) refers to class A. First it checks whether class A is loaded into heap memory or not. If it is not loaded, it loads class A into heap memory. Randomly some memory is allocated to class A. All static members of class A , ‘staticVariable’ and ‘staticMethod()’ , are loaded into this memory. ‘staticVariable’ is first initialized with default value 0. No SIBs in Class A. So, after loading static members, main() method assigns value 10 to ‘staticVariable’ of class A.

Second statement (Line 24) of main() method is commented. **Because, you can’t refer a non-static members through a class name. Because, non-static members are stored inside the object memory. You have to refer them through objects only.**

**Step 3 :**

In Line 25, it calls staticMethod() of class A. staticMethod() comes to stack for execution. First statement(Line 8) prints value of ‘staticVariable’ i. e 10 on the console.

Second statement(Line 9) is commented. Because, **directly** **you can’t use non-static member inside a static method. Because, non-static members are stored inside the object memory. You have to create objects to use them. You have to refer them through objects only.**

No statements left in staticMethod(). So, it leaves the stack memory.

**Step 4 :**

Control comes back to main() method. The next statement (Line 26) is also commented. **Because, You can’t refer non-static member through a class name.** In the next statement (Line 28), an object of class A type is created. Randomly, some memory is allocated to object. **All non-static members, ‘nonStaticVariable’ and ‘nonStaticMethod()’,  of class A are loaded into this object memory.** ‘nonStaticVariable’ is a global variable, so it is first initialized with default value 0. A reference variable of type class A  **‘a1’** is created in main() method. It points to this newly created object.

In the same manner, object ‘a2’ is also created (Line 29). In the next statement (Line 31), value of ‘nonStaticVariable’ of ‘a1’ i.e 0 is printed. In the next statement (Line 32), value of ‘staticVariable’ of class A i.e 10 is printed.

**You can refer a static member of a class through object of that class like in Line 32. Whenever you refer a static member through a object, compiller replaces object name with its class name like a1.staticVariable is treated as A.staticVariable by the compiler.**

In the next statement (Line 33), it calls ‘nonStaticMethod()’ of a1.

**Step 5 :**

‘nonStaticMethod()’ of a1 comes to the stack for execution. First statement (Line 14) prints value of  ‘staticVariable’ of class A i.e 10 on the console. Second statement (Line 15) prints the value of ‘nonStaticVariable’ of a1 i.e 0. There are no other statements left in ‘nonStaticMethod()’ , so it leaves the stack.

**Step 6 :**

Control comes back to Line 34 of main() method. It calls staticMethod() of class A. ‘staticMethod()’ enters the stack for execution. First statment (Line 8) prints value of  ‘staticVariable’  i.e 10 on the console. It leaves the memory after executing this statement.

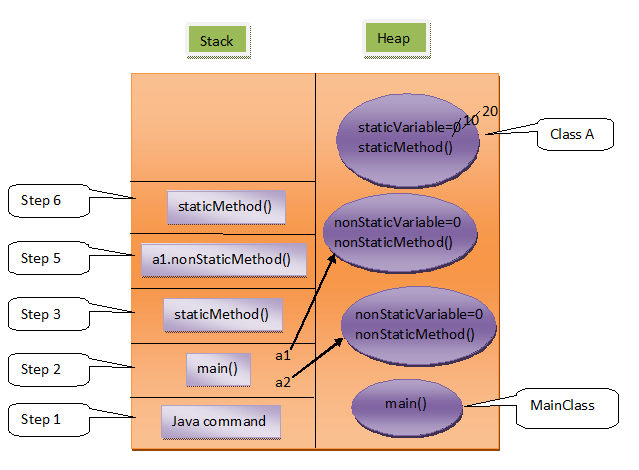
**Step 7 :**

Control comes back to the main() method. Line 36 prints value of ‘staticVariable’ i.e 10 on the console through object a2. In the next statement it changes value of ‘staticVariable’ to 20 through a1. In the next statement, again it prints the value of ‘staticVariable’ through a2. This time 20 is printed on the console.

**This means changes made to static components through one object is reflected in another object also. Because, the same copy of static components is available to all the objects of that class.**

As all statements are executed, first main() method then java command leaves the stack memory.

Diagramatic representation of memory allocation of above program looks like this,



**Output :**

10  
0  
10  
10  
0  
10  
10  
20

# 3. Constructor:

Basic Rules Need to Follow When Writing Constructors In Java:

Let’s discuss some basic rules need to follow when writing the constructors in java.

* Name of the constructor must be same as that of a class name. If you give another name it will give compile time error. If you give another name, it is neither a method because of no return type, nor constructor because name is different from class name.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | class A  {       A()       {           // Constructor of Class A       }       A1()       {           // Compile time error, It is neither a constructor nor a method       }  } |

* Constructors must not have a return type. If you keep return type for the constructor, it will be treated as another method.But compiler gives a warning saying that this method has a constructor name. That means, it is legal to have method name same as constructor name or same as class name but it is not recommended.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | class A  {       A()       {           // Constructor of Class A, not having any return type.       }         void A()       {           // constructor having a return type, It will be treated as method but with a warning.       }  } |

* Every class should have at least one constructor. If you don’t write constructor for your class, compiler will give default constructor. Default constructor is always public and it has no arguments (No-Arg Constructor).

|  |  |
| --- | --- |
| 1  2  3  4 | class A  {       // No Constructors written  } |

Compiler will treat the above code as,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class A  {      public A()      {          //   Constructor provided by the compiler.      }  } |

* Constructor can be declared as private. If you declare constructor as private, you can’t use it outside that class.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | class A  {       private A()       {            // Private Constructor       }         void methodOne()       {            //You can use private constructor inside the class            A a1 = new A();       }  }    class MainClass  {       public static void main(String[] args)       {            //You can't use private constructor ouside the class like this            // A a1 = new A();       }  } |

* One class can have more than one constructors. It is called **Constructor Overloading.**Through constructor overloading, you can have multiple ways to create objects.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | class A  {       A()       {          // First Constructor       }         A(int i)       {          // Second Constructor       }        A(int i, int j)      {         // Third Constructor      }  } |

you can create the objects to the above class in three ways like below,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class MainClass  {       public static void main(String[] args)       {            A a1 = new A();      //Using First Constructor            A a2 = new A(10);    // Using Second Constructor            A a3 = new A(10, 20);    // Using Third Constructor       }  } |

* Duplicate Constructors not allowed. If you keep duplicate constructors, you will get compile time error.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | class A  {       A(int i)       {           // Duplicate Constructor       }         A(int i)       {          // Duplicate Constructor       }  } |

* Multiple arguments of the constructors can’t have same name. If the two arguments have the same name, you will get compile time error.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class A  {       A(int i, int i)       {           // Duplicate Arguments Passed. It gives compile time error       }  } |

* Only public, protected and private keywords are allowed before a constructor name. If you keep any other keyword before a constructor name, it gives compile time error.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | class A  {       final A()       {           //Constructor can not be final       }       static A()       {           //Constructor can not be static       }       abstract A()       {          //Constructors can not be abstract       }  } |

* First statement in a constructor must be either super() or this(). If you put any other statements you will get compile time error.If you don’t include these statements, by default compiler will keep super() calling statement. super() – It is a calling statement to default constructor of super class. this()- it is a calling statement to constructor of the same class.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | class A  {       A()       {            //By Default, Compile will keep super() calling statement here.            System.out.println("First Constructor");       }       A(int i)       {            //Compiler will not keep any statement here            super();            System.out.println("Second Constructor");       }       A(int i, int j)       {            //Compiler will not keep any statement here            this();            System.out.println("Third Constructor");       }       A(int i, int j, int k)       {            System.out.println("Fourth Constructor");            // super(); It will give error if you keep super() here       }  } |

* Recursive constructor calling is not allowed.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | class A  {       A()       {            this();            // It gives compile time error       }  } |

* No Cylic calling of constructors.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | class A  {       A()       {            this(10);            // It gives compile time error       }       A(int i)       {            this();            // It gives compile time error       }  } |

# 4. IIB – Instance Initialization Block:

IIB stands for **Instance Initialization Block**. As the name suggest this block is used to initialize **state of an object**. State of an object is indicated by instance variables or non-static variables. So, IIB is used to initialize instance variables or non-static variables.

This is a block with no name and enclosed within {}. The syntax for IIB is,

|  |  |
| --- | --- |
| 1  2  3 | {        //Set Of Statements, mostly initialization statements  } |

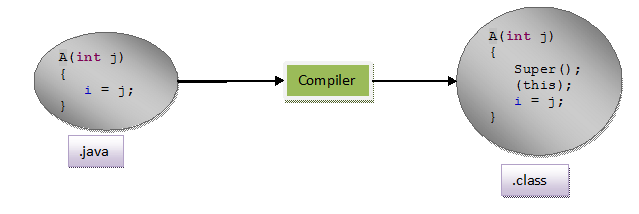
Consider this example,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | class A  {       int i;         {           i = 10;       }         A(int j)       {           i = j;       }  }    class MainClass  {       public static void main(String[] args)       {            A a = new A(50);            System.out.println(a.i);       }  } |

In the above example, **Class A**has one instance variable (int i), one IIB block (From Line 5 to Line 7) and one constructor.

We all know that [first statement of constructor](https://javaconceptoftheday.com/constructors-in-java/) is super() or this(). After executing first statement, IIB blocks are called. After executing IIB blocks, remaining statements are executed.

So, when the constructor is called while creating an object (Line 19), compiler will treat constructor code like this,



where (this) is a calling statement to IIB block.

You can keep any number of IIB blocks in a class. All blocks are called after super() in the constructor in the order they appear.

**Important Note :** IIB blocks will not be called from the constructor in which **this()** statement is written as a first statement. For example,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | class A  {       int i;         {            System.out.println("First IIB Block");       }         {            System.out.println("Second IIB Block");       }         A(int j)       {            this();            System.out.println("First Constructor");       }         A()       {            System.out.println("Second Constructor");       }  }    class MainClass  {       public static void main(String[] args)       {            A a = new A(50);       }  } |

For the above code, both the IIBs are executed only once. You are creating an object through First constructor. It has this() statement as first statement. It is nothing but the calling statement to second constructor. IIBs will not be executed in first constructor. They will be executed only in second constructor. If you execute above program, output will be,

First IIB Block  
Second IIB Block  
Second Constructor  
First Constructor,

IIBs can also be written as,

|  |  |
| --- | --- |
| 1  2  3  4 | class A  {       int i = 10;  } |

This is same as,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | class A  {       int i;         {            i = 10;       }  } |

# 5. Class Variables And Instance Variables In Java:

While methods represent the behavior of an object, variables represent the state of an object. Variables in Java are of two types – Class Variables and Instance Variables. Class variables (or static variables) are common to all instances of a class where as instance variables (or non-static variables) are specific to an object. Let’s discuss class variables and instance variables in Java and difference between them in detail.

### **Class Variables OR Static Variables In Java :**

1) Class variables, also called as static variables, are declared with static keyword.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | class StaticVariables  {      static int i;        //Static Variable        static String s;     //Static Variable  } |

2) Class variables are common to all instances of a class i.e these variables will be shared by all objects of a class. Hence, changes made to these variables through one object will reflect in all objects.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46 | class ClassVariables  {      static int i = 10;        //Static Variable        static String s = "STATIC";     //Static Variable  }    public class MainClass  {      public static void main(String[] args)      {          ClassVariables obj1 = new ClassVariables();            ClassVariables obj2 = new ClassVariables();            //accessing class variables through obj1            System.out.println(obj1.i);      //Output : 10            System.out.println(obj1.s);      //Output : STATIC            //accessing class variables through obj2            System.out.println(obj2.i);      //Output : 10            System.out.println(obj2.s);      //Output : STATIC            //Making changes to class variables through obj2            obj2.i = 20;            obj2.s = "STATIC - STATIC";            //accessing class variables through obj1            System.out.println(obj1.i);      //Output : 20            System.out.println(obj1.s);      //Output : STATIC - STATIC            //accessing class variables through obj2            System.out.println(obj2.i);      //Output : 20            System.out.println(obj2.s);      //Output : STATIC - STATIC      }  } |

3) Class variables can be accessed through class name as well as object reference.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | class A  {      static int i = 100;        //Class Variable  }    public class MainClass  {      public static void main(String[] args)      {          //Accessing class variable through class name            System.out.println(A.i);            A a = new A();            //Accessing class variable through object reference            System.out.println(a.i);      }  } |

### **Instance Variables OR Non-Static Variables In Java :**

1) Instance variables, also called as non-static variables, are declared without static keyword.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | class InstanceVariables  {      int i;        //Instance Variable        String s;     //Instance Variable  } |

2) Instance variables are not common to all instances of a class. Each object will have it’s own copy of instance variables. Hence, changes made to instance variables through one object will not reflect in another object.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46 | class InstanceVariables  {      int i = 10;        //Instance Variable        String s = "NON-STATIC";     //Instance Variable  }    public class MainClass  {      public static void main(String[] args)      {          InstanceVariables obj1 = new InstanceVariables();            InstanceVariables obj2 = new InstanceVariables();            //obj1 instance variables            System.out.println(obj1.i);       //Output : 10            System.out.println(obj1.s);       //Output : NON-STATIC            //obj2 instance variables            System.out.println(obj2.i);       //Output : 10            System.out.println(obj2.s);       //Output : NON-STATIC            //changing obj1 instance variables            obj1.i = 20;            obj1.s = "INSTANCE VARIABLE";            //obj1 instance variables            System.out.println(obj1.i);       //Output : 20            System.out.println(obj1.s);       //Output : INSTANCE VARIABLE            //obj2 instance variables            System.out.println(obj2.i);       //Output : 10            System.out.println(obj2.s);       //Output : NON-STATIC      }  } |

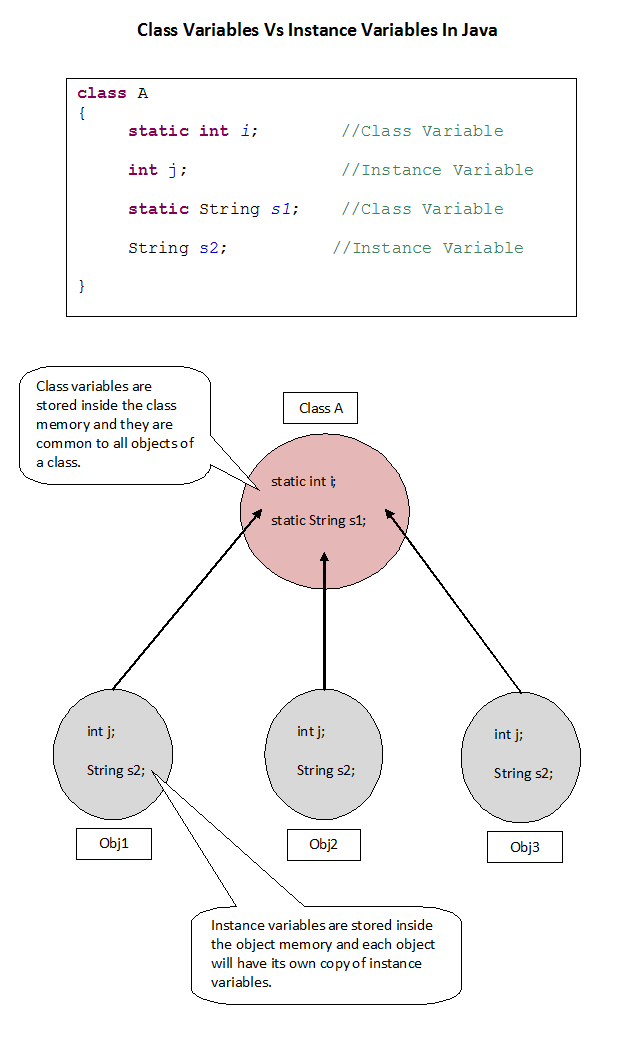
3) Instance variables can be accessed only through object reference.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | class A  {      int i = 100;        //Instance Variable  }    public class MainClass  {      public static void main(String[] args)      {          A a = new A();            //Accessing instance variable through object reference            System.out.println(a.i);            //You can't access instance variable through class name, you will get compile time error            //System.out.println(A.i);      }  } |

### **Differences Between Class Variables And Instance Variables In Java :**

|  |  |
| --- | --- |
| **Class Variables** | **Instance Variables** |
| Class variables are declared with static keyword. | Instance variables are declared without static keyword. |
| Class variables are common to all instances of a class. These variables are shared between the objects of a class. | Instance variables are not shared between the objects of a class. Each instance will have it’s own copy of instance variables. |
| As class variables are common to all objects of a class, changes made to these variables through one object will reflect in another. | As each object will have its own copy of instance variables, changes made to these variables through one object will not reflect in another object. |
| Class variables can be accessed using either class name or object reference. | Instance variables can be accessed only through object reference. |

### **Class Variables Vs Instance Variables In Java :**



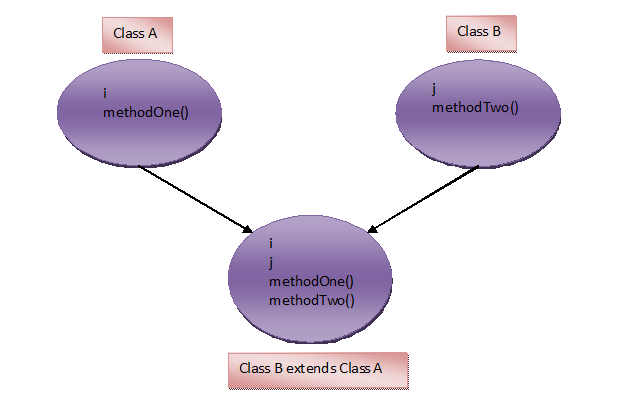
# ****6.**** Inheritance In Java:

**Inheritance** in java is used to add additional functionalities to the existing class. Inheritance is used to extend the present class by adding some more properties to it. Inheritance is used to reuse the present tried and tested code so that you may not have to write them and compile them again.

Inheritance in java is implemented by using **extend** keyword like below,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | class A  {      int i;      void methodOne()      {          System.out.println("From methodOne");      }  }    class B extends A  {      int j;      void methodTwo()      {          System.out.println("From methodTwo");      }  } |

Here, **Class A** has two members – **‘i’** and **‘methodOne()’**. **Class B** also has two members – **‘j’** and **‘methodTwo()’**. **Class B** is extending **Class A**. Therefore, members of **class A** are inherited to **Class B**. Now, **Class B** will have two additional members inherited from **class A** along with its members. Have a look at the below diagram,



Here, **Class A** is called **super class** and **Class B** is called **sub class**. Here, you can analyse the inheritance like this – we are extending **class A** by adding additional properties to it through **Class B** or We are reusing properties of **Class A** in **Class B**.

## **Points-To-Remember About Inheritance In Java :**

Here are some points regarding inheritance in java.

* Constructors, SIB – Static Initialization Block and IIB – Instance Initialization Block of super class will not be inheriting to its sub class. But they are executed while creating an object to sub class. For example,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | class A  {      int i;        static      {          System.out.println("Class A SIB");      }        {          System.out.println("Class A IIB");      }        A()      {          System.out.println("Class A Constructor");      }  }    class B extends A  {      int j;  }    class MainClass  {      public static void main(String[] args)      {          B b = new B();      }  } |

In the above example, **Class B** is extending **Class A**. In the **MainClass**, We are creating an object to **Class B**. While creating this object, SIB, IIB and constructor of **Class A** are also executed. The output of the above program will be,

**Output :**  
Class A SIB  
Class A IIB  
Class A Constructor

* Static members of super class are inheriting to sub class as static members and non-static members are inheriting as non-static members only.
* Try to compile the following program,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class A  {      int i;        A(int i)      {          System.out.println("Class A Constructor");      }  }    class B extends A  {      int j;  } |

You will get a compile time error saying implicit default constructor A() is undefined for Class A. Compiler will force you to write constructor in Class B. Because, we are not defining constructor for Class B. So, compiler will be providing default constructor. In that default constructor, first statement is super() – it is a calling statement to default constructor of Class A. But it is not defined in Class A. Therefore you will get a compile time error. To avoid this error, write the constructor for sub class. From that constructor call super class constructor explicitly. See the below code,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | class A  {      int i;        A(int i)      {          System.out.println("Class A Constructor");      }  }    class B extends A  {      int j;        public B()      {          super(10);     //Explicitly Calling Class A constructor          System.out.println("Class B Constructor");      }  } |

* By default, every class is a sub class of **java.lang.Object** class. So, every class in java has properties inherited from Object class. Look at the below code,

|  |  |
| --- | --- |
| 1  2  3  4 | class A  {       //some satements  } |

Compiler will treat the above code as,

|  |  |
| --- | --- |
| 1  2  3  4 | class A extends Object  {       //some satements  } |

* Any class can not extend itself i.e

|  |  |
| --- | --- |
| 1  2  3  4 | class A extends A  {       //It gives compile time error  } |

* We can call super class constructor explicitly through **super()** calling statement from sub class constructor and we can call other constructors of the same class through **this()** calling statement but, we can’t call sub class constructor from super class constructor.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | class A  {       A()       {            //B();      There is no statement in java to call subclass constructor            System.out.println("Class A Constructor");       }  }    class B extends A  {       B()       {            super();        // calling statement to super class constructor            System.out.println("Class B Constructor");       }  } |

## **Effect of private, default, protected and public keyword on inheritance in java:**

**private :**Private members can not be inherited to sub class.

**Default** : Default members can be inherited to sub class within package.

**protected** : protected members can be inherited to any sub class but usage of protected member is limited within package.

**public** : public members are inherited to all sub classes.

Let’s discuss this concept with example,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67 | package com1;    public class A  {      private int i;      int j;      protected int k;      public int m;  }    class B extends A  {      void methodOfClassB()      {          //System.out.println(i);        Private member can not be inherited          System.out.println(j);           //Default member can be inherited within package          System.out.println(k);         //protected member can be inherited to any subclass          System.out.println(m);       //public member can be inherited to all sub classes      }  }    class C extends B  {      void methodOfClassC()      {          System.out.println(j);     //Default member can be inherited within package          System.out.println(k);    //protected member can be inherited to any subclass          System.out.println(m);    //public member can be inherited to any subclass            B b = new B();          System.out.println(b.j);   //Default member can be used within package          System.out.println(b.k);   //Protected member can be used anywhere in the package          System.out.println(b.m);  //Public member can be used anywhere      }  }    package com2;  import com1.A;    public class D extends A  {      void methodOfClassD()      {          //System.out.println(j);   Default members can not be inherited outside package          System.out.println(k);  //Protected member can be inherited to any subclass          System.out.println(m);  //public member is always inherited to any subclass            A a = new A();          //System.out.println(a.i);   private member not visible outside the class          //System.out.println(a.j);   Default members are not visible outside package          //System.out.println(a.k);   Protected member can not be used outside the package.          System.out.println(a.m);     //public member can be used anywhere      }  }    class E extends D  {      void methodOfClassE()      {          System.out.println(k);     //Protected member can be inherited to any subclass          System.out.println(m);     //public member is always inherited            D d = new D();          //System.out.println(d.k);     Protected member can not be used outside the package.          System.out.println(d.m);    //public member can be used anywhere      }  } |

## **Types Of Inheritance In Java :inheritance in java**

**1) Single Inheritance :** One class is extended by only one class.

**2) Multilevel Inheritance :** One class is extended by a class and that class is extended by another class thus forming chain of inheritance.

**3) Hierarchical Inheritance :** One class is extended by many class.

**4) Hybrid Inheritance :** It is a combination of above types of inheritance.

There exist one more type of inheritance – **Multiple Inheritance.**

**5) Multiple Inheritance :** One class extends more than one class.

But, **Multiple Inheritance is not supported in java.** To avoid the ambiguity, complexity and confusion, multiple inheritance is not supported in java. Look at the below example,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | class A  {      void methodOne()      {          System.out.println("From methodOfClassA");      }  }    class B  {      void methodOne()      {          System.out.println("From methodOfClassB");      }  }    class C extends A, B (If it is supported)  {      //Both the methods with same name are inherited to Class B      //This causes ambiguity and confusion. Therefore,      //Multiple Inheritance is not supported in java  } |

In the above example, **Class A** also has **methodOne()** and **Class B** also has **methodOne()**. **Class C** is extending both the classes. So both the methods with same name are inheriting to **Class C**. It causes confusion and ambiguity for which method to use. Therefore, To avoid this, multiple inheritance is not supported in java.

# 7. super and this keywords in java:

## **super keyword :**

super keyword is used to access super class members inside the sub class. Using super keyword, we can access super class methods, super class fields and super class constructors in the sub classes.

For example, following program calls super class constructor, field and method from sub class.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | class SuperClass  {      int i;    //Field        SuperClass(int j)      {          System.out.println("Super Class Constructor");      }        void methodOfSuperClass()     //method      {          System.out.println("From method of super class");      }  }    class SubClass extends SuperClass  {      SubClass()      {          super(10);          //Calling statement to super class constructor      }        void methodOfSubClass()      {          System.out.println(super.i);  //super class field is accessed          super.methodOfSuperClass();  // super class method is called          System.out.println("From method of sub class");      }  } |

super class constructor is called by **super()** calling statement.You can’t use super() calling statement outside the constructor. By default, super() calling statement is the first statement in any constructor. You can go through the constructors rules [here](https://javaconceptoftheday.com/constructors-in-java/).

## **When to Use super keyword?.**

If you want same implementation as that of super class method in the sub class, but want to add some more extra statements to it, in such cases, super keyword will be very useful. First call the super class method using super keyword and after it add extra statements according to requirements in the sub class method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | class SuperClass  {      void methodOfSuperClass()      {          //Some task      }  }    class SubClass extends SuperClass  {      void methodOfSubClass()      {          super.methodOfSuperClass();  // super class method is called            //add some other extra statements fulfilling the requirements      }        //you can implement same task by overriding super class method also        void methodOfSuperClass()      {          //super class method is overrided.            super.methodOfSuperClass();            //add some other extra statements fulfilling the requirements        }  } |

## **this keyword :**

this keyword is used to access other members of the same class. Using this keyword, you can access methods, fields and constructors of the same class within the class. this refers to current instance of the class.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | class AnyClass  {      int i;        AnyClass()      {          System.out.println("First Constructor");      }        AnyClass(int j)      {          this();    //calling statement to First Constructor          System.out.println("Second Constructor");      }        void methodOne()      {          System.out.println("From method one");      }        void methodTwo()      {          System.out.println(this.i);  //Accessing same class field          this.methodOne();      //Accessing same class method      }  } |

this() is the calling statement to same class constructor. It must be used within constructor only. If it is used, it must be the first statement in the constructor. You can go through the constructors rules [here](https://javaconceptoftheday.com/constructors-in-java/).

## **Important Notes :**

* You can’t use super and this keywords in a static method and in a static initialization block even though you are referring static members.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | class SuperClassOne  {      int i;      //Non-Static member        static void methodOne()      {          //static method          System.out.println("From Super Class");      }  }    class SubClassOne extends SuperClassOne  {      static      {          System.out.println(super.i);          this.methodTwo();            //Above statements give compile time error          //You can't use super and this keywords inside SIB      }        static void methodTwo()      {          super.methodOne();          this.methodOne();            //These also give compile time error          //You can't use super and this keywords inside static method          //even though you are accessing static methods      }  } |

* You should call super() and this() calling statements inside the constructors only and they must be first statement in the constructors.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | class SuperClassOne  {      void methodOne()      {          System.out.println("From Super Class");      }  }    class SubClassOne extends SuperClassOne  {      public SubClassOne()      {          System.out.println("constructors");          super();          //compile time error          //super() calling statement must be first statement in constructor      }      void methodTwo()      {          super();          this();          //compile time error          //you should call super() and this()          //calling statements only in constructors.      }  } |

# 8. Abstraction In Java:

In the computer science perspective, Abstraction is the process of separating ideas from their action. (Courtesy : [Wiki](http://en.wikipedia.org/wiki/Abstraction_(computer_science))).

Yes, In the computer science, Abstraction is used to separate ideas from their implementation. Abstraction in java is used to define only ideas in one class so that the idea can be implemented by its sub classes according to their requirements. For example,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | abstract class Animal  {      abstract void soundOfAnimal();  // It is just an idea  }    class Cat extends Animal  {      void soundOfAnimal()      {          System.out.println("Meoh");          //Implementation of the idea according to requirements of sub class      }  }    class Dog extends Animal  {      void soundOfAnimal()      {          System.out.println("Bow Bow");          //Implementation of the idea according to requirements of sub class      }  } |

Abstraction in java is implemented using Abstract classes and interfaces.

Today we will discuss only Abstract Classes. In the next concept, we will discuss about interfaces.

**Abstract Classes :**

Abstract classes contain abstract methods (you can refer them as ideas) so that they can be implemented in sub classes according to their requirements. They are also called as incomplete classes as they have some unimplemented abstract methods(ideas).

Let’s discuss some rules need to follow while using abstract classes and abstract methods.

* Abstract classes and abstract methods are declared using ‘**abstract**‘ keyword. We can’t create objects to those classes which are declared as abstract. But, we can create objects to sub classes of abstract class, provided they must implement abstract methods.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | abstract class AbstractClass  {      abstract void abstractMethod();  }    class ConcreteClass extends AbstractClass  {      void abstractMethod()      {          System.out.println(&quot;Abstract Method Implemented&quot;);      }  }    public class Abstraction  {      public static void main(String[] args)      {          //AbstractClass A = new AbstractClass();  Can't create objects to Abstract class          ConcreteClass C = new ConcreteClass();          //ConcreteClass implements abstract method,          //so we can create object to ConcreteClass          AbstractClass A1 = C;          //ConcreteClass object is auto-upcasted to AbsractClass      }  } |

* The methods which are not implemented or which don’t have definitions must be declared with ‘abstract’ keyword and the class which contains it must be also declared as abstract.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | // It gives compile time error  class AbstractClass  {      void abstractMethod();  //This method must be declared as abstract or must be defined      abstract void abstractMethod();  //The Class must be also declared as abstract  }    //      \*\*\*\*\*   \*\*\*\*\*     \*\*\*\*\*   \*\*\*\*\*  // This is OK  abstract class AbstractClass  {      abstract void abstractMethod();  } |

* It is not compulsory that abstract class must have abstract methods. It may or may not have abstract methods. But the class which has at least one abstract method must be declared as abstract.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | abstract class AbstractClass  {      void methodOne()      {          //Concrete Method      }      //No Abstract methods but class is abstract  } |

* You can’t create objects to abstract class even though it does not contain any abstract methods.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | abstract class AbstractClass  {      void methodOne()      {          //Concrete Method      }        void methodTwo()      {          //Concrete Method      }  }    public class Abstraction  {      public static void main(String[] args)      {          AbstractClass a = new AbstractClass();  //Compile time error          //You can't create objects to AbstractClass          //even though it does not contain any abstract methods.      }  } |

* Abstract Class can be a combination of concrete and abstract methods.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | abstract class AbstractClass  {      void methodOne()      {          //Concrete Method      }        void methodTwo()      {          //Concrete Method      }        abstract void methodThree();  //Abstract Method        abstract void methodFour();  //Abstract Method  } |

* Any class extending an abstract class must implement all abstract methods. If it does not implement, it must be declared as abstract.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | abstract class AbstractClass  {      abstract void abstractMethodOne();  //Abstract Method        abstract void abstractMethodTwo();  //Abstract Method  }    class ConcreteClass extends AbstractClass  {      void abstractMethodOne()      {          //abstractMethodOne() is implemented      }        //This class must implement second abstract method also,      //otherwise, this class has to be declared as abstract        void abstractMethodTwo()      {          //abstractMethodTwo() is also implemented.          //No need to declare this class as abstract      }  } |

* Inside abstract class, we can keep any number of constructors. If you are not keeping any constructors, then compiler will keep default constructor.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | abstract class AbstractClass  {      AbstractClass()      {          //First Constructor      }        AbstractClass(int i)      {          //Second Constructor      }        abstract void abstractMethodOne();  //Abstract Method  } |

* Abstract methods can not be private. Because, abstract methods must be implemented somehow in the sub classes. If you declare them as private, then you can’t use them outside the class.

|  |  |
| --- | --- |
| 1  2  3  4  5 | abstract class AbstractClass  {      private abstract void abstractMethodOne();      //Compile time error, abstract method can not be private.  } |

* Constructors and fields can not be declared as abstract.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | abstract class AbstractClass  {      abstract int i;      //Compile time error, field can not be abstract        abstract AbstractClass()      {          //Compile time error, constructor can not be abstract      }  } |

* Abstract methods can not be static.

|  |  |
| --- | --- |
| 1  2  3  4  5 | abstract class AbstractClass  {      static abstract void abstractMethod();      //Compile time error, abstract methods can not be static  } |

# 9. Interface:

interfaces, another way of implementing abstraction in java.

Interfaces in java are very much similar to abstract classes but interfaces contain only abstract methods (you can refer to them as only ideas). Abstract classes may contain both abstract methods as well as concrete methods. But interfaces must contain only abstract methods. Concrete methods are not allowed in interfaces. Therefore, Interfaces show 100% abstractness.

Let’s discuss some of the points regarding Interfaces.

* Interfaces are declared with keyword ‘**interface**‘ and interfaces are implemented by the class using ‘**implements**‘ keyword.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | interface InterfaceClass  {      //Some Abstract methods  }    class AnyClass implements InterfaceClass  {      //Use 'implements' while implementing Interfaces      //Don't use 'extends'  } |

* Interfaces should contain only abstract methods. Interfaces should not contain a single concrete method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | interface InterfaceClass  {      abstract void abstractMethodOne();  //abstract method        abstract void abstractMethodTwo();  //abstract method        void concreteMethod()      {          //Compile Time Error.          //Concrete Methods are not allowed in interface      }  } |

* Interface can have two types of members.  **1) Fields     2) Abstract Methods.**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | interface InterfaceClass  {      int i = 0;      //Field        abstract void abstractMethodOne();  //abstract method        abstract void abstractMethodTwo();  //abstract method  } |

* By default, Every field of an interface is public, static and final (we will discuss about final keyword Later). You can’t use any other modifiers other than these three for a field of an interface.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | interface InterfaceClass  {      int i = 0;      //By default, field is public, static and final        //Following statements give compile time errors        private double d = 10;      protected long l = 15;        //You can't use any other modifiers other than public, static and final  } |

* You can’t change the value of a field once they are initialized. Because they are static and final. Therefore, sometimes fields are called as **Constants**. (We will discuss this feature in detail while covering ‘final’ keyword)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | interface InterfaceClass  {      int i = 0;  }    class AnyClass implements InterfaceClass  {      void methodOne()      {          //Following statement gives compile time error.            InterfaceClass.i = 10;            //final field can not be re-assigned      }  } |

* By default, All methods of an interface are public and abstract.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | interface InterfaceClass  {      void abstractMethodOne();  //Abstract method        void abstractMethodTwo();  //Abstract Method            //No need to use abstract keyword,          //by default methods are public and abstract  } |

* Like classes, for every interface .class file will be generated after compilation.
* While implementing any interface methods inside a class, that method must be declared as public. Because, according to [method overriding](https://javaconceptoftheday.com/method-overriding-java/) rule, you can’t reduce visibility of super class method. By default, every member of an interface is public and while implementing you should not reduce this visibility.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | interface InterfaceClass  {      void methodOne();  }    class AnyClass implements InterfaceClass  {      void methodOne()      {          //It gives compile time error.          //Interface methods must be implemented as public      }  } |

* By default, Interface itself is not public but by default interface itself is abstract like below,

|  |  |
| --- | --- |
| 1  2  3  4  5 | abstract interface InterfaceClass  {      //By default interface is abstract      //No need to use abstract keyword  } |

* [SIB](https://javaconceptoftheday.com/static-members-java/) – Static Initialization Block and [IIB](https://javaconceptoftheday.com/instance-initialization-block-in-java/) – Instance Initialization Block are not allowed in interfaces.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | interface InterfaceClassOne  {      static      {          //compile time error          //SIB's are not allowed in interfaces      }        {          //Here also compile time error.          //IIB's are not allowed in interfaces      }        void methodOne();  //abstract method  } |

* As we all know that, any class in java can not extend more than one class. But class can implement more than one interfaces. This is how **multiple inheritance** is implemented in java.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | interface InterfaceClassOne  {      void methodOne();  }    interface InterfaceClassTwo  {      void methodTwo();  }    class AnyClass implements InterfaceClassOne, InterfaceClassTwo  {      public void methodOne()      {          //method of first interface is implemented      }        //method of Second interface must also be implemented.      //Otherwise, you have to declare this class as abstract.        public void methodTwo()      {          //Now, method of Second interface is also implemented.          //No need to declare this class as abstract      }  } |

# 10. Does An Interface Extend Object Class In Java.?

## **Does an interface extend Object class in java.?**

You may have come across this question while reading about interfaces in java. You may also know that only classes in java are inherited from java.lang.Object class. Interfaces in java don’t inherit from Object class. They don’t have default parent like classes in java. But, following two cases may surprise you.

**Case 1 :**

If an interface does not extend Object class, then why we can call methods of Object class on interface variable like below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | interface A  {    }    class InterfaceAndObjectClass  {      public static void main(String[] args)      {          A a = null;            a.equals(null);            a.hashCode();            a.toString();      }  } |

**Case 2 :**

If an interface does not extend Object class, then why the methods of Object class are visible in interface.?

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | interface A  {      @Override      public boolean equals(Object obj);        @Override      public int hashCode();        @Override      public String toString();  } |

**Explanation :**

This is because, for every public method in Object class, there is an implicit abstract and public method declared in every interface which does not have direct super interfaces. This is the standard Java Language Specification which states like this,

“If an interface has no direct superinterfaces, then the interface implicitly declares a public abstract member method m with signature s, return type r, and throws clause tcorresponding to each public instance method m with signature s, return type r, and throws clause t declared in Object, unless a method with the same signature, same return type, and a compatible throws clause is explicitly declared by the interface.”

# 11. Final Keyword in java:

# 10 Points Every Java Programmer Should Know About final keyword in java

A **final keyword in java** can be used with a class, with a variable and with a method. final keyword restricts the further modification. When you use final keyword with an entity (class or variable or method), it gets the meaning that entity is complete and can not be modified further.

In this post, we will discuss some about 10 important points about final keyword which every java programmer should know. Let’s start with some simple basic things about final keyword in java.

## **final class in java :**

We can’t create a subclass to the class or we can’t extend a class or we can’t modify a class which is declared as **final**.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | final class FinalClass  {      //some statements  }    class SubClass extends FinalClass  {      //compile time error      //Can't create sub class to the final class  } |

## **final method in java :**

We can’t override a method or we can’t modify a method in the sub class which is declared as **final** in the super class.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | class SuperClass  {      final void methodOne()      {          //some statements      }  }    class SubClass extends SuperClass  {      @Override      void methodOne()      {          //Compile time error          //can not override final method      }  } |

## **final variable in java :**

The value of a final variable can not be changed in the whole execution once it got initialized.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | class AnyClass  {      final int i = 10;        void methodOne()      {          i = 20;     //compile time error          //final field can not be re-assigned      }  } |

## **10 Points Every Java Programmer Should Know About final Keyword In Java :**

**1)** Any class or any method can be either **abstract or final** but not both. abstract and final are totally opposite. Because, abstract class or abstract method must be implemented or modified in the sub classes but final does not allow this. This creates an ambiguity.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | //The following class gives compile time error    final abstract class AnyClass  {      //Any class can not be final and abstract        final abstract void methodOne();      //method can not be final and abstract at a time  } |

**2)** final method can be overloaded and that overloaded method can be overridden in the sub class.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | class SuperClass  {      final void methodOne()      {          //final method      }        void methodOne(int i)      {          //final method can be overloaded      }  }    class SubClass extends SuperClass  {      @Override      void methodOne(int i)      {          //Overloaded method can be overridden      }  } |

**3)** final variable can not be re-initialized but final variable can be used to initialize other variables.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | class AnyClassOne  {      final int i = 10;        void methodOne()      {          i++;          //above statement gives Compile time error.          //value of final variable can not be changed            int j = i;        //final variable can be used to initialize other variables.            System.out.println(i);  //final variable can be used      }  } |

**4)** When an array reference variable is declared as final, only variable itself is final but not the array elements.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class UseOfFinalKeyword  {      public static void main(String[] args)      {          final int X[] = new int[10];     //final array variable            X[2] = 10;          X[2] = 20;     //Array element can be re-assigned            X = new int[30];  //compile time error          //can't re-assign new array object to final array variable      }  } |

**5)** When a reference variable is declared as final, you can’t re-assign a new object to it once it is referring to an object. But, you can change the state of an object to which final reference variable is referring.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | class A  {      int i = 10;  }    public class UseOfFinalKeyword  {      public static void main(String[] args)      {          final A a = new A();  //final reference variable            a.i = 50;          //you can change the state of an object to which final reference variable is pointing            a = new A();  //compile time error            //you can't re-assign a new object to final reference variable      }  } |

**6)** Static variables, non-static variables and local variables all can be final. once the final variables are initialized, even you can’t re-assign the same value.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | class A  {      static final int i = 10;   //final static variable      final int j = 20;          //final non-static variable        void methodOne(final int k)      {          //k is final local variable          k = 20;   //compile time error      }  }    public class UseOfFinalKeyword  {      public static void main(String[] args)      {          A a = new ();            a.i = 10;     //Compile time error          a.j = 20;     //even you can't assign same value to final variables            a.methodOne(20);      }  } |

**7)** If the global variables are not initialized explicitly, they get default value at the time of object creation. But final global variables don’t get default value and they must be explicitly initialized at the time of object creation. Uninitialized final field is called **Blank Final Field**.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | class A  {      int i;   //Non-final global variable, no need to initialize them        final int j;         //Blank Final Field        A()      {          j=20;            //final global variable must get a value at the time of object creation.      }  }    public class UseOfFinalKeyword  {      public static void main(String[] args)      {          A a = new A();      }  } |

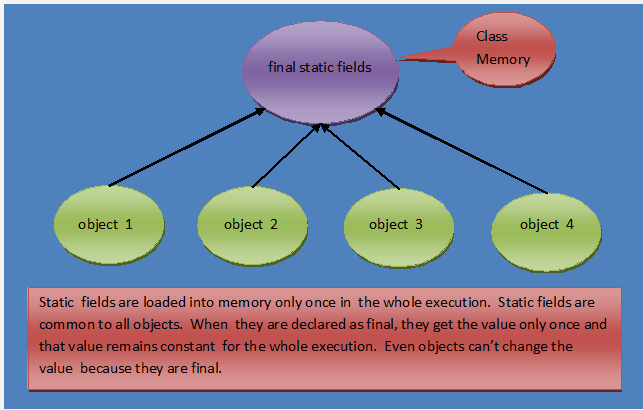
**8)** final non-static global variable must be initialized at the time of declaration or in all constructors or in any one of IIBs – Instance Initialization Blocks.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | class A  {      final int i;  //Final non-static global variable may be initialized here  OR        //may be initialized in any one of IIB's,      // because while object creation, all IIBs are called.  OR        {          i = 30;      }        {          //i = 40;      }        //must be initialized in all constructors.      //because while object creation, only one constructor is called        A()      {          //i=20;      }        A(int j)      {         // i=j;      }        A(int j, int k)      {         // i = 50;      }  } |

**9)** final static global variable must be initialized at the time of declaration or in any one of SIBs – Static Initialization Blocks. (final static global variable can’t be initialized in constructors)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | class A  {      static final int i;   //final static global variable may be initialized here OR        //may be initialized in any one of SIBs.        static      {          i = 30;      }        static      {          //i = 40;      }        //final static global variable can not be initialized in constructors        A()      {          //i=20;      }        A(int j)      {          //i=j;      }        A(int j, int k)      {          //i = 50;      }  } |

**10)** The global variable which is declared as final and static remains unchanged for the whole execution. Because, Static members are stored in the class memory and they are loaded only once in the whole execution. They are common to all objects of the class. If you declare static variables as final, any of the objects can’t change their value as it is final. Therefore, variables declared as final and static are sometimes referred to as **Constants**. All fields of interfaces are referred as constants, because they are final and static by default.



# 12. Nested Class in Java:

Nested classes in java can be defined as classes within the class. i.e A class can be a member of another class. For example,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class OuterClass  {      int i;       //Field as a member        void methodOne()      {          //method as a member      }        class NestedClass      {          //class as a member      }  } |

There are 2 types of Nested Classes.

1. Static Nested Classes
2. Non-Static Nested Classes or Inner Classes

Today, we will discuss about Static Nested Classes.

## **Static Nested Classes In Java :**

If nested class is declared as static, then that nested class is called as **static nested class**.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class OuterClass  {      int i;       //Field as a member        void methodOne()      {          //method as a member      }        static class NestedClass      {          //class as a member which is declared as static      }  } |

Let’s discuss some interesting points about Static Nested Classes.

* Static nested classes can contain both static and non-static members.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | class OuterClass  {      //Some members of OuterClass        static class NestedClass      {          static int i;    //Static Field            int j;      //Non-static Field            void methodOne()          {              //Non-static method          }            static void methodTwo()          {              //Static Method          }      }  } |

* We can access only static members of outer class inside a static nested class. We can’t access non-static members of outer class inside a static nested class.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | class OuterClass  {      static int i;  //static field of OuterClass      int j;         //Non-static field of OuterClass        void methodOne()      {          //Non-static method of OuterClass      }        static void methodTwo()      {          //static method of OuterClass      }        static class NestedClass      {          void methodOfInnerClass()          {              System.out.println(i);    //static field can be accessed                System.out.println(j);    //This gives Compile time error                //can't access non-static field                methodTwo();   //can access static method                methodOne();   //This gives Compile time error                //can't access non-static method          }      }  } |

* We have seen that static methods can’t be abstract but static nested classes can be abstract.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class OuterClass  {      //static and abstract inner class        abstract static class NestedClass      {          abstract void methodOne();  //abstract method of NestedClass            void methodTwo()          {              //concrete method of NestedClass          }      }  } |

* Static nested class can be final.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | class OuterClass  {      //final and static nested class        final static class NestedClass      {          void methodOne()          {              //concrete method of NestedClass          }      }  } |

* Below example shows how to refer Objects of the static nested class.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47 | class OuterClass  {      int i = 10;   //Non-static Field of OuterClass        static void methodOne()      {          System.out.println("Static method of OuterClass");      }        static class NestedClassOne      {          int i = 20;   //Non-static Field of NestedClassOne            static void methodOne()          {              System.out.println("Static method of NestedClassOne");          }      }        static class NestedClassTwo      {          int i = 30;    //Non-static Field of NestedClassTwo            static void methodOne()          {              System.out.println("static method of NestedClassTwo");          }      }  }    public class NestedClasses  {      public static void main(String[] args)      {          OuterClass.methodOne();      //static member can be accessed directly through class name.          OuterClass outer = new OuterClass();          System.out.println(outer.i);  //Instance member must be accessed through object reference            OuterClass.NestedClassOne.methodOne();  //static member can be accessed directly through class name.          OuterClass.NestedClassOne nestedOne = new OuterClass.NestedClassOne();          System.out.println(nestedOne.i);     //Instance member must be accessed through object reference            OuterClass.NestedClassTwo.methodOne();  //static member can be accessed directly through class name.          OuterClass.NestedClassTwo nestedTwo = new OuterClass.NestedClassTwo();          System.out.println(nestedTwo.i);     //Instance member must be accessed through object reference      }  } |

* Constructors and methods of nested classes can be overloaded.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | class OuterClass  {      static class NestedClass      {          NestedClass()          {              //First constructor          }            NestedClass(int i)          {              //Second Constructor          }            NestedClass(int i, int j)          {              //Third Constructor          }            void methodOne()          {              //Overloaded method          }            void methodOne(int i)          {              //Overloaded method          }            void methodOne(int i, int j)          {              //Overloaded method          }      }  } |

* Static Nested Classes can be chained. i.e Nested class may contain another nested class and that nested class may contain another nested class and so on.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | class OuterClass  {      static class NestedClass      {          static class NestedClassOne          {              static class NestedClassTwo              {                  static class NestedclassThree                  {                      static void methodOne()                      {                          System.out.println("Chain Of Nested Classes");                      }                  }              }          }      }  }    public class NestedClasses  {      public static void main(String[] args)      {          OuterClass.NestedClass.NestedClassOne.NestedClassTwo.NestedclassThree.methodOne();      }  } |

If you compile the above program, for each class, .class file will be generated. The generated .class files are –  OuterClass.class, OuterClass$NestedClass.class, OuterClass$NestedClass$NestedClassOne.class, OuterClass$NestedClass$NestedClassOne$NestedClassTwo.class, OuterClass$NestedClass$NestedClassOne$NestedClassTwo$NestedclassThree.class.

If you observe names of generated .class files, you will come to know that name contains name of outer class and nested classes seperated by $.

# 13. Non-static Nested classes or Inner Classes:

## **Non-Static Nested Classes In Java :**

Nested classes which are declared as non-static or nested classes which can be accessed only though instantiating it’s outer class are called non-static nested classes. Non-static nested classes are also called as **Inner Classes.**

They are 3 types of Inner Classes in java.

1. Member Inner Classes
2. Local Inner Classes
3. Anonymous Inner classes

Today we will discuss about Member Inner Classes.

## **Member Inner Classes :**

Member Inner Classes are non-static nested classes which are declared as non-static members of outer class.

Let’s discuss important observations about member inner classes.

* Member inner classes must contain only non-static members. Static members are not allowed inside member inner classes.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | class OuterClass  {      //Member Inner Class : Class As a Non-Static Member      class InnerClass      {          int i;    //can contain non-static field            static int j = 10;    //It gives compile time error            //Should not contain static field            void methodOne()          {              //can have non-static method          }            static void methodTwo()          {              //Compile time error              //should not contain static method          }      }  } |

* But, here is the interesting point. You can declare a static field inside a member inner class if the field is final. And such field must be initialized at the time of declaration only. Remember, this rule is only for the fields not for the methods.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | class OuterClass  {      class InnerClass      {          int i;    //can contain non-static field            static final int j = 10;   //can contain static and final field            //it must be initialized at the time of declaration.      }  } |

* Member inner class may contain any number of IIB’s but should not contain any SIB’s.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | class OuterClass  {      class InnerClass      {          int i;            {              System.out.println("First IIB");          }            {              System.out.println("Second IIB");          }            static          {              //compile time error              //Member Inner Class should not contain SIB          }      }  } |

* We can access both static and non-static members of outer class inside a member inner class.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | class OuterClass  {      int i;     //Non-static field of OuterClass        static int j;    //Static field of OuterClass        void methodOne()      {          System.out.println("Non-Static Method Of OuterClass");      }        static void methodTwo()      {          System.out.println("Static Method Of OuterClass");      }        class InnerClass      {          void methodOfInnerClass()          {              System.out.println(i); //can use non-static field of OuterClass                System.out.println(j);   //can use static field of OuterClass                methodOne();    //can call non-static method of OuterClass                methodTwo();    //can call static method of OuterClass          }      }  } |

* Below example shows how to instantiate member inner class and how to access it’s members.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | class OuterClass  {      class InnerClass      {          int i;     //Non-static field of InnerClass            static final int j = 10;  //static and final field of InnerClass            void methodOne()          {              System.out.println("Non-static method of InnerClass");          }      }  }    public class InnerClasses  {      public static void main(String args[])      {          OuterClass outer = new OuterClass();  //creating an instance of OuterClass            OuterClass.InnerClass inner = outer.new InnerClass();  //creating an instance of InnerClass            System.out.println(inner.i);    //accessing non-static field of InnerClass            System.out.println(OuterClass.InnerClass.j);    //static field can be accessed directly through class name            inner.methodOne();     //accessing non-static method of InnerClass      }  } |

* All members of outer class are accessible inside member inner class and all members of member inner class are accessible inside the outer class irrespective of their visibility.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47 | class OuterClass  {      private int i;   //private field of OuterClass        int j;           //Default field of OuterClass        protected int k; //protected field of OuterClass        public int m;    //public field of OuterClass        void methodOfOuterClass()      {          InnerClass inner = new InnerClass(); //creating instance of InnerClass            System.out.println(inner.a);  //accessing private field of InnerClass            System.out.println(inner.b);  //accessing default field of InnerClass            System.out.println(inner.c);  //accessing protected field of InnerClass            System.out.println(inner.d);  //accessing public field of InnerClass      }        class InnerClass      {          private int a;    //private field of InnerClass            int b;            //Default field of InnerClass            protected int c;  //protected field of InnerClass            public int d;     //public field of InnerClass            void methodOfInnerClass()          {              OuterClass outer = new OuterClass(); //creating an instance of OuterClass                System.out.println(outer.i);    //accessing private field of OuterClass                System.out.println(outer.j);    //accessing default field of OuterClass                System.out.println(outer.k);    //accessing protected field of OuterClass                System.out.println(outer.m);    //accessing public field of OuterClass          }      }  } |

* Member inner classes can be abstract or can be final but not both.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | class OuterClass  {      abstract class InnerClassOne      {          //abstract Inner Class      }        final class InnerClassTwo      {          //final inner class      }  } |

This is all about Member Inner Classes. Tomorrow, we will discuss about Local Inner Classes.

# 14. Local Inner classes in Java:

Local inner class in java is non-static nested class which is declared inside a method or a block.

Let’s discuss some of behaviors of Local Inner Class in java.

* Local inner classes must be defined inside a method or a block.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52 | class OuterClass  {      static      {          class LocalInnerClassOne          {              //Class defined inside Static Initialization Block          }      }        {          class LocalInnerClassTwo          {              //Class defined inside Instance Initialization Block          }      }        void methodOne()      {          class LocalInnerClassThree          {              //Class defined inside a non-static method          }      }        static void methodTwo()      {          class LocalInnerClassFour          {              //Class defined inside a static method          }      }        void methodThree()      {          if(true)          {              class LocalInnerClassFive              {                  //Class defined inside if-statement              }          }            for(int i=0; i<=5; i++)          {              class LocalInnerClassSix              {                  //Class defined inside a for loop              }          }      }  } |

* Local Inner Classes can’t be static. Because, local inner classes are nothing but local variables and local variables can’t be static.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | class OuterClass  {      void methodOne()      {          static class LocalInnerClass          {              //compile time error              //Local Inner class can't be static          }      }  } |

* Local inner classes can’t have static members. Only non-static members are allowed inside local inner classes. But local inner classes can contain static and final field.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | class OuterClass  {      void methodOne()      {          class LocalInnerClass          {              int i;     //can contain Non-static field                static final int j = 10; //can contain static and final field                static int k;   //Compile time error : can't have static field                {                  //can contain instance initializer              }                static              {                  //can't have static initializer              }                void methodOne()              {                  //can contain non-static method              }                static void methodTwo()              {                  //compile time error : can't have static method              }          }      }  } |

* To access members of local inner class, you must create an instance of it.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | class OuterClass  {      void methodOne()      {          class LocalInnerClass          {              int i;     //Non-static field                static final int j = 10; //static and final field                void methodOne()              {                  System.out.println("From LocalInnerClass");              }          }            System.out.println(LocalInnerClass.j);  //static and final field can be accessed directly through class name            LocalInnerClass inner = new LocalInnerClass();  //Creatin an object to LocalInnerClass            System.out.println(inner.i);     //accessing non-static field through object reference            inner.methodOne();   //calling non-static method through object reference      }  }    public class InnerClasses  {      public static void main(String args[])      {          OuterClass outer = new OuterClass();          outer.methodOne();      }  } |

* Local inner classes are local to a method or a block in which they are defined. i.e you can’t use local inner classes outside the method or block in which they are defined.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | class OuterClass  {      void methodOne()      {          class LocalInnerClass          {              int i;     //Non-static field                static final int j = 10; //static and final field                void methodOne()              {                  System.out.println("From LocalInnerClass");              }          }            LocalInnerClass inner = new LocalInnerClass();            //can use LocalInnerClass within the method in which it is defined      }        void methodTwo()      {          LocalInnerClass inner = new LocalInnerClass();  //compile time error            //can't use LocalInnerClass outside the methodOne().      }  } |

* Only final local variables of methods or blocks containing local inner class can be used inside local inner class.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | class OuterClass  {      void methodOne()      {          int i;    //Non-final local variable            final int j;   //final local variable            class LocalInnerClass          {              void methodOne()              {                  System.out.println(i);  //compile time error                    //can't use non-final local variable                    System.out.println(j);  //can use final local variable              }          }      }  } |

* Local inner classes can not be declared with access modifiers. i.e Local inner classes can not be private, protected and public. But they can have private, public, protected and default members in them.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | class OuterClass  {      void methodOne()      {          private class LocalInnerClassOne          {              //Compile time error              //Local inner class can't be private          }            protected class LocalInnerClassTwo          {              //Compile time error              //Local inner class can't be protected          }            public class LocalInnerClassThree          {              //Compile time error              //Local inner class can't be public          }            class LocalInnerClassFour          {              private int i;    //can have private member              protected int j;  //can have protected member              public int k;    //can have public member              int m;          //can have default member          }      }  } |

* Local inner classes can be abstract or can be final but not both.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | class OuterClass  {      void methodOne()      {          abstract class LocalInnerClassOne          {              //abstract local inner class          }            final class LocalInnerClassTwo          {              //final local inner class          }      }  } |

This is all about Local Inner Classes in java. Tomorrow, we will discuss about Anonymous Inner Classes.

# 15. Anonymous Inner classes in Java:

Anonymous inner class, the name itself suggest that it is a class without a name. Anonymous inner class in java is an inner class or non-static nested class without a name.

Consider the below class definition.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | class SuperClass  {      void methodOne()      {          System.out.println("From SuperClass");      }        void methodTwo()      {          System.out.println("From SuperClass");      }  } |

Let’s consider that we have a requirement in which the above class ‘SuperClass’ has to be re-used with little modification to the ‘methodOne()’. To do this, we have to create a subclass to ‘SuperClass’ and override the ‘methodOne()’ method. Let’s implement this.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | class SubClass extends SuperClass  {      @Override      void methodOne()      {          System.out.println("From Sub Class");      }  } |

To use the methodOne(), we have to create an object of ‘SubClass’ type and call ‘methodOne()’ from that object.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public class InnerClasses  {      public static void main(String args[])      {          SubClass subclass = new SubClass();          subclass.methodOne();      }  } |

This method of implementing is little bit lengthy. There is one more method of implementing this requirement which takes less time than this and you need to write only few lines of code to implement this requirement. That is called Anonymous Inner Class.

Let’s implement this requirement with anonymous inner class.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public class InnerClasses  {      public static void main(String args[])      {          SuperClass superclass = new SuperClass()          {              @Override              void methodOne()              {                  System.out.println("From Anonymous Inner Class");              }          };          superclass.methodOne();      }  } |

That’s it. You just have to create an object reference variable of type ‘SuperClass’ and override the method which needs modification in the curly brackets and end with semicolon. No need to create sub class separately. This method is easier than the above method. isn’t it?.

Let’s discuss some of features of anonymous inner class.

* Anonymous inner classes don’t have name. They are nameless.
* You can create only one object to anonymous inner class. If you want to create another object, you have to write the whole class again.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | public class InnerClasses  {      public static void main(String args[])      {          //First Object Creation          SuperClass firstObject = new SuperClass()          {              @Override              void methodOne()              {                  System.out.println("From First Object");              }          };            //Second Object Creation          SuperClass secondObject = new SuperClass()          {              @Override              void methodOne()              {                  System.out.println("From Second Object");              }          };      }  } |

* When you are creating an anonymous inner class, you are actually creating a sub class to a class which needs to be modified. This sub class doesn’t have name and it is declared in another class. That’s why it is called Anonymous Inner Class.
* While creating an anonymous inner class you are also creating an object to that subclass and it is referenced by super class reference variable. This also shows the polymorphism. Because, Super class reference variable can refer to super class object and also it’s sub class object.
* Using anonymous inner class, you can implement both abstract classes and interfaces.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46 | abstract class AbstractClass  {      abstract void methodOne();        abstract void methodTwo();  }    interface InterfaceClass  {      abstract void methodOfInterface();  }    public class InnerClasses  {      public static void main(String args[])      {          //Implementing abstract class          AbstractClass a = new AbstractClass()          {              @Override              void methodOne()              {                  System.out.println("From AbstractClass");                }                //You have to override second abstract method also,              //otherwise, you will get compile time error.              @Override              void methodTwo()              {                  System.out.println("From AbstractClass");              }          };            //Implementing Interface          InterfaceClass i = new InterfaceClass()          {              @Override              public void methodOfInterface()              {                  System.out.println("From Interface");              }          };      }  } |

# 16. Variable Hiding or Shadowing in Java:

As we all know that there are two types of variables. One is Local Variables and another one is global variables. Global variables may be static or non-static.

Have a look at the following example.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public class Shadowing  {      int x = 10;        void methodOne(int x)      {          //local x shadows or hides global x            System.out.println(x);   //output : 20      }        public static void main(String[] args)      {          Shadowing shadow = new Shadowing();          shadow.methodOne(20);      }  } |

In the above example, Class ‘Shadowing’ has global variabe ‘x’ and it’s scope is inside the class. ‘methodOne()’ of this class also has a local variable with the same name  and it’s scope is inside the method. If you try to print the ‘x’ inside the methodOne(), the value of local variable is printed but not that of global variable. Because, local ‘x’ shadows or hides global ‘x’. This is called variable hiding or shadowing.

## **Variable Hiding Or Shadowing In Java :**

A variable is hidden or shadowed, if there is another variable exist with the same name in the nearer scope.

## **How To access hidden or shadowed variable?.**

* When a global variable is hidden or shadowed by local variable.

To access the hidden static variable inside a method or block – use class name and to access the hidden non-static variable – use this keyword. this keyword refers to current instance of the class.

For example,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | public class Shadowing  {      static int x = 10;        int y = 20;        void methodOne(int x, int y)      {          //accessing hidden static variable using class name            System.out.println(Shadowing.x);   //output : 10            //accessing hidden non-static variable using this keyword.            System.out.println(this.y);    //output : 20      }        public static void main(String[] args)      {          Shadowing shadow = new Shadowing();          shadow.methodOne(30, 40);      }  } |

**Note :** Static hidden variable can be accessed using this keyword also, but only in non-static context. Because, this can not be used in static context.

* When a Super Class variable is hidden or shadowed by sub class variable :

If sub class has variables with the same name as that of super class, it hides or shadows super class variable. To access, Super class hidden variables, just refer sub class object through super class reference variable. For example,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | class SuperClass  {      int x = 10;        static int y = 20;  }    class SubClass extends SuperClass  {      //subclass hides superclass variables with same name        int x = 30;        static int y = 40;  }    public class Shadowing  {        public static void main(String[] args)      {          SubClass subClass = new SubClass();  //Creating object to SubClass            System.out.println(subClass.x);   //Output : 30          System.out.println(subClass.y);   //Output : 40            //accessing super class hidden variables in the same object.            SuperClass superClass = subClass;   //casting from subclass to superclass            System.out.println(superClass.x);     //Output : 10          System.out.println(superClass.y);     //Output : 20      }  } |

From the above example, it is clear that super class variables are not overridden in the sub class but they are hidden.

**Note :**If you want to access super class variable in sub class itself, use [super](https://javaconceptoftheday.com/super-and-this-keywords-in-java/) keyword.

* When Outer Class Variable is hidden in Inner Class :

If inner class has a variable with the same name as that of an outer class, then it hides or shadows outer class variable. To access non-static hidden outer class variable in inner class, use this syntax : **OuterClassName.this.variableName** and to access static hidden outer class variable in inner class use the same syntax or use **OuterClassName.variableName**. For example,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47 | class OuterClass  {      int x = 10;        static int y = 20;        class InnerClass      {          int x = 30;            static final int y = 40;            void methodOfInnerClass()          {              System.out.println(x);  //output : 30                System.out.println(y);  //output : 40                //accessing non-static hidden outer class variable in inner class                System.out.println(OuterClass.this.x);  //Output : 10                //accessing static hidden outer class variable in inner class                System.out.println(OuterClass.this.y);   //Output : 20                //OR                System.out.println(OuterClass.y);  //Output : 20            }      }  }    public class Shadowing  {        public static void main(String[] args)      {          OuterClass outer = new OuterClass();            OuterClass.InnerClass inner = outer.new InnerClass();            inner.methodOfInnerClass();        }  } |

# 17. Inheritance Of Inner Classes in Java:

we will see various scenarios of inheritance of inner classes in java.

* One inner class can extend another inner class of the same class.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | class OuterClass  {      class InnerClassOne      {          int x = 10;            void methodOfInnerClassOne()          {              System.out.println("From InnerClassOne");          }      }        class InnerClassTwo extends InnerClassOne      {          //One Inner Class can extend another inner class      }  }    public class InnerClasses  {      public static void main(String args[])      {          OuterClass outer = new OuterClass();   //Instantiating OuterClass            OuterClass.InnerClassTwo innerTwo = outer.new InnerClassTwo();  //Instantiating InnerClassTwo            System.out.println(innerTwo.x);    //Accessing inherited field x from InnerClassOne          innerTwo.methodOfInnerClassOne();  //calling inherited method from InnerClassOne      }  } |

* An inner class can be extended by another class outside of it’s outer class. If you are extending static inner class (Static nested class), then it is a straight forward implementation. If you are extending non-static inner class, then sub class constructor must explicitly call super class constructor using an instance of outer class. Because, you can’t access non-static inner class without the instance of outer class.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | class OuterClass  {      static class InnerClassOne      {          //Class as a static member      }        class InnerClassTwo      {          //Class as a non-static member      }  }    //Extending Static inner class or static nested class  class AnotherClassOne extends OuterClass.InnerClassOne  {      //static nested class can be referred by outer class name,  }    //Extending non-static inner class or member inner class  class AnotherClassTwo extends OuterClass.InnerClassTwo  {      public AnotherClassTwo()      {          new OuterClass().super();  //accessing super class constructor through OuterClass instance      }  } |

* When an outer class is extended by it’s sub class, Member inner classes will not be inherited to sub class. To use inner class properties inside the sub class of outer class, sub class must also have an inner class and that inner class must extend inner class of the outer class. For example,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47 | class OuterClass  {      int x;        void methodOfOuterClass()      {          System.out.println("From OuterClass");      }        //Class as a member      class InnerClass      {          int y;      }  }    class AnotherClass extends OuterClass  {       //Only fields and methods are inherited.      // To use inner class properties,      //it's inner class must extend inner class of it's super class      class AnotherInnerClass extends InnerClass      {          //Inner Class of AnotherClass extends Inner Class of OuterClass      }  }    public class InnerClasses  {      public static void main(String args[])      {          AnotherClass anotherClass = new AnotherClass();  //creating AnotherClass Object            System.out.println(anotherClass.x);    //accessing inherited field x from OuterClass            anotherClass.methodOfOuterClass();    //calling inherited method from OuterClass           //Using the properties of InnerClass            AnotherClass.AnotherInnerClass anotherInnerClass = anotherClass.new AnotherInnerClass();            //creating object to AnotherInnerClass            System.out.println(anotherInnerClass.y);  //accessing inherited field y from InnerClass        }  } |

* Inner class can extend it’s outer class. But, it does not serve any meaning. Because, even the private members of outer class are available inside the inner class. Even though, When an inner class extends its outer class, only fields and methods are inherited but not inner class itself.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | class OuterClass  {      int x;        void methodOfOuterClass()      {          System.out.println("From OuterClass");      }        //Class as a member      class InnerClass extends OuterClass      {          //only fields and methods are inherited, but not member Inner Classes      }        class InnerClassOne      {          //another class as a member      }  }    public class InnerClasses  {      public static void main(String args[])      {          OuterClass outer = new OuterClass();          //You have to create OuterClass object to access non-static inner class            OuterClass.InnerClass inner = outer.new InnerClass();  //creating object to InnerClass            System.out.println(inner.x);   //accesiing inherited field x            inner.methodOfOuterClass();   //accessing inherited method      }  } |

# 18. Packages in Java:

Packages in java are used to organize related or similar classes, interfaces and enumerations into one group. For example, java.sql package has all classes needed for database operation. java.io package has classes related to input-output operation.Packages are also used to avoid naming conflict between the classes. Using packages, you can give same name to different classes.

Let’s discuss Packages in java.

* Packages are declared using keyword ‘package’. They should be declared in the first statement in a java file. If you try to declare packages at any other statements, you will get compile time error.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | package com;  class A  {       //Some statements  }  //package com; If you declare here, it gives compile time error |

* Only alphabets, numbers and an underscore are allowed in naming the packages. By convention, names of package should start with lowercase although it is not a condition. Package name should start with a alphabets or underscore but not with a number.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | package javaConcept;      //Valid package name  package java\_Concept;     //Valid package name  package java\_12;          //Valid package name  package 12\_java;          //Invalid package name, should not start with a number.  package \_java12           //Valid package Name  package JAVA;            //Valid package name but not recommended. |

* When you declare a package name in your java file, and after compiling it with [-d option](https://javaconceptoftheday.com/javac-command-and-java-command/), a folder with the same name is created in the specified location and all generated .class files will be stored in that folder.
* You can give same name to more than one classes in different packages.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | package pack1;  class A  {       //Some statements  }    package pack2;  class A  {       //Some statements  } |

* You can compile all the classes in a package at a time like this,

**>javac pack1/\*.java**

all the classes in the package pack1 are compiled at a time.

To run the program, simply call  the class which has main method in it.

**>java pack1.MainClass**

* Packages can have any number of sub packages. While declaring, packages and sub packages are separated by **‘.’**For example,

|  |  |
| --- | --- |
| 1  2  3  4  5 | package pack1.subpack1;  class A  {      //Some Statements  } |

When you compile above example with -d option, generated .class file is stored in subfolder subpack1 of pack1 folder in the specified location.

# 19. Access Modifier:

## **Access Modifiers In Java :**

Access modifiers in java are used to control the visibility of a field, method, class and constructor. There are 4 access modifiers in java. They are : **1). Private   2). Default or Package  3). Protected  4). Public**

Let’s discuss these access modifiers one by one.

## **1). Private**

**Usage of Private members :**

Private members of a class whether it is a field or method or constructor can not be accessed outside the class.

**Inheritance of Private Members :**

Private members will not be inherited to sub class.

**Important Note :**

1). Class can not be a private except inner classes. Inner classes are nothing but again members of outer class. So members of a class (field, method, constructor and inner class) can be private but not the class itself.

2). We can’t create sub classes to that class which has only private constructors.

Look at the below examples,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | class A  {      private int i;        private void methodOfClassA()      {          //Private Method          System.out.println(i);  //Private field can be used within class          B b = new B();          //Private inner class can be used within class      }        private class B      {        //Private Inner Class      }  }    class C extends A  {      void methodOfClassC()      {          //System.out.println(i);  Private member can not be inherited          A a = new A();          //System.out.println(a.i);     Private field can not be used outside the class          //a.methodOfClassA();          Private method can not be used outside the class          //A.B b = new A.B();           Private inner class can not be used outside the class      }  } |
| 1  2  3  4 | private class A  {       //Outer class Can not be private  } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | class A  {      private A()      {          //Private Constructor      }      private A(int i)      {          //Private constructor      }  }    class B extends A  {      //Can't create subclass to the class      //which has only private constructors  } |

### **2). Default or Package or No-Access Modifiers**

**Usage of Default members :**

Default members or members with No-Access modifiers are accessed or visible within the package only. It applies to outer classes also.

**Inheritance Of Default Members :**

Default members can be inherited to sub classes within package.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51 | package pack1;  class A  {      int i;        A()      {          //Constructor with default modifier      }        void methodOfClassA()      {          //Method with default access modifier          System.out.println(i);          B b = new B();      }        class B      {        //Inner Class with default access modifier      }  }    class C extends A  {      void methodOfClassC()      {          System.out.println(i);        //Default field can be inherited within package            A a = new A();          System.out.println(a.i);     //Default field can be used within the package          a.methodOfClassA();          //Default method can be used within the package          A.B b = new A.B();           //Default inner class can be used within the package      }  }    package pack2;  //import pack1.A;      Class A with default access modifier not visible outside the package    /\*class D extends A      Default Class can not have sub class outside the package  {      void methodOfClassD()      {          System.out.println(i);        Default field can not be inherited outside package            A a = new A();           Can't use constructor with default access modifier outside the package          System.out.println(a.i);     Default field can not be used outside the package          a.methodOfClassA();          Default method can not be used outside the package          A.B b = new A.B();           Default inner class can not be used outside the package      }  }\*/ |

### **3). Protected**

**Usage of Protected Member :**

Protected member can be used within the package only.

**Inheritance Of Protected Member :**

Protected Member can be inherited to any sub classes.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47 | package pack1;    public class A  {      protected int i;        protected void methodOfClassA()      {          //Protected method          System.out.println(i); //Protected field can be used within class          B b = new B();         //Protected Inner Class can be used within class.      }        protected class B      {        //Protected Inner Class      }  }    class C extends A  {      void methodOfClassC()      {          System.out.println(i);        //Protected field can be inherited to any sub class            A a = new A();          System.out.println(a.i);     //Protected field can be used within the package          a.methodOfClassA();          //Protected method can be used within the package          A.B b = new A.B();           //Protected Inner Class can be used within the package      }  }    package pack2;  import pack1.A;    class D extends A  {      void methodOfClassD()      {          System.out.println(i);        //Protected field can be inherited to any sub class            A a = new A();          //System.out.println(a.i);     Protected field can not be used outside the package          //a.methodOfClassA();          Protected method can not be used outside the package          //A.B b = new A.B();           Protected inner class can not be used outside the package      }  } |

**Important Note :**

1). Outer class can not be protected.

2). We can create sub classes to a class which has only protected constructors but we can’t create objects to that class outside the package.

### **4). Public**

**Usage of Public members :**

Public members can be used anywhere.

**Inheritance Of Public Members :**

Public members can be inherited to any sub class.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47 | package pack1;    public class A  {      public int i;        public void methodOfClassA()      {          //public method          System.out.println(i); //public field can be used anywhere          B b = new B();         //public Inner Class can be used anywhere.      }        public class B      {        //public Inner Class      }  }    class C extends A  {      void methodOfClassC()      {          System.out.println(i);        //public field can be inherited to any sub class            A a = new A();          System.out.println(a.i);     //public field can be used anywhere          a.methodOfClassA();          //public method can be used anywhere          A.B b = new A.B();           //public Inner Class can be used anywhere.      }  }    package pack2;  import pack1.A;    class D extends A  {      void methodOfClassD()      {          System.out.println(i);        //public field can be inherited to any sub class            A a = new A();          System.out.println(a.i);     //Public field can be used anywhere          a.methodOfClassA();          //Public method can be used anywhere          A.B b = new A.B();           //Public inner class can be used anywhere      }  } |

Above concepts can be summarized like below,

|  |  |  |
| --- | --- | --- |
| **Access Modifier** | **Usage or Access or Visibility** | **Inheritance** |
| **private** | Within Class Only | Can not be inherited |
| **Default or No-Access Modifier** | Within Package Only | Can be inherited to sub class within package |
| **Protected** | Within Package Only | Can be inherited to any subclass |
| **Public** | Anywhere | To any subclass |

# 20. Illegal Forward Reference Error In Java:

## **Illegal Forward Reference Error In Java :**

Illegal forward reference error in java is a compile time error. You will encounter with this error when you try to use a field before it is defined just like in the below example,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | class A  {      static      {          System.out.println(i);  //You will get compile time error here      //  can not refer a field before it is defined      }        static int i;  } |

That means we can’t use a field before it is defined. But there is an interesting point, We can initialize a field before it is defined. Have look at below code. There will be no compile time errors here.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class A  {      static      {          i = 10;   //A field can be initialized before it is defined.      }        static int i;  } |

Above two examples can be summarized like this, **We can’t use a field before it is defined but we can initialize a field before it is defined.**

Let’s have a look at some other examples.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public class A  {      static int i;        static      {             System.out.println(i);      //     System.out.println(j);  can't use a field before it is defined      //     i=j;                   even you can't use it to initialize other fields             j=i;                    //but, can be initialized.      }        static int j;  } |

Even you can’t use undefined field to initialize other fields (Line 9 in the above example).

**Important Note :  In the assignment statement, you can use undefined field on LHS only but not on RHS.**

Illegal forward reference error applies to non-static variables also.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public class A  {      {      //  System.out.println(i); can't use a field before it is defined          i = 20;               //but, can be initialized      }        int i;  } |

In the case of local variables, neither you can use it nor you can initialize it before it is defined.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class A  {      void methodOfClassA()      {      //  System.out.println(i); can't use a local field before it is defined      //  i = 10;                can't initialize a local field before it is defined          int i;      }  } |

Illegal forward reference issue can be analysed like this. Consider the following statements.

|  |  |
| --- | --- |
| 1  2  3 | int a = 10;  int b = c;  int c = 30; |

Compiler always compiles the program from top to bottom. In the first statement, compiler declares variable ‘a’ and assigns value 10 to it. In the second statement, it declares variable ‘b’ and tries to find where is ‘c’. But, it is not declared yet. Therefore, it gives illegal forward reference error.**Illegal forward reference is nothing but you are referring to something in advance that does not exist yet.**

# 21. Type Casting in Java:

Type casting in java or simply casting is used to convert data from one data type to another data type. Please note that by using casting, data can not be modified but only type of data can be modified.

There are two types of casting,

1) Primitive Casting.

2) Derived Casting

## **1) Primitive Casting.**

Primirive Casting is used to convert data from one primitive data type to another primitive data type.

Consider primitive data types in java which represent the numbers.

**These are data types with no decimal places.**

1) byte     2) short     3) int      4) long

**and these are data types with decimal places.**

5) float     6) double

When you put them in the increasing order of their memory size, you get

**byte < short < int < long < float < double.**

Please remember this order we will be using this order in below examples. byte is the smallest data type and double is the biggest data type in terms of memory size.

There are two types in primitive casting. **1) Auto Widening   2) Explicit Narrowing**

**1) Auto Widening**

When you are converting data from small sized data type to big sized data type, i.e when you are converting data from left-placed data type to right-placed data type in the above order, auto widening will be used. For example, when you are converting byte to short or short to int, auto widening will be used.

Go through this example.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | class AutoWidening  {      static float methodOne(int i)      {          long j = i;     //int is auto widened to long          return j;       //long is auto widened to float      }        public static void main(String[] args)      {          byte b = 10;          short s = b;      //byte is auto widened to short          double d = methodOne(s);    //short is auto widened to int and float to double          System.out.println(d);      }  } |

**2) Explicit Narrowing**

When you are converting data from big sized data type to small sized data type, i.e when you are converting data from right-placed data type to left-placed data type in the above order, explicit narrowing will be used. For example, when you are converting double to float or float to int, explicit narrowing will be used.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | class ExplicitlyNarrowing  {      static short methodOne(long l)      {          int i = (int) l;     //long is explicitly narrowed to int          return (short)i;       //int is explicitly narrowed to short      }        public static void main(String[] args)      {          double d = 10.25;          float f = (float) d;      //double is explicitly narrowed to float          byte b = (byte) methodOne((long) f);    //float is explicitly narrowed to long and short to byte          System.out.println(b);      }  } |

## **2) Derived Casting**

Derived casting is used to change the type of object from one user defined data type to another user defined data type in the class hierarchy.

There are two types in derived casting. **1) Auto-up Casting  2) Explicit Down Casting.**

**1) Auto-Up Casting**

Auto-Up Casting is used to change the type of object from sub class type to super class type. i.e an object of sub class type is automatically converted to an object of super class type. For example,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | class A  {      int i = 10;  }    class B extends A  {      int j = 20;  }    class C extends B  {      int k = 30;  }    class D extends C  {      int m = 40;  }    public class AutoUpCasting  {      public static void main(String[] args)      {          D d = new D();          C c = d;       // D type object is Auto-Up Casted to C type          B b = d;      // D type object is Auto-Up Casted to B type          C c1 = new C();          A a = c1;    // C type object is Auto-Up Casted to A type          A a1 = new B(); // B type object is Auto-Up Casted to A type      }  } |

**2) Explicit Down Casting**

Explicit down Casting is used to change the type of object from super class type to sub class type. i.e you have to explicitly convert an object of super class type to an object of sub class type. For example,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | class A  {      int i = 10;  }    class B extends A  {      int j = 20;  }    class C extends B  {      int k = 30;  }    class D extends C  {      int m = 40;  }    public class ExplicitDownCasting  {      public static void main(String[] args)      {          A a = new A();          B b = (B) a;   //A type is explicitly downcasted to B type          C c = (C) a;   //A type is explicitly downcasted to C type          D d = (D) a;   //A type is explicitly downcasted to D type          B b1 = new B();          D d1 = (D) b1;  //B type is explicitly downcasted to D type          d1 = (D) new C();  //C type is explicitly downcasted to D type      }  } |

# 22. ClassCastException In Java:

ClassCastException in java is a run time error it occurs when an object can not be casted to another type.

An object is automatically upcasted to its super class type. You need not to mention class type explicitly. But, when an object is supposed to be downcasted to its sub class type, then you have to mention class type explicitly. In such case, there is a possibility of occurring class cast exception. In most of time, it occurs when you are trying to downcast an object explicitly to its sub class type.

Try to run below program.

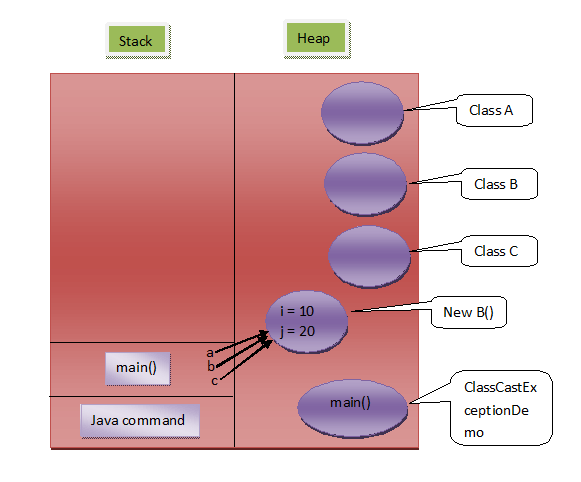
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | package com;  class A  {      int i = 10;  }    class B extends A  {      int j = 20;  }    class C extends B  {      int k = 30;  }    public class ClassCastExceptionDemo  {      public static void main(String[] args)      {          A a = new B();   //B type is auto up casted to A type          B b = (B) a;     //A type is explicitly down casted to B type.          C c = (C) b;    //Here, you will get class cast exception          System.out.println(c.k);      }  } |

You will get ClassCastException. Below is the sample of the error.

**Exception in thread “main” java.lang.ClassCastException: com.B cannot be cast to com.C**  
**at com.ClassCastExceptionDemo.main(ClassCastExceptionDemo.java:23)**

In the above example, Class B extends Class A and Class C extends Class B. In the main method, Class B-type object is created (Line 21). It will be having two non-static fields. one field (int i) is inherited from class A and another one is its own field (int j). ‘a’ is Class A-type reference variable which will be pointing to this newly created object. In the next statement (Line 22), reference variable ‘a’ is assigned to ‘b’ which is Class B-type reference variable. After execution of this statement, ‘b’ will also be pointing to the same object to which ‘a’ is pointing. In the third statement, ‘b’ is assigned to ‘c’ which is Class C-type reference variable. So, ‘c’ will also be pointing to same object to which ‘a’ and ‘b’ are pointing. While executing this statement, you will get run time exception called Class Cast Exception.

The memory allocation of above program can be diagrammatically represented as,



**Why you got this exception?**

Every sub class extends its super class. i.e every child class will have some additional properties along with some inherited properties from its parent class. In the above example, Class A has one property (int i). Class B has two properties, one is it’s own and another one is inherited. Class C has three properties. one is it’s own and two are inherited. In this example, Class C-type reference variable is referring to Class B-type object. Class B-type object will be having only two properties. But, through Class C-type reference variable, you can access Class C’s own property (int k) like in the line 24. But, actually this property does not exist in Class B-type object. This creates the confusion. Class B-type can not be casted to Class C-type. That’s why, you will get class cast exception.

Put ClassCastException in simple terms. ClassCastException occurs when code has attempted to cast an object to a type of which it is not an object. In the above example, Class B is a Class A type but Class B is not a Class C type. Therefore, you are getting ClassCastException.

Consider one more case of ClassCastException.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public class ClassCastExceptionDemo  {      public static void main(String[] args)      {          Object o = new String();          Integer i = (Integer) o;      }  } |

We all know that every class in java is a sub class of java.lang.Object class. String is also a subclass of Obeject class and Integer is also a subclass of Object class. In the above example, String object is created and it is automatically up casted to Object type. Further, this object is explicitly downcasted to Integer type. This causes ClassCastException, because, String object is not an Integer type.

# 23. Method Overloading In Java:

When a class has more than one method with same name, then we call that method is overloaded. The overloaded methods will have different number of arguments or different types of arguments, but name of the methods remains same.

Compiler checks **method signature** for duplicate methods or for method overloading. method signature consist of three things, **1) Method Name   2) Number Of Arguments   3) Types of arguments.**

If these three things are same for any two methods in a class, then compiler gives **duplicate method error.**

Compiler first checks method name. If it is same, then it checks number of arguments. If methods differs in number of arguments, then it does not check types of argument. It treats as methods are overloaded. If number of arguments are same then compiler checks types of arguments. If types of arguments are also same, then compiler will give duplicate method error. If types of arguments are not same, then compiler will treat them as methods are overloaded.

For method overloading to be successful, method name must be same and methods must have different number of arguments or different types of arguments. If method names are different, then those methods will be treated as two different methods.

Go though this example,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | public class MethodOverloading  {      void methodOverloaded()      {          //No argument method      }        void methodOverloaded(int i)      {          //One argument is passed      }        void methodOverloaded(double d)      {          //One argument is passed but type of argument is different      }        void methodOverloaded(int i, double d)      {          //Two argument method          //Method signature of this method is methodOverloaded(int, double)      }        void methodOverloaded(double d, int i)      {          //It is also two argument method but type of arguments changes          //Method signature of this method is methodOverloaded(double, int)      }        void methodOverloaded(double d1, int i1)      {                  //It has same method signature methodOverloaded(double, int) as of above method          //So, it is a Duplicate method, You will get compile time error here      }        void differentMethod()      {          //Different method      }  } |

Overloaded methods may have same return types or different return types. It does not effect method overloading.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | public class MethodOverloading  {      void methodOverloaded()      {          //No argument method, return type is void      }        int methodOverloaded(int i)      {          //Returns int type          return i;      }        int methodOverloaded(double d)      {          //Same return type as of above method          return 0;      }        void methodOverloaded(double d)      {          //Duplicate method because it has same method signature as of above method      }  } |

**Important Note :**

If two methods have same signature and different return types, then those methods will not be treated as two different methods or methods overloaded. For duplication, compiler checks only method signature not return types. If method signature is same, straight away it gives duplicate method error.

Overloaded methods may have same access modifiers or different access modifiers. It also does not effect method overloading.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | public class MethodOverloading  {      private void methodOverloaded()      {          //No argument, private method      }        private int methodOverloaded(int i)      {          //One argument private method          return i;      }        protected int methodOverloaded(double d)      {          //Protected Method          return 0;      }        public void methodOverloaded(int i, double d)      {          //Public Method      }  } |

Overloaded methods may be static or non-static. This also does not effect method overloading.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | public class MethodOverloading  {      private static void methodOverloaded()      {          //No argument, private static method      }        private int methodOverloaded(int i)      {          //One argument private non-static method          return i;      }        static int methodOverloaded(double d)      {          //static Method          return 0;      }        public void methodOverloaded(int i, double d)      {          //Public non-static Method      }  } |

From the above examples, it is clear that compiler will check only method signature for method overloading or for duplicate methods. It does not check return types, access modifiers and static or non-static.

# 24. Method Overridding In Java:

## **Method Overriding In Java :**

When a class extends its super class, all or some members of super class are inherited to sub class. When a inherited super class method is modified in the sub class, then we call it as method is overrided. Through method overriding, we can modify super class method according to requirements of sub class.

Method Overriding in java is most useful features of java. Through inheritance we can reuse already existed code and through method overriding we can modify that reused code according to our requirements. This can be best explained with example.

Read through this example,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | class SuperClass  {      void methodOfSuperClass()      {          System.out.println("From Super Class");      }  }    class SubClass extends SuperClass  {      void methodOfSuperClass()      {          //SuperClass method is overrided          //We can keep any tasks here according to our requirements.          System.out.println("From Sub Class");      }  }    public class MethodOverriding  {      public static void main(String[] args)      {          SuperClass superclass = new SuperClass();          superclass.methodOfSuperClass();         //Output : From Super Class          SubClass subclass = new SubClass();          subclass.methodOfSuperClass();          //Output : From Sub Class      }  } |

Let’s discuss rules to be followed while overriding a method.

* **Name of the overrided method** must be same as in the super class. You can’t change name of the method in subclass.
* **Return Type Of Overrided Method :**

The return type of the overrided method must be compatible with super class method. If super class method has primitive data type as its return type, then overrided method must have same return type in sub class also. If super class method has derived or user defined data type as its return type, then return type of sub class method must be of same type or its sub class. For example,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | class SuperClass  {      void firstMethodOfSuperClass()      {          System.out.println("From Super Class");      }        double secondMethodOfSuperClass()      {          return 0.0;      }        Object thirdMethodOfSuperClass()      {          return new Object();      }  }    class SubClass extends SuperClass  {      int firstMethodOfSuperClass()      {          //Compile time error, return type must be void not int      }        void secondMethodOfSuperClass()      {          //Complie time error, return type must be double not void      }        String thirdMethodOfSuperClass()      {          //No Compile time error,          //return type is compatible with super class method, because          //String is sub class of Object class          return new String();      }  } |

* **Visibility Of Overrided method :**

You can keep same visibility or increase the visibility of overrided method but you can’t reduce the visibility of overrided methods in the subclass. For example, default method can be overided as default or protected or public method but not as private.For example,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | class SuperClass  {      protected void methodOfSuperClass()      {          System.out.println("From Super Class");      }  }    class SubClass extends SuperClass  {      private void methodOfSuperClass()      {          //Compile time error, can't reduce visibility of overrided method          //here, visibility must be protected or public but not private or default      }  } |

**Note : Visibility goes on decreasing from public to protected to default to private members.**

* **Arguments Of Overrided Methods :**

For method to be properly overrided, You must not change arguments of method in subclass. If you change the number of arguments or types of arguments of overrided method in the subclass, then method will be overloaded not overrided.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | class SuperClass  {      void methodOfSuperClass()      {          System.out.println("From Super Class");      }  }    class SubClass extends SuperClass  {      //This class will have two methodOfSuperClass() methods.      //one is from super class which takes no argument      //and one is below method which takes one argument      void methodOfSuperClass(int i)      {          System.out.println(i);      }  }    public class MethodOverloading  {      public static void main(String[] args)      {          SuperClass superclass = new SuperClass();          superclass.methodOfSuperClass();         //Output : From Super Class          SubClass subclass = new SubClass();          subclass.methodOfSuperClass();          //Output : From Super Class          subclass.methodOfSuperClass(10);       // Output : 10      }  } |

# 25. Polymorphism In Java:

In greek, Poly means many and morph means shapes or forms. So. Polymorphism refers to any entity which takes many form.

Polymorphism in java refers to any entity whether it is an operator or a constructor or any method which takes many forms or can be used for multiple tasks either while compiling or while running a java program.

There are two types of polymorphism in Java.   **1) Static Polymorphism       2) Dynamic Polymorphism**

## **1) Static Polymorphism**

Any entity which shows polymorphism during compile time is called static polymorphism. Operator Overloading, Constructor Overloading and [method overloading](https://javaconceptoftheday.com/method-overloading-in-java/) are best examples of static polymorphism. Because, they show polymorphism during compilation.

In static polymorphism, object used is determined during compilation itself. So, it is called **static binding or Early Binding**.

**Operator Overloading :**For example, Operator ‘+’ can be used to add two numbers and also can be used to concatenate two strings. It is called operator overloading. ‘+’ is the only operator in java which is used for operator overloading.

**Constructor Overloading :**We can include multiple constructors in a class. This is called constructor overloading. Through [constructor overloading](https://javaconceptoftheday.com/constructors-in-java/), we can create objects to the class in multiple ways. This shows the polymorphism.

**Method Overloading :**We can have different forms of same method in the same class. This is called [method overloading](https://javaconceptoftheday.com/method-overloading-in-java/). Through method overloading we can perform different tasks through different forms of the same method. This shows the polymorphism.

In [casting](https://javaconceptoftheday.com/type-casting-in-java/), we have seen super class reference variable can refer to objects of its sub class. This also shows polymorphism. For example,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | class A  {       //Some Statements  }  class B extends A  {       //Some Statements  }  class C extends B  {      //Some Statements  }    public class D  {      public static void main(String[] args)      {          A a = new A();  //A-Class variable refers to A-Class object          a = new B();    //A-Class variable refers to B-Class object          a = new C();    //A-Class variable refers to C-Class object      }  } |

In the above example, ‘a’ is Class A-type reference variable which can be used to refer objects of A-type, B-type or C-type. Because, B-type and C-type are sub class of A-type. This also shows the polymorphism.

## **2) Dynamic Polymorphism**

Any entity which shows polymorphism during run time is called dynamic polymorphism. [Method Overriding](https://javaconceptoftheday.com/method-overriding-java/) is the best example of dynamic polymorphism. It is also called **dynamic binding or late binding**, because type of the object used will be determined at run time only.

Consider the following example,

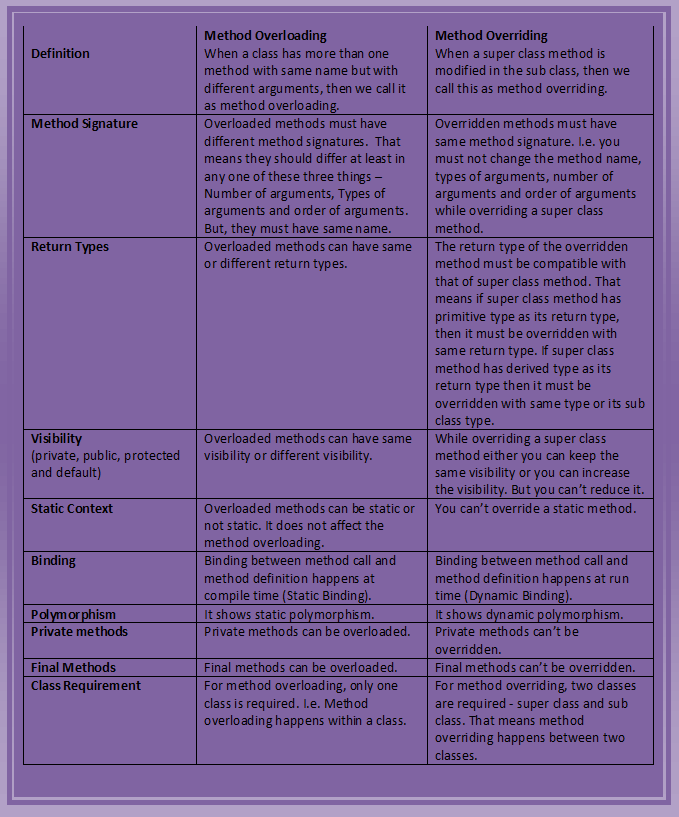
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | class SuperClass  {      void methodOfSuperClass()      {          System.out.println("From Super Class");      }  }  class SubClass extends SuperClass  {      //Super Class Method Overrided      void methodOfSuperClass()      {          System.out.println("From Sub Class");      }  }    public class D  {      static void util(SuperClass superclass)      {          superclass.methodOfSuperClass();          //For each execution of this method, different objects will be passed to it.          //which Object will be used is determined during run time only.          //This shows dynamic polymorphism.      }        public static void main(String[] args)      {          SuperClass superclass1 = new SuperClass();          SubClass subclass = new SubClass();          SuperClass superclass2 = new SubClass();            util(superclass1);  //SuperClass object is passes to util()          util(subclass);     //SubClass object is passed to util()          util(superclass2); //SubClass object is passed to util()      }  } |

# 26. Difference Between Method Overloading And Method Overriding In Java:

## **What is the difference between method overloading and method overriding in java?**

Method overloading and method overriding are two important java concepts which allows java programmer to define the methods with same name but different behavior. Both method overloading and method overriding shows polymorphism. It is also one of the most asked java interview question for freshers. In this article, I have tried to list out the differences between method overloading and method overriding in java. You can also go through the some basics about method overloading and method overriding in java [here](https://javaconceptoftheday.com/method-overloading-in-java/) and [here](https://javaconceptoftheday.com/method-overriding-java/).

|  |  |  |
| --- | --- | --- |
|  | **Method Overloading** | **Method Overriding** |
| **Definition** | When a class has more than one method with same name but with different arguments, then we call it as method overloading. | When a super class method is modified in the sub class, then we call this as method overriding. |
| **Method Signature** | Overloaded methods must have different method signatures.  That means they should differ at least in any one of these three things – Number of arguments, Types of arguments and order of arguments. But, they must have same name. | Overridden methods must have same method signature. I.e. you must not change the method name, types of arguments, number of arguments and order of arguments while overriding a super class method. |
| **Return Types** | Overloaded methods can have same or different return types. | The return type of the overridden method must be compatible with that of super class method. That means if super class method has primitive type as its return type, then it must be overridden with same return type. If super class method has derived type as its return type then it must be overridden with same type or its sub class type. |
| **Visibility**(private, public, protected and default) | Overloaded methods can have same visibility or different visibility. | While overriding a super class method either you can keep the same visibility or you can increase the visibility. But you can’t reduce it. |
| **Static Context** | Overloaded methods can be static or not static. It does not affect the method overloading. | You can’t override a static method. |
| **Binding** | Binding between method call and method definition happens at compile time (Static Binding). | Binding between method call and method definition happens at run time (Dynamic Binding). |
| **Polymorphism** | It shows static polymorphism. | It shows dynamic polymorphism. |
| **Private methods** | Private methods can be overloaded. | Private methods can’t be overridden. |
| **Final Methods** | Final methods can be overloaded. | Final methods can’t be overridden. |
| **Class Requirement** | For method overloading, only one class is required. I.e. Method overloading happens within a class. | For method overriding, two classes are required – super class and sub class. That means method overriding happens between two classes. |



## **Method Overloading Example :**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | public class MainClass  {      static String concateString(String s1, String s2)      {          return s1+s2;      }        static String concateString(String s1, String s2, String s3)      {          return s1+s2+s3;      }        static String concateString(String s1, String s2, String s3, String s4)      {          return s1+s2+s3+s4;      }        public static void main(String[] args)      {          concateString("ONE", "TWO");            concateString("ONE", "TWO", "THREE");            concateString("ONE", "TWO", "THREE", "FOUR");      }  } |

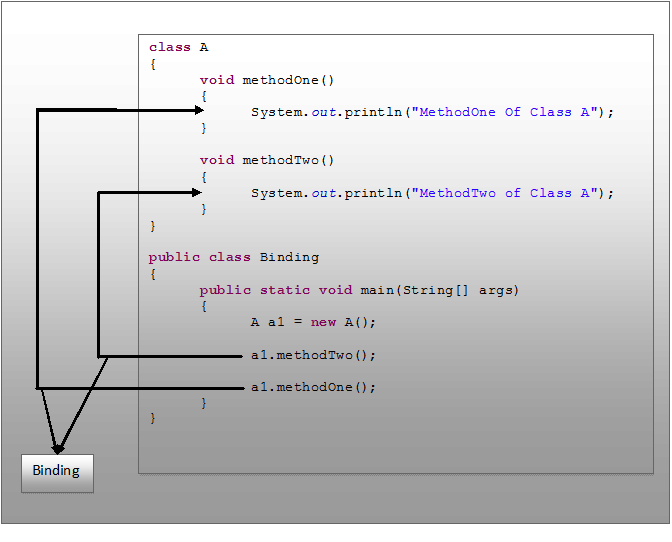
## **Method Overriding Example :**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | class SuperClass  {      void SuperClassMethod()      {          System.out.println("SUPER CLASS METHOD");      }  }    class SubClass extends SuperClass  {      @Override      void SuperClassMethod()      {          System.out.println("SUPER CLASS METHOD IS OVERRIDDEN");      }  } |

# 27. Static Binding And Dynamic Binding:

# Difference Between Static Binding And Dynamic Binding In Java

Before knowing what are the differences between static binding and dynamic binding in java, let’s know what is **binding** first.

**Binding** refers to the link between method call and method definition. This picture clearly shows what is binding.  


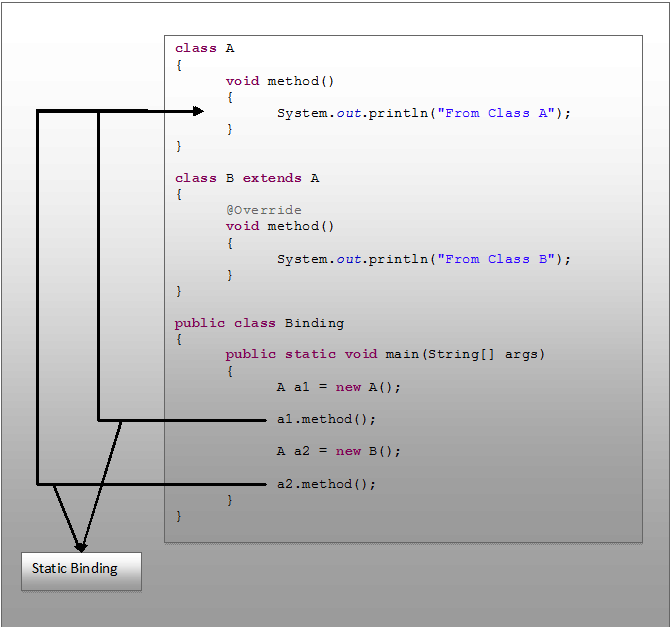
In this picture, **“a1.methodOne()”** call is binding to corresponding **methodOne()** definition and **“a1.methodTwo()”** call is binding to corresponding **methodTwo()** definition.

For every method call there should be proper method definition. This is a rule in java. If compiler does not see the proper method definition for every method call, it throws error.

Now, come to static binding and dynamic binding in java.

## **Static Binding In Java :**

**Static binding** is a binding which happens during **compilation**. It is also called **early binding** because binding happens before a program actually runs.

Static binding can be demonstrated like in the below picture.  


In this picture, **‘a1’** is a reference variable of type Class A pointing to object of class A.  **‘a2’** is also reference variable of type class A but pointing to object of Class B.

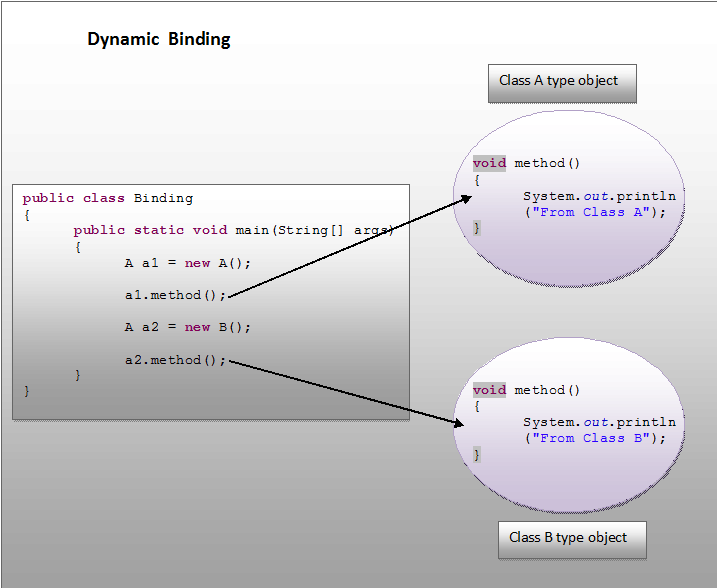
During compilation, while binding, compiler does not check the type of object to which a particular reference variable is pointing. It just checks the type of reference variable through which a method is called and checks whether there exist a method definition for it in that type.

For example, for **“a1.method()”** method call in the above picture, compiler checks whether there exist method definition for **method()** in Class A. Because ‘**a1′** is Class A type. Similarly, for **“a2.method()”** method call, it checks whether there exist method definition for **method()** in Class A. Because ‘**a2′** is also Class A type. It does not check to which object, **‘a1’** and **‘a2’** are pointing. This type of binding is called **static binding**.

## **Dynamic Binding In Java :**

**Dynamic binding** is a binding which happens during **run time**. It is also called **late binding** because binding happens when program actually is running.

During run time actual objects are used for binding. For example, for **“a1.method()”** call in the above picture, **method()** of actual object to which **‘a1’** is pointing will be called. For **“a2.method()”** call, **method()** of actual object to which **‘a2’** is pointing will be called. This type of binding is called dynamic binding.

The dynamic binding of above example can be demonstrated like below.  


## **Differences Between Static Binding And Dynamic Binding In Java :**

The above findings can be summarized like below.

|  |  |
| --- | --- |
| **Static Binding** | **Dynamic Binding** |
| It is a binding that happens at compile time. | It is a binding that happens at run time. |
| Actual object is not used for binding. | Actual object is used for binding. |
| It is also called early binding because binding happens during compilation. | It is also called late binding because binding happens at run time. |
| Method overloading is the best example of static binding. | Method overriding is the best example of dynamic binding. |
| Private, static and final methods show static binding. Because, they can not be overridden. | Other than private, static and final methods show dynamic binding. Because, they can be overridden. |

# 2. ENums:

Enums in java are mainly used for grouping similar kind of constants as a one unit. constants means static and final. Enums are introduced in JDK 1.5 onward. Before that similar kind of constants are grouped by declaring them as static and final in one class. Below example shows how the constants will look without enums.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | class ConstantsWithoutEnums  {      public static final String north = "NORTH";      public static final String south = "SOUTH";      public static final String east = "EAST";      public static final String west = "WEST";  }    public class MainClass  {      public static void main(String[] args)      {          System.out.println(ConstantsWithoutEnums.north);          System.out.println(ConstantsWithoutEnums.south);          System.out.println(ConstantsWithoutEnums.east);          System.out.println(ConstantsWithoutEnums.west);      }  } |

Above constatnts can be defined with enums as below,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | enum Directions  {      NORTH, SOUTH, EAST, WEST;  }    public class EnumsExample  {      public static void main(String[] args)      {          Directions d1 = Directions.EAST;          System.out.println(d1);            Directions d2 = Directions.NORTH;          System.out.println(d2);            System.out.println(Directions.SOUTH);            System.out.println(Directions.WEST);      }  } |

Let’s see some things-to-remember about java enums.

* Enums in java are declared with **enum** keyword and constants in enums must be valid java identifier. It is good practice to declare constants with UPPERCASE letters.

|  |  |
| --- | --- |
| 1  2  3  4 | enum Days  {      MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;  } |

* Duplicate enum constants are not allowed.

|  |  |
| --- | --- |
| 1  2  3  4 | enum Directions  {      NORTH, NORTH, SOUTH, EAST, WEST;  //Compile Time Error : Duplicate Constants  } |

* Every constant of enum is public, static and final by default. As every constant is static, they can be accessed directly using enum name.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | enum enums  {      A, B, C;  }    public class EnumsExample  {      public static void main(String[] args)      {          enums en = enums.A;   //accessing constant A through enum name            enums en1 = enums.B;  //accessing constant B through enum name            enums en2 = enums.C;  //accessing constant C through enum name      }  } |

* Enums can have any number of fields. methods and constructors and Each constant will have their own copy of fields and methods.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | enum enums  {      A, B, C;        int i;  //enums can have fields        private enums()      {          //enums can have Constructor      }        void methodOfEnum()      {          //enums can have methods      }  }    public class EnumsExample  {      public static void main(String[] args)      {          enums en = enums.A;          System.out.println(en.i);  //Constant A has field i          en.methodOfEnum();         //Constant A has methodOfEnum()            enums en1 = enums.B;          System.out.println(en1.i);  //Constant B has field i          en1.methodOfEnum();         //Constant B has methodOfEnum()            enums en2 = enums.C;          System.out.println(en2.i);   //Constant C has field i          en2.methodOfEnum();          //Constant C has methodOfEnum()      }  } |

* If enum has only constants, then semicolon (;) at the end of constant declaration is not mandatory. But, if enum has other members, then semicolon is mandatory.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | enum enums  {      A, B, C, D     //semicolon at the end of this statement is not mandatory  }    enum enums  {      A, B, C;    //semocolon at the end of this statement is mandatory, because it has other members        int i;  //enums has a field        void methodOfEnum()      {          //enums has a method      }  } |

* Every enum in java extends java.lang.Enum class. Enum class is an abstract class in java.lang package. As every enum extends Enum class, it should not extend any other class. Because, Multiple inheritance is not allowed in java. But enums can implement any number of interfaces.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | interface AnyInterface  {      abstract void methodOfInterface();  }    enum enums implements AnyInterface  {      A, B, C;        @Override      public void methodOfInterface()      {          //MethodOfInterface is implemented      }  } |

* Enums can be declared inside a class. If declared inside a class, they are static by default and can be accessed directly by Class name.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | class ClassContainingEnum  {      enum enums      {          A, B, C      }  }    public class MainClass  {      public static void main(String[] args)      {          System.out.println(ClassContainingEnum.enums.A);  //Accessing enums directly using class name      }  } |

* Enum constants can override generalized method defined in the enum body.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46 | enum enums  {      FIRST      {          @Override          void commonMethod()          {              System.out.println("Common method Overridden in FIRST");          }      },        SECOND      {          @Override          void commonMethod()          {              System.out.println("Common method Overridden in SECOND");          }      },        THIRD      {          @Override          void commonMethod()          {              System.out.println("Common method Overridden in THIRD");          }      };        void commonMethod()      {          System.out.println("Generalized method, Common to all constants");      }  }    public class EnumsExample  {      public static void main(String[] args)      {          enums.FIRST.commonMethod();     //Output : Common method Overridden in FIRST            enums.SECOND.commonMethod();    //Output : Common method Overridden in SECOND            enums.THIRD.commonMethod();     //Output : Common method Overridden in THIRD      }  } |

* Enum can have abstract method declared in it’s body provided each enum constants must implement it.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | enum enums  {      FIRST      {          @Override          void abstractMethod()          {              //Abstract Method Implemented          }      },        SECOND      {          @Override          void abstractMethod()          {              //Abstract Method Implemented          }      },        THIRD      {          @Override          void abstractMethod()          {              //Abstract Method Implemented          }      };        abstract void abstractMethod();  } |

* Enum Constants can have their own fields and method defined in their body, but these fields and methods are visible only within the constant body.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56 | enum enums  {      FIRST      {          int j = 10;   // Field of FIRST            void methodOfFirst()          {              System.out.println(j);  //Field j can be used within the constant body          }            @Override          void abstractMethod()          {              methodOfFirst();     //methodOfFirst() can be used within the constant body          }      },        SECOND      {          int k = 20;   //Field of SECOND            void methodOfSecond()          {              System.out.println(k);  //Field k can be used within the constant body          }            @Override          void abstractMethod()          {              methodOfSecond();     //methodOfSecond() can be used within the constant body          }      };        int i;   //this field is available in all constants        abstract void abstractMethod();  //this method is available in all constants  }    public class EnumsExample  {      public static void main(String[] args)      {          System.out.println(enums.FIRST.j);   //Compile time error : Field j is not visible here            enums.FIRST.methodOfFirst();  //Compile time error : methodOfFirst() is not visible here            enums.FIRST.abstractMethod();            System.out.println(enums.SECOND.k);   //Compile time error : Field k is not visible here            enums.SECOND.methodOfSecond();  //Compile time error : methodOfSecond() is not visible here            enums.SECOND.abstractMethod();      }  } |

* After observing all the above features of enums, we come to know that enums can have constuctors, fields and methods. Enums can implement interface. Enums extend Enum class. That means they have all features of classes. Therefore they are special type of classes. Moreover, after compilation, .class files are generated for all enums. Then what enum constants are?….  You can treat them as static inner classes of enums as they can be referred directly using enum name and they can hold fields and methods in them

# 3. VarArgs OR Variable Argument Method In Java:

If a method has VarArgs as an argument, then you can pass any number of arguments of same type to that method and the method is called Variable Argument Method. The VarArgs should have datatype, followed by 3 dots and an argument name. The syntax of VarArgs is as follows,

**DataType … ArgumentName**

Let’s start the discussion with simple program which calculates average of two, three and four numbers. Here is the code.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | public class VarArgsDemo  {      static double average(int a, int b)      {          //Average Of two numbers            return (a + b)/2;      }        static double average(int a, int b, int c)      {          //Average of three numbers            return (a + b + c)/3;      }        static double average(int a, int b, int c, int d)      {          //Average of four numbers            return (a + b + c +d)/4;      }        public static void main(String[] args)      {          System.out.println(average(10, 20));   //prints average of two numbers            System.out.println(average(10, 20, 30));  //prints average of three numbers            System.out.println(average(10, 20, 30, 40));  //prints average of four numbers      }  } |

This program can be used to calculate average of two, three and four numbers. What if user wants to calculate average of ‘n‘ numbers?. The soloution for this question is VarARgs or Variable Argument Method. To Variable argument method you can pass any number of arguments of same type. The above program can be written using VarArgs as below,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | public class VarArgsDemo  {      static double average(int... a)      {          //Average Of 'n' numbers            int n = a.length;            int sumOfNumbers = 0;            //Claculating sum of numbers          for(int i : a)          {              sumOfNumbers = sumOfNumbers + i;          }            return (sumOfNumbers/n);      }        public static void main(String[] args)      {          System.out.println(average(10, 20, 30, 40, 50));   //prints average of 5 numbers            System.out.println(average(10, 20, 30, 40, 50, 60, 70, 80));  //prints average of 8 numbers            System.out.println(average(10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110));  //prints average of 11 numbers      }  } |

In this program, Variable argument method is used to calculate average of different set of numbers. Now, user can calculate average of  ‘n‘ numbers. It improves the flexibility of the code.

## **Here are some points you should know about VarArgs or Variable argument method in java :**

* VarArgs internally behave like array. The best example for this is, you can replace String array argument of main method with VarArgs.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class VarArgsDemo  {      public static void main(String... args)      {          //main method taking VarArgs as an argument      }  } |

* Even constructors can have VarArgs as an argument.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public class VarArgsDemo  {      public VarArgsDemo(Double... d)      {          //Constructor having Variable arguments      }        public static void main(String[] args)      {          VarArgsDemo demo1 = new VarArgsDemo(10.2, 15.4);            VarArgsDemo demo2 = new VarArgsDemo(15.6, 23.5, 14.7, 16.9);      }  } |

* VarArgs, if used in the method, must be the last argument.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class VarArgsDemo  {       static void methodOne(String... strings, int a)       {            //Compile time error : VarArgs must be last parameter       }       static void methodTwo(int a, String... strings)       {            //This one is OK       }  } |

# 4. Wrapper classes:

Java is an object oriented language and java treats everything as an object. For example, A simple file (java.io.File), a Calendar (java.util.Calendar), a date (java.util.Date), an image (java.awt.Image), a color (java.awt.Color) are represented by the classes and their objects. Primitive data types like int, float, double, boolean, char etc. are also represented by the classes called **Wrapper Claases**. For every primitive data type, there is a Wrapper Class in java. All wrapper classes are placed in **java.lang package**. Below is the list of primitive data types and their corresponding wrapper classes.

|  |  |
| --- | --- |
| **Primitive Data Type** | **Corresponding Wrapper Class** |
| **byte** | java.lang.Byte |
| **short** | java.lang.Short |
| **int** | java.lang.Integer |
| **long** | java.lang.Long |
| **float** | java.lang.Float |
| **double** | java.lang.Double |
| **boolean** | java.lang.Boolean |
| **char** | java.lang.Character |

Wrapper classes are mainly used to wrap the primitive content into an object. This operation of wrapping primitive content into an object is called **boxing.**The reverse process i.e  unwrapping the object into corresponding primitive data is called **Unboxing**.

## **Boxing :**

Following example shows the boxing i.e wrapping primitive content into corresponding wrapper object.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | public class WrapperClasses  {      public static void main(String[] args)      {          byte b = 10;             //Primitive byte data          Byte B = new Byte(b);   //Wrapping primitive byte data into Byte Object            short s = 15;              //Primitive short data          Short S = new Short(s);   //Wrapping of primitive short data into Short Object            int i = 20;                   //Primitive int Data          Integer I = new Integer(i);   //Wrapping of primitive int data into Integer Object            long l = 25;            //Primitive long data          Long L = new Long(l);   //Wrapping primitive long data into Long Object            float f = 12;            //Primitive float data          Float F = new Float(f);  //Wrapping primitive float data into Float Object            double d = 18.58;           //Primitive double data          Double D = new Double(d);   //Wrapping primitive double data into Double Object            boolean bln = true;               //Primitive boolean data          Boolean BLN = new Boolean(bln);   //Wrapping primitive boolean data into Boolean Object            char c = 'C';                     //Primitive char data          Character C = new Character(c);   //Wrapping primitive char data into Character Object      }  } |

## **Auto-Boxing :**

From JDK 1.5 onwards, Auto-Boxing is introduced. According to this feature, you need not to explicitly wrap the primitive content into an object. Just assign primiive data to corresponding wrapper class reference variable, java automatically wraps primitive data into corresponding wrapper object.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | public class WrapperClasses  {      public static void main(String[] args)      {          byte b = 10;    //Primitive byte data          Byte B = b;    //Auto-Boxing of byte data            short s = 15;     //Primitive short data          Short S = s;     //Auto-Boxing of short data            int i = 20;       //Primitive int Data          Integer I = i;    //Auto-Boxing of int data            long l = 25;    //Primitive long data          Long L = l;     //Auto-Boxing of long data            float f = 12;     //Primitive float data          Float F = f;     //Auto-Boxing of float data            double d = 18.58;     //Primitive double data          Double D = d;        //Auto-Boxing of double data            boolean bln = true;    //Primitive boolean data          Boolean BLN = bln;     //Auto-Boxing of boolean data            char c = 'C';          //Primitive char data          Character C = c;     //Auto-Boxing of char data      }  } |

## **Unboxing :**

All wrapper classes have methods to unwrap the object to corresponding primitive data. Go through the following example, it shows unboxing i.e unwrapping wrapper object into corresponding primitive data.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | public class WrapperClasses  {      public static void main(String[] args)      {          Byte B = new Byte((byte) 10);   //Byte Object          byte b = B.byteValue();         //Unwrapping Byte object to byte data            Short S = new Short((short) 20);   //Short Object          short s = S.shortValue();          //Unwrapping Short object to short data            Integer I = new Integer(15);    //Integer Object          int i = I.intValue();           //Unwrapping Integer object to int data            Long L = new Long(50);     //Long Object          long l = L.longValue();    //Unwrapping Long object to long data            Float F = new Float(20);      //Float Object          float f = F.floatValue();    //Unwrapping Float object to float data            Double D = new Double(20.5);   //Double Object          double d = D.doubleValue();    //Unwrapping Double object to double data            Boolean BLN = new Boolean(true);      //Boolean Object          boolean bln = BLN.booleanValue();    //Unwrapping Boolean object to boolean data            Character C = new Character('C');    //Character Object          char c = C.charValue();              //Unwrapping Character object to char data      }  } |

## **Auto-Unboxing :**

From JDK 1.5 onwards, Auto-Unboxing is introduced. According to this feature, you need not to call method of wrapper class to unbox the wrapper object. Java implicitly converts wrapper object to corresponding primitive data if you assign wrapper object to primitive type variable.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | public class WrapperClasses  {      public static void main(String[] args)      {          Byte B = new Byte((byte) 10);   //Byte Object          byte b = B;                    //Auto-Unboxing of Byte Object            Short S = new Short((short) 20);   //Short Object          short s = S;                      //Auto-Unboxing of Short Object            Integer I = new Integer(15);    //Integer Object          int i = I;                     //Auto-Unboxing of Integer Object            Long L = new Long(50);     //Long Object          long l = L;               //Auto-Unboxing of Long Object            Float F = new Float(20);      //Float Object          float f = F;                 //Auto-Unboxing of Float Object            Double D = new Double(20.5);   //Double Object          double d = D;                 //Auto-Unboxing of Double Object            Boolean BLN = new Boolean(true);      //Boolean Object          boolean bln = BLN;                   //Auto-Unboxing of Boolean Object            Character C = new Character('C');    //Character Object          char c = C;                         //Auto-Unboxing of Character Object      }  } |

# 5. Constructors And Parsing Methods Of Wrapper Classes In Java:

## **Constructors Of Wrapper Classes In Java:**

Every wrapper class in java has two constructors,

1. First constructor takes corresponding primitive data as an argument
2. Second constructor takes string as an argument.

## **Notes :**

* The string passed to second constructor should be parse-able to number , otherwise you will get run time NumberFormatException.
* Wrapper Class Character has only one constructor which takes char type as an argument. It doesn’t have a constructor which takes String as an argument. Because, String can not be converted into Character.
* Wrapper class Float has three constructors. The third constructor takes double type as an argument.

Following example shows constructors in wrapper classes.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | public class WrapperClasses  {      public static void main(String[] args)      {          Byte B1 = new Byte((byte) 10);     //Constructor which takes byte value as an argument          Byte B2 = new Byte("10");          //Constructor which takes String as an argument            //Byte B3 = new Byte("abc");      //Run Time Error : NumberFormatException            //Because, String abc can not be parse-able to byte            Short S1 = new Short((short) 20);   //Constructor which takes short value as an argument          Short S2 = new Short("10");         //Constructor which takes String as an argument            Integer I1 = new Integer(30);     //Constructor which takes int value as an argument          Integer I2 = new Integer("30");   //Constructor which takes String as an argument            Long L1 = new Long(40);      //Constructor which takes long value as an argument          Long L2 = new Long("40");    //Constructor which takes String as an argument            Float F1 = new Float(12.2f);      //Constructor which takes float value as an argument          Float F2 = new Float("15.6");   //Constructor which takes String as an argument          Float F3 = new Float(15.6d);    //Constructor which takes double value as an argument            Double D1 = new Double(17.8d);       //Constructor which takes double value as an argument          Double D2 = new Double("17.8");     //Constructor which takes String as an argument            Boolean BLN1 = new Boolean(false);       //Constructor which takes boolean value as an argument          Boolean BLN2 = new Boolean("true");      //Constructor which takes String as an argument            Character C1 = new Character('D');      //Constructor which takes char value as an argument          Character C2 = new Character("abc");    //Compile time error : String abc can not be converted to character      }  } |

## **Important Note :**

If you pass a string other than true or false to the second constructor of Boolean wrapper class, the object is initialized with false.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public class WrapperClasses  {      public static void main(String[] args)      {          Boolean BLN1 = new Boolean("true");    //passed string "true"            System.out.println(BLN1);       //output : true            Boolean BLN2 = new Boolean("false");   //passed string "false"            System.out.println(BLN2);       //output : false            Boolean BLN3 = new Boolean("abc");     //passed string "abc"            System.out.println(BLN3);       //output : false      }  } |

## **Parsing Methods Of Wrapper Classes In Java :**

All wrapper classes in java have methods to parse the given string to corresponding primitive data provided the string should be parse-able. If the string is not parse-able, you will get NumberFormatException. All parsing methods of wrapper classes are static i.e you can refer them directly using class name.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | public class WrapperClasses  {      public static void main(String[] args)      {          byte b = Byte.parseByte("10");          System.out.println(b);               //Output : 10            short s = Short.parseShort("25");          System.out.println(s);              //Output : 25            int i = Integer.parseInt("123");          System.out.println(i);              //Output : 123            long l = Long.parseLong("100");          System.out.println(l);             //Output : 100            float f = Float.parseFloat("12.35");          System.out.println(f);            //Output : 12.35            double d = Double.parseDouble("12.87");          System.out.println(d);            //Output : 12.87            boolean bln = Boolean.parseBoolean("true");          System.out.println(bln);                   //Output : true            boolean bln1 = Boolean.parseBoolean("abc");          System.out.println(bln1);                 //Output : false            char c = Character.parseChar("abc");     //compile time error            //parseChar() is not defined for Character wrapper class      }  } |

# 6. ValueOf() Method Of Wrapper Classes In Java:

Every wrapper class in java has 3 forms of valueOf method. They are,

1. A valueOf() method which takes primitive type as an argument
2. A valueOf() method which takes String type as an argument
3. A valueOf() method which takes two arguments. One is String type and another one is int type.

All these forms are public and static. The valueOf() methods are mainly used to wrap or box the primitive content into wrapper class objects.

Let’s discuss these three forms in detail.

## **A valueOf() method with primitive type as an argument :**

This form of valueOf method takes primitive type data as an argument and returns corresponding wrapper class object. The template of this form looks like :

**public static return\_type valueOf ( primitive\_type )**

where return\_type is any Wrapper Class.

Following example shows the usage of this form of valueOf() method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | public class WrapperClasses  {      public static void main(String[] args)      {          Byte B = Byte.valueOf((byte) 123);          System.out.println(B);              //Output : 123            Short S = Short.valueOf((short) 25);          System.out.println(S);             //Output : 25            Integer I = Integer.valueOf(46);          System.out.println(I);            //Output : 46            Long L = Long.valueOf(235);          System.out.println(L);            //Output : 235            Float F = Float.valueOf(23.5f);          System.out.println(F);           //Output : 23.5            Double D = Double.valueOf(15.4d);          System.out.println(D);           //Output : 15.4            Boolean BLN = Boolean.valueOf(true);          System.out.println(BLN);        //Output : true            BLN = Boolean.valueOf(false);          System.out.println(BLN);        //Output : false            Character C = Character.valueOf('C');          System.out.println(C);          //Output : C      }  } |

## **valueOf() Method with string as an argument :**

This form of valueOf method takes string as an argument and returns corresponding wrapper class object. It throws NumberFormatException, if string is not a valid numeric value. **Character wrapper class doesn’t have this method as string can not be converted to character.**The template of this form looks like :

**public static return\_type valueOf(String s) throws NumberFormatException**

where return\_type is any wrapper class.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | public class WrapperClasses  {      public static void main(String[] args)      {          Byte B = Byte.valueOf("123");          System.out.println(B);              //Output : 123            Short S = Short.valueOf("25");          System.out.println(S);             //Output : 25            Integer I = Integer.valueOf("46");          System.out.println(I);            //Output : 46            Long L = Long.valueOf("235");          System.out.println(L);            //Output : 235            Float F = Float.valueOf("23.5");          System.out.println(F);           //Output : 23.5            Double D = Double.valueOf("15.4");          System.out.println(D);           //Output : 15.4            Boolean BLN = Boolean.valueOf("true");          System.out.println(BLN);        //Output : true            BLN = Boolean.valueOf("false");          System.out.println(BLN);        //Output : false            BLN = Boolean.valueOf("abc");          System.out.println(BLN);       //Output : false            //Character C = Character.valueOf("C");    //Compile Time Error : Character Class doesn't have this method      }  } |

## **valueOf() Method with string and int as an arguments :**

This form of valueOf method takes two arguments. One is String type which holds valid numeric value to be converted into wrapper class object and second argument is int type which indicates the radix or base of that numeric value. This form also throws NumberFormatException if String is not a valid numeric value. **This method is available only in Byte, Short, Integer and Long wrapper classes.** The template of this form is,

**public static return\_type valueOf(String s, int radix) throws NumberFormatException**

where return\_type is any wrapper class.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | public class WrapperClasses  {      public static void main(String[] args)      {          Byte B = Byte.valueOf("10110", 2);  //A number with base 2 is converted into decimal value          System.out.println(B);              //Output : 22            B = Byte.valueOf("43", 5);         //A number with base 5 is converted into decimal value          System.out.println(B);             //Output : 23            Short S = Short.valueOf("12043", 8);     //A number with base 8 is converted into decimal value          System.out.println(S);                  //Output : 5155            S = Short.valueOf("4751", 10);      //A number with base 10 is converted into decimal value          System.out.println(S);              //Output : 4751            Integer I = Integer.valueOf("4673AB", 12);    //A number with base 12 is converted into decimal value          System.out.println(I);            //Output : 1132403            I = Integer.valueOf("6489CDF", 16);     //A number with base 16 is converted into decimal value          System.out.println(I);                  //Output : 105422047            Long L = Long.valueOf("GHFTDJ", 36);    //A number with base 36 is converted into decimal value          System.out.println(L);                 //Output : 996750199            L = Long.valueOf("DFGCHJ", 36);      //A number with base 36 is converted into decimal value          System.out.println(L);               //Output : 812017207      }  } |

Following example shows some cases where you may get NumberFornatException.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | public class WrapperClasses  {      public static void main(String[] args)      {          Byte B = Byte.valueOf("1012", 2);            //NumberFormatException : Because, Number with base 2 must contain only 0 and 1.            Byte B = Byte.valueOf("12043", 5);            //NumberFormatException : Value is out of range for Byte Type.            Short S = Short.valueOf("12043", 5);  //No Run time error : value is within the range of Short          System.out.println(S);                //Output : 898            Integer I = Integer.valueOf("1891BGH", 16);            //NumberFormatException : Because, Number with base 16 must contain any digits from 0 to 9          //and any character in A, B, C, D, E and F.            Long L = Long.valueOf("GHJBDFR", 36);            //No Run Time error : A number with base 36 can contain any digits and any alphabets from A to Z          System.out.println(L);     //Output : 35888885703      }  } |

# 7. Number Class:

Number Class in java is an abstract class. It is placed in **java.lang package**. It has four abstract methods and two concrete methods. The definition of Number Class Looks like this,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | public abstract class Number extends Object implements Serializable  {       //Four Abstract Methods       public abstract int intValue();       public abstract long longValue();       public abstract float floatValue();       public abstract double doubleValue();         //Two Concrete Methods       public byte byteValue()       {            return (byte)intValue();       }         public short shortValue()       {            return (short)intValue();       }  } |

You can notice that, even concrete methods byteValue() and shortValue() call abstract intValue() method from their body. That means, to use these two methods, abstract intValue() method must be implemented.

All wrapper classes which represent numeric values i.e Byte, Short, Integer, Long, Float and Double are sub classes of Number class and they implement all four abstract methods.

Let’s see these methods one by one.

* **intValue() Method :**

This method returns int value of the specified object. It may involve truncation of the value.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | public class WrapperClasses  {      public static void main(String[] args)      {          Byte B = new Byte("55");          int i = B.intValue();          System.out.println(i);      //Output : 55            Short S = new Short("23");          i = S.intValue();          System.out.println(i);     //Output : 23            Integer I = new Integer("56");          i = I.intValue();          System.out.println(i);    //Output : 56            Long L = new Long("125");          i = L.intValue();          System.out.println(i);    //Output : 125            Float F = new Float("23.56");          i = F.intValue();          System.out.println(i);    //Output : 23  (Value is truncated)            Double D = new Double("521.56");          i = D.intValue();          System.out.println(i);   //Output : 521  (Value is truncated)      }  } |

* **longValue() Method :**

It returns long value of  the specified object. It may involve truncation of the value.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | public class WrapperClasses  {      public static void main(String[] args)      {          Byte B = new Byte("55");          long l = B.longValue();          System.out.println(l);      //Output : 55            Short S = new Short("23");          l = S.longValue();          System.out.println(l);     //Output : 23            Integer I = new Integer("56");          l = I.longValue();          System.out.println(l);    //Output : 56            Long L = new Long("125");          l = L.longValue();          System.out.println(l);    //Output : 125            Float F = new Float("23.56");          l = F.longValue();          System.out.println(l);    //Output : 23  (Value is truncated)            Double D = new Double("521.56");          l = D.longValue();          System.out.println(l);   //Output : 521  (Value is truncated)      }  } |

* **floatValue() Method :**

It returns float value of the specified object.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | public class WrapperClasses  {      public static void main(String[] args)      {          Byte B = new Byte("55");          float f = B.floatValue();          System.out.println(f);      //Output : 55.0            Short S = new Short("23");          f = S.floatValue();          System.out.println(f);     //Output : 23.0            Integer I = new Integer("56");          f = I.floatValue();          System.out.println(f);    //Output : 56.0            Long L = new Long("125");          f = L.floatValue();          System.out.println(f);    //Output : 125.0            Float F = new Float("23.56");          f = F.floatValue();          System.out.println(f);    //Output : 23.56            Double D = new Double("521.56");          f = D.floatValue();          System.out.println(f);   //Output : 521.56      }  } |

* **doubleValue() Method :**

It returns double value of the specified object.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | public class WrapperClasses  {      public static void main(String[] args)      {          Byte B = new Byte("55");          double d = B.doubleValue();          System.out.println(d);      //Output : 55.0            Short S = new Short("23");          d = S.floatValue();          System.out.println(d);     //Output : 23.0            Integer I = new Integer("56");          d = I.floatValue();          System.out.println(d);    //Output : 56.0            Long L = new Long("125");          d = L.floatValue();          System.out.println(d);    //Output : 125.0            Float F = new Float("23.56");          d = F.floatValue();          System.out.println(d);    //Output : 23.559999465942383            Double D = new Double("521.56");          d = D.floatValue();          System.out.println(d);   //Output : 521.5599975585938      }  } |

* **byteValue() Method :**

It returns byte value of the specified object. It may involve truncation.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | public class WrapperClasses  {      public static void main(String[] args)      {          Byte B = new Byte("55");          byte b = B.byteValue();          System.out.println(b);      //Output : 55            Short S = new Short("23");          b = S.byteValue();          System.out.println(b);     //Output : 23            Integer I = new Integer("56");          b = I.byteValue();          System.out.println(b);    //Output : 56            Long L = new Long("125");          b = L.byteValue();          System.out.println(b);    //Output : 125            Float F = new Float("23.56");          b = F.byteValue();          System.out.println(b);    //Output : 23    (Value is truncated)            Double D = new Double("521.56");          b = D.byteValue();          System.out.println(b);   //Output : 9      }  } |

* **shortValue() Method :**

It returns short value of specified object. It may involve truncation.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | public class WrapperClasses  {      public static void main(String[] args)      {          Byte B = new Byte("55");          short s = B.shortValue();          System.out.println(s);      //Output : 55            Short S = new Short("23");          s = S.shortValue();          System.out.println(s);     //Output : 23            Integer I = new Integer("56");          s = I.shortValue();          System.out.println(s);    //Output : 56            Long L = new Long("125");          s = L.shortValue();          System.out.println(s);    //Output : 125            Float F = new Float("23.56");          s = F.shortValue();          System.out.println(s);    //Output : 23     (value is truncated)            Double D = new Double("521.56");          s = D.shortValue();          System.out.println(s);   //Output : 521      (value is truncated)      }  } |

# 8. Auto-Widening Vs Auto-Boxing Vs Auto-UpCasting In Java:

Just go through the following example,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | public class WrapperClasses  {      static void overloadedMethod(Integer I)      {          System.out.println("Integer Wrapper Class Type");      }        static void overloadedMethod(long l)      {          System.out.println("long primitive type");      }        public static void main(String[] args)      {          int i = 21;            overloadedMethod(i);      }  } |

In the above example, ‘overloadedMethod’ is overloaded. One method takes Integer wrapper class type as an argument and another method takes primitive long type as an argument. In the main method, we are calling this ‘overloadedMethod’ by passing primitive int type as an argument. When you run this program, you will get “long primitive type” as output. That means, auto-widening is happening not auto-boxing.

Now, make little modification to the above example. Change the argument of second method from primitive long type to Long wrapper class type.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | public class WrapperClasses  {      static void overloadedMethod(Integer I)      {          System.out.println("Integer Wrapper Class Type");      }        static void overloadedMethod(Long L)      {          System.out.println("Long Wrapper Class Type");      }        public static void main(String[] args)      {          int i = 21;            overloadedMethod(i);      }  } |

Now run this program. you will get “Integer Wrapper Class Type” as output. That means auto-boxing is happening.

Now, make one more modification to the above program. Change the argument of first method from Integer Wrapper Class Type to Double Wrapper Class Type.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | public class WrapperClasses  {      static void overloadedMethod(Double D)      {          System.out.println("Double Wrapper Class Type");      }        static void overloadedMethod(Long L)      {          System.out.println("Long Wrapper Class Type");      }        public static void main(String[] args)      {          int i = 21;            overloadedMethod(i);   //compile time error      }  } |

Above example gives compile time error. Because, there is no method definition which takes int type as an argument. Primitive int type can be auto-widened to big sized primitive types or can be auto-boxed to Integer wrapper class type but can not be converted into Double or Long wrapper class type.

Now, add one more overloadedMethod which takes Number Class type as an argument to the above class.

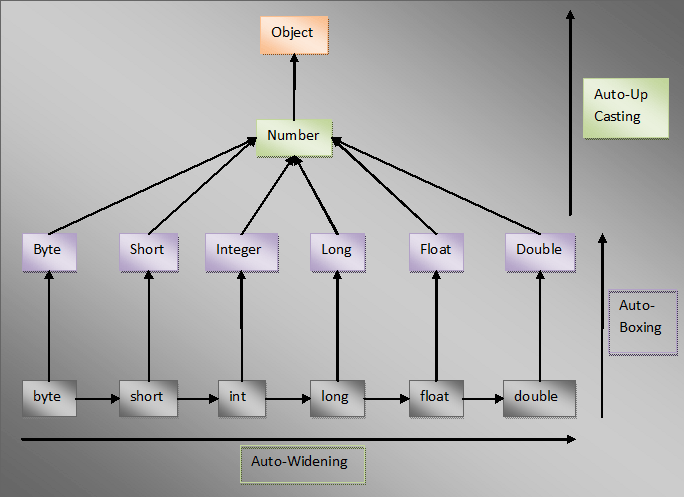
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | public class WrapperClasses  {      static void overloadedMethod(Number N)      {          System.out.println("Number Class Type");      }        static void overloadedMethod(Double D)      {          System.out.println("Double Wrapper Class Type");      }        static void overloadedMethod(Long L)      {          System.out.println("Long Wrapper Class Type");      }        public static void main(String[] args)      {          int i = 21;            overloadedMethod(i);      }  } |

Now run this program, you will get “Number Class Type” as output. What happened here is, internally primitive int type is auto-boxed to Integer type and Integer type is auto-UpCasted to Number type as Integer wrapper class is a sub class of Number class.

Above examples can be summarized like below,

* If you are passing primitive data type as an argument to the method call, compiler first checks for a method definition which takes **same data type** as an argument.
* If such method does not exist, then it checks for the method definition which takes big sized primitive data type than passed data type. i.e It tries to perform **auto-widening conversion** of passed data type.
* If auto-widening conversion is not possible, then it checks for method definition which takes corresponding wrapper class type as an argument. i.e It tries to perform **auto-boxing conversion**.
* If such method does not exist, then it checks for the method which takes super class type (Number or Object type) as an argument.
* If such method also does not exist, then compiler gives compile time error.

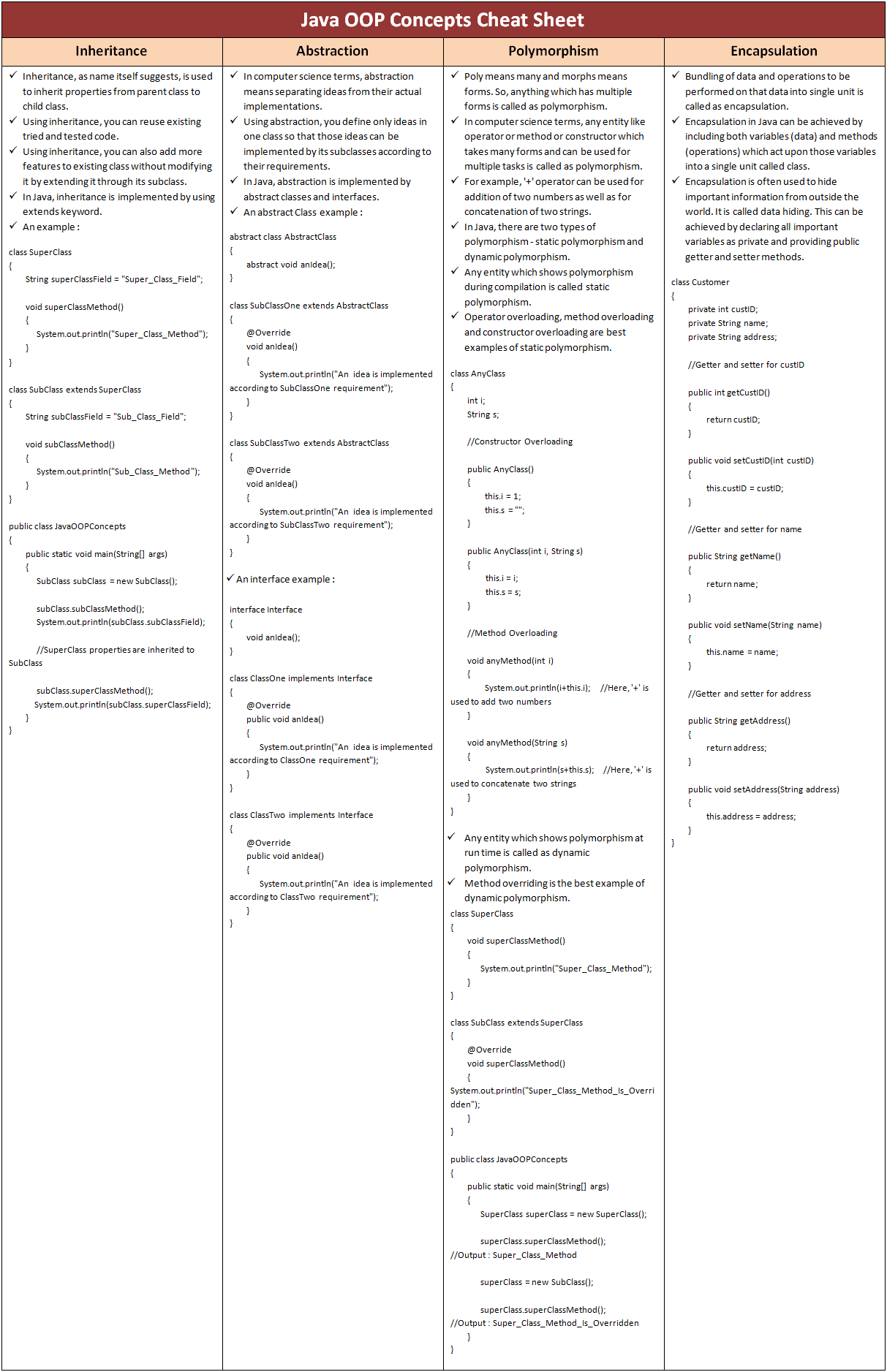
It can be diagrammatically represented as,



# Java OOP Concepts Cheat Sheet:

Java is an object oriented programming language. Object oriented programming model visualizes everything as object. An object, in object oriented programming concept, is a real world entity which has both state and behavior. A class is a template for same type of objects. For example, A Car is a class which acts as a template for car1, car2, car3… objects. The whole object oriented programming paradigm or simply OOP model is based on four fundamental concepts. They are – Inheritance, Abstraction, Polymorphism and Encapsulation. These are often called as four main pillars of the object oriented programming model. Let’s see these OOP concepts one by one and how they are implemented in Java.

Below is the Java OOP Concepts Cheat Sheet. You can download it and refer whenever required.

[](https://i0.wp.com/javaconceptoftheday.com/wp-content/uploads/2022/08/Java_OOP_Concepts_Cheat_Sheet.png?ssl=1)

#### **1) Inheritance**

Inheritance, as name itself suggests, is used to inherit properties from parent class to child class. Using inheritance, you can reuse existing tried and tested code or else you can add more features to existing class without modifying it by extending it through its subclass.

In Java, inheritance is implemented by using extends keyword.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | class SuperClass  {      String superClassField = "Super\_Class\_Field";        void superClassMethod()      {          System.out.println("Super\_Class\_Method");      }  }    class SubClass extends SuperClass  {      String subClassField = "Sub\_Class\_Field";        void subClassMethod()      {          System.out.println("Sub\_Class\_Method");      }  }    public class JavaOOPConcepts  {      public static void main(String[] args)      {          SubClass subClass = new SubClass();            subClass.subClassMethod();          System.out.println(subClass.subClassField);            //SuperClass properties are inherited to SubClass            subClass.superClassMethod();          System.out.println(subClass.superClassField);        }  } |

In the above program, SubClass is extending SuperClass. Hence, SubClass will have properties inherited from SuperClass along with its own properties. You can say that either we are reusing properties of SuperClass in SubClass or we are adding additional features to SuperClass by extending it through SubClass.

#### **2) Abstraction**

In computer science terms, abstraction means separating ideas from their actual implementations. Using abstraction, you define only ideas in one class so that those ideas can be implemented by its subclasses according to their requirements.

In Java, abstraction is implemented by abstract classes and interfaces.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | abstract class AbstractClass  {      abstract void anIdea();  }    class SubClassOne extends AbstractClass  {      @Override      void anIdea()      {          System.out.println("An idea is implemented according to SubClassOne requirement");      }  }    class SubClassTwo extends AbstractClass  {      @Override      void anIdea()      {          System.out.println("An idea is implemented according to SubClassTwo requirement");      }  } |

In the above code, AbstractClass has only idea and that idea is implemented by SubClassOne and SubClassTwo according to their requirements.

You can also use interfaces in Java to implement abstraction.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | interface Interface  {      void anIdea();  }    class ClassOne implements Interface  {      @Override      public void anIdea()      {          System.out.println("An idea is implemented according to ClassOne requirement");      }  }    class ClassTwo implements Interface  {      @Override      public void anIdea()      {          System.out.println("An idea is implemented according to ClassTwo requirement");      }  } |

In the above code snippet, Interface has anIdea() which is implemented by ClassOne and ClassTwo according to their requirements.

Till Java 8, interfaces are allowed to have only abstract methods. From Java 8, they can also have concrete methods in the form of default and static methods thus reducing the gap between abstract classes and interfaces. See the differences between abstract classes Vs Interfaces after Java 8 [here](https://javaconceptoftheday.com/interface-vs-abstract-class-after-java-8/).

#### **3) Polymorphism**

Poly means many and morphs means forms. So, anything which has multiple forms is called as polymorphism. In computer science terms, any entity like operator or method or constructor which takes many forms and can be used for multiple tasks is called as polymorphism. For example, ‘+’ operator can be used for addition of two numbers as well as for concatenation of two strings.

In Java, there are two types of polymorphism – static polymorphism and dynamic polymorphism.

**Static Polymorphism :**

Any entity which shows polymorphism during compilation is called static polymorphism. In this type of polymorphism, object used is determined during compilation itself. Hence, it is also called as static binding or early binding.

Operator overloading, method overloading and constructor overloading are best examples of static polymorphism.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | class AnyClass  {      int i;      String s;        //Constructor Overloading        public AnyClass()      {          this.i = 1;          this.s = "";      }        public AnyClass(int i, String s)      {          this.i = i;          this.s = s;      }        //Method Overloading        void anyMethod(int i)      {          System.out.println(i+this.i);   //Here, '+' is used to add two numbers      }        void anyMethod(String s)      {          System.out.println(s+this.s);   //Here, '+' is used to concatenate two strings      }  } |

**Dynamic Polymorphism :**

Any entity which shows polymorphism at run time is called as dynamic polymorphism. It is also called as late binding or dynamic binding as object used is determined at run time. Method overriding shows dynamic polymorphism.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | class SuperClass  {      void superClassMethod()      {          System.out.println("Super\_Class\_Method");      }  }    class SubClass extends SuperClass  {      @Override      void superClassMethod()      {          System.out.println("Super\_Class\_Method\_Is\_Overridden");      }  }    public class JavaOOPConcepts  {      public static void main(String[] args)      {          SuperClass superClass = new SuperClass();            superClass.superClassMethod();        //Output : Super\_Class\_Method            superClass = new SubClass();            superClass.superClassMethod();        //Output : Super\_Class\_Method\_Is\_Overridden        }  } |

#### **4) Encapsulation**

Bundling of data and operations to be performed on that data into single unit is called as encapsulation. Encapsulation in Java can be achieved by including both variables (data) and methods (operations) which act upon those variables into a single unit called class.

Encapsulation is often used to hide important information from outside the world. It is called data hiding. This can be achieved by declaring all important variables as private and providing public getter and setter methods.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | class Customer  {      private int custID;      private String name;      private String address;        //Getter and setter for custID        public int getCustID()      {          return custID;      }        public void setCustID(int custID)      {          this.custID = custID;      }        //Getter and setter for name        public String getName() {          return name;      }        public void setName(String name)      {          this.name = name;      }        //Getter and setter for address        public String getAddress()      {          return address;      }        public void setAddress(String address)      {          this.address = address;      }  } |

# Exception Handling

# 1. Exception Handling in Java:

An exception is an abnormal condition which occurs during run time and disrupts the normal flow of the program.  This exception must be handled to maintain the normal flow of the program. If this exception is not handled properly, the rest of the program will not be executed. Hence, causing the abrupt termination of the program. Therefore, you must handle the exceptions for the smooth flow of the program. To handle the run time exceptions, one mechanism is provided in java and it is called exception handling.

Let’s see some of the exceptions and their handling using **try and catch** blocks.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class ExceptionHandling  {      public static void main(String[] args)      {          System.out.println("This statement will be executed");            Integer I = new Integer("abc");  //This statement throws NumberFormatException            System.out.println("This statement will not be executed");      }  } |

In the above example, Line 7 throws NumberFormatException. This causes the abrupt termination of the program i.e remaining statement (Line 8) will not be executed. To make it run normally, enclose the statement which is throwing an exception in try-catch blocks. This is shown below,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | public class ExceptionHandling  {      public static void main(String[] args)      {          System.out.println("This statement will be executed");            try          {              Integer I = new Integer("abc");  //This statement throws NumberFormatException          }          catch (Exception e)          {              System.out.println("exception caught");          }            System.out.println("Now, This statement will also be executed");      }  } |

Below example throws ArithmaticException.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class ExceptionHandling  {      public static void main(String[] args)      {          System.out.println("This statement will be executed");            int i = 1000/0;    //This statement throws ArithmaticException : / by zero            System.out.println("This statement will not be executed");      }  } |

Below example shows how ArithmaticException can be handled using try-catch blocks.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | public class ExceptionHandling  {      public static void main(String[] args)      {          System.out.println("This statement will be executed");            try          {              int i = 1000/0;    //This statement throws ArithmaticException : / by zero          }          catch (Exception e)          {              System.out.println("Exception Caught");          }            System.out.println("Now, This statement will also be executed");      }  } |

This example throws ArrayIndexOutOfBoundsException.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class ExceptionHandling  {      public static void main(String[] args)      {          System.out.println("This statement will be executed");            String s = args[1];    //This statement throws ArrayIndexOutOfBoundsException            System.out.println("This statement will not be executed");      }  } |

and this can be handled using try-catch blocks.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | public class ExceptionHandling  {      public static void main(String[] args)      {          System.out.println("This statement will be executed");            try          {              String s = args[1];    //This statement throws ArrayIndexOutOfBoundsException          }          catch (Exception e)          {              System.out.println("Exception Caught");          }            System.out.println("Now, This statement will also be executed");      }  } |

Below example throws ClassCastException.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class ExceptionHandling  {      public static void main(String[] args)      {          System.out.println("This statement will be executed");            Object o = new Object();            ExceptionHandling e = (ExceptionHandling) o;   //This statement throws ClassCastException            System.out.println("This statement will not be executed");      }  } |

Below example shows handling of ClassCastException using try-catch blocks.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | public class ExceptionHandling  {      public static void main(String[] args)      {          System.out.println("This statement will be executed");            Object o = new Object();            try          {              ExceptionHandling e = (ExceptionHandling) o;   //This statement throws ClassCastException          }          catch (Exception e)          {              System.out.println("Exception Caught");          }            System.out.println("Now, This statement will also be executed");      }  } |

Below example throws NullPointerException.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class ExceptionHandling  {      public static void main(String[] args)      {          System.out.println("This statement will be executed");            String s = null;            System.out.println(s.length());  //This statement throws NullPointerException            System.out.println("This statement will not be executed");      }  } |

and this example shows how NullPointerException can be handled using try-catch blocks.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | public class ExceptionHandling  {      public static void main(String[] args)      {          System.out.println("This statement will be executed");            String s = null;            try          {              System.out.println(s.length());  //This statement throws NullPointerException          }          catch (Exception e)          {              System.out.println("Exception Caught");          }            System.out.println("Now, This statement will also be executed");      }  } |

# 2. try, catch and finally blocks In Java:

Exception Handling in java is implemented using five keywords.

1) try       2)catch       3)finally       4)throw         5)throws

Today we will discuss about try, catch and finally keywords. Remaining keywords will be discussed in subsequent concepts.

* try, catch and finally keywords are main fundamentals of exception handling in java. The syntax for using these three keywords is,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | try  {      //This is the try block      //In this block, keep those statements which may      //throw run time exceptions  }    catch(Exception e)  {      //This is the catch block.      //It takes one argument of type java.lang.Exception      //This block catches the exceptions thrown by try block  }    finally  {      //This is the finally block.  } |

**try block :**In try block, keep those statements which may throw exceptions during run time.

**catch block :** This block handles the exceptions thrown by try block. It takes one argument of type java.lang.Exception.

**finally block :** Whether exception is thrown or not and thrown exception is caught or not, this block will be always executed.

For example,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | public class ExceptionHandling  {      public static void main(String[] args)      {          String[] s = {"abc", "123", "xyz", "456"};   //String Array containing valid and invalid numeric values            for (int i = 0; i < s.length; i++)          {              try              {                  int intValue = Integer.parseInt(s[i]); //This statement may throw NumberFormatException              }              catch(NumberFormatException ex)              {                  System.out.println("The thrown NumberFormatException will be caught here");              }              finally              {                  System.out.println("This block is always executed");              }          }      }  } |

In the above example, A string array, containing valid and invalid numeric values, is iterated through for loop. Each element of an array is parsed to primitive int type. Element with valid numeric value is parsed without throwing an exception. Element with invalid numeric value can not be parsed to int type and it throws the NumberFormatException. This exception is caught in catch block making the flow of the program normal. finally block is executed for every iteration whether element is parsed or not.

* When a statement throws an exception in the try block, the remaining part of the try block will not be executed. Program control comes out of the try block and enters directly into catch block.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | public class ExceptionHandling  {      public static void main(String[] args)      {          try          {              int i = 10/0;           //This statement throws ArithmeticException                System.out.println("This statement will not be executed");          }          catch(Exception ex)          {              System.out.println("This block is executed immediately after an exception is thrown");          }          finally          {              System.out.println("This block is always executed");          }      }  } |

* try, catch and finally blocks form one unit. i.e You can’t keep other statements in between try, catch and finally blocks.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | public class ExceptionHandling  {      public static void main(String[] args)      {          System.out.println("You can keep any number of statements here");            try          {              int i = 10/0;           //This statement throws ArithmeticException                System.out.println("This statement will not be executed");          }            //You can't keep statements here            catch(ArithmeticException ex)          {              System.out.println("This block is executed immediately after an exception is thrown");          }            //You can't keep statements here            finally          {              System.out.println("This block is always executed");          }            System.out.println("You can keep any number of statements here");      }  } |

* You can display the description of an exception thrown using Exception object in the catch block.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | public class ExceptionHandling  {      public static void main(String[] args)      {          try          {              String s = null;                System.out.println(s.length());   //This statement throws NullPointerException                System.out.println("This statement will not be executed");          }          catch(Exception ex)          {              System.out.println(ex);    //Output : java.lang.NullPointerException                ex.printStackTrace();     //This prints stack trace of exception          }          finally          {              System.out.println("This block is always executed");          }      }  } |

# 3. Multiple Catch Blocks In Java:

We will discuss about Multiple Catch Blocks In Java

* In some cases, A single statement may throw more than one type of exception. In such cases, Java allows you to put more than one catch blocks. One catch block handles one type of exception.  When an exception is thrown by the try block, all the catch blocks are examined in the order they appear and one catch block which matches with exception thrown will be executed. After, executing catch block, program control comes out of try-catch unit.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | public class ExceptionHandling  {      public static void main(String[] args)      {          String[] s = {"abc", "123", null, "xyz"};   //String array containing one null object            for (int i = 0; i < 6; i++)          {              try              {                  int a = s[i].length() + Integer.parseInt(s[i]);                    //This statement may throw NumberFormatException, NullPointerException and ArrayIndexOutOfBoundsException                }                catch(NumberFormatException ex)              {                  System.out.println("NumberFormatException will be caught here");              }                catch (ArrayIndexOutOfBoundsException ex)              {                  System.out.println("ArrayIndexOutOfBoundsException will be caught here");              }                catch (NullPointerException ex)              {                  System.out.println("NullPointerException will be caught here");              }                System.out.println("After executing respective catch block, this statement will be executed");          }      }  } |

In the above example, a string array is iterated through for loop. First statement of try block (Line 11) may throw NumberFormatException because string array contain some non-numeric values or It may throw NullPointerException because string array contains one null object or it may throw ArrayIndexOutOfBoundsException because we are trying to iterate 6 elements, but actually string array contains only 4 elements. Depending upon the type of exception thrown, corresponding catch block will be executed.

* From Java 7 onward, there is one more way for handling multiple exceptions. Multiple exceptions thrown by the try block can be handled by a single catch block using **pipe (|) operator**. By using pipe operator, the above example can be written as,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | public class ExceptionHandling  {      public static void main(String[] args)      {          String[] s = {"abc", "123", null, "xyz"};   //String array containing one null object            for (int i = 0; i < 6; i++)          {              try              {                  int a = s[i].length() + Integer.parseInt(s[i]);                    //This statement may throw NumberFormatException, NullPointerException and ArrayIndexOutOfBoundsException              }                catch(NumberFormatException | NullPointerException | ArrayIndexOutOfBoundsException ex)              {                  System.out.println("Now, this block handles NumberFormatException, NullPointerException and ArrayIndexOutOfBoundsException");              }          }      }  } |

* java.lang.Exception is super class of all types of exception. (Types of exceptions will be discussed later). It handles all types of exceptions. In the above example, all catch blocks can be replaced by one catch block which handles all types of exceptions. This type of exception handling comes very handy when you are not sure about the types of exceptions your code may throw.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | public class ExceptionHandling  {      public static void main(String[] args)      {          String[] s = {"abc", "123", null, "xyz"};   //String array containing one null object            for (int i = 0; i < 6; i++)          {              try              {                  int a = s[i].length() + Integer.parseInt(s[i]);                    //This statement may throw NumberFormatException, NullPointerException and ArrayIndexOutOfBoundsException              }                catch(Exception ex)              {                  System.out.println("This block handles all types of exceptions");              }          }      }  } |

* The order of catch blocks should be from most specific to most general ones. i.e Sub classes of Exception must come first and super classes later. If you keep the super classes first and sub classes later, you will get compile time error : **Unreachable Catch Block**.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | public class ExceptionHandling  {      public static void main(String[] args)      {          try          {              int i = Integer.parseInt("abc");   //This statement throws NumberFormatException          }            catch(Exception ex)          {              System.out.println("This block handles all exception types");          }            catch(NumberFormatException ex)          {              //Compile time error              //This block becomes unreachable as              //exception is already handled by above catch block          }      }  } |

Below is the correct way to write above program.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | public class ExceptionHandling  {      public static void main(String[] args)      {          try          {              int i = Integer.parseInt("abc");   //This statement throws NumberFormatException          }            catch(NumberFormatException ex)          {              System.out.println("This block handles NumberFormatException");          }            catch(Exception ex)          {              System.out.println("This block handles all exception types");          }            catch (Throwable ex)          {              System.out.println("Throwable is super class of Exception");          }      }  } |

# 4. Nested try catch Blocks In Java:

In Java, try-catch blocks can be nested. i.e one try block can contain another try-catch block. The syntax for nesting try blocks is,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | try     //Outer try block  {      //Some Statements        try    //Inner try block      {          //Some Statements      }      catch (Exception ex)    //Inner catch block      {        }  }  catch(Exception ex)     //Outer catch block  {    } |

Let’s discuss some of the points about nested try catch blocks in java.

* Nested try blocks are useful when different statements of try block throw different types of exceptions.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | public class ExceptionHandling  {      public static void main(String[] args)      {          String[] s = {"abc", "123", null, "xyz"};   //String array containing one null object            for (int i = 0; i < s.length; i++)          {              try     //Outer try block              {                  int a = s[i].length();    //This statement may throw NullPointerException                    try    //Inner try block                  {                      a = Integer.parseInt(s[i]);    //This statement may throw NumberFormatException                  }                  catch (NumberFormatException ex)   //Inner catch block                  {                      System.out.println("NumberFormatException will be caught here");                  }              }              catch(NullPointerException ex)     //Outer catch block              {                  System.out.println("NullPointerException will be caught here");              }          }      }  } |

* try blocks can be nested at any level.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | public class ExceptionHandling  {      public static void main(String[] args)      {          String[] s = {"abc", "123", null, "xyz"};   //String array containing one null object            for (int i = 0; i < s.length; i++)          {              //First Level try-catch block              try              {                  int a = s[i].length();    //This statement may throw NullPointerException                    //second level try-catch block                  try                  {                      System.out.println(s[i+1]);   //This statement may throw ArrayIndexOutOfBoundsException                        //third level try-catch block                      try                      {                          a = Integer.parseInt(s[i]);    //This statement may throw NumberFormatException                      }                      catch (NumberFormatException e)                      {                          System.out.println("NumberFormatException will be caught here");                      }                  }                  catch (ArrayIndexOutOfBoundsException ex)                  {                      System.out.println("ArrayIndexOutOfBoundsException will be caught here");                  }              }              catch(NullPointerException ex)              {                  System.out.println("NullPointerException will be caught here");              }          }      }  } |

* If the exception thrown by the inner try block can not be caught by it’s catch block, then this exception is propagated to outer try blocks. Any one of the outer catch block should handle this exception otherwise program will terminate abruptly.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | public class ExceptionHandling  {      public static void main(String[] args)      {          String[] s = {"abc", "123", null, "xyz"};   //String array containing one null object            for (int i = 0; i < s.length; i++)          {              //First Level try-catch block              try              {                  int a = s[i].length();    //This statement may throw NullPointerException                    //second level try-catch block                  try                  {                      System.out.println(s[i+1]);   //This statement may throw ArrayIndexOutOfBoundsException                        //third level try-catch block                      try                      {                          a = Integer.parseInt(s[i]);    //This statement may throw NumberFormatException                      }                      catch (NullPointerException e)                      {                          System.out.println("NumberFormatException will not be caught here");                      }                  }                  catch (NumberFormatException ex)                  {                      System.out.println("NumberFormatException will be caught here");                  }              }              catch(Exception ex)              {                  System.out.println("This block catches all types of exceptions");              }          }      }  } |

* In the following example also, try-catch blocks are nested. main() method calls nestedTry() method. nestedTry() method has one try-catch block. First statement in try block throws NumberFormatException which is not handled by it’s catch block. So, It propagates to try-catch block of main method which handles this exeption.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | public class ExceptionHandling  {      public static void main(String[] args)      {          try          {              nestedTry();          }          catch(Exception ex)          {              System.out.println("NumberFormatException will be caught here");          }      }        static void nestedTry()      {          try          {              int i = Integer.parseInt("abc");    //This statement throws NumberFormatException          }          catch(NullPointerException ex)          {              System.out.println("NumberFormatException will not be caught here");          }      }  } |

* From the above examples, we come to know that exception thrown must be handled somewhere in the program. If it is not handled, then program will be terminated abruptly.

# 5. Return Value From try-catch-finally Blocks:

we will see some examples in which try-catch-finally blocks return a value.

* If method returns a value and also has try, catch and finally blocks in it, then following two rules need to follow.

1) If finally block returns a value then try and catch blocks may or may not return a value.

2) If finally block does not return a value then both try and catch blocks must return a value.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | public class ReturnValueFromTryCatchFinally  {      public static void main(String[] args)      {          System.out.println(methodReturningValue());      }        static int methodReturningValue()      {          try          {              //This block may or may not return a value as finally block is returning a value          }          catch (Exception e)          {              //This block may or may not return a value as finally block is returning a value          }          finally          {              return 20;          }      }  } |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | public class ReturnValueFromTryCatchFinally  {      public static void main(String[] args)      {          System.out.println(methodReturningValue());      }        static int methodReturningValue()      {          try          {              return 10;          }          catch (Exception e)          {              return 20;          }          finally          {              //Now, This block may or may not return a value              //as both try and catch blocks are returning a value          }      }  } |

* If try-catch-finally blocks are returning a value according to above rules, then you should not keep any statements after finally block. Because they become unreachable and in Java, Unreachable code gives compile time error.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | public class ReturnValueFromTryCatchFinally  {      public static void main(String[] args)      {          try          {              return;          }          catch (Exception e)          {              return;          }          finally          {              return;          }            System.out.println("Unreachable code");    //Compile Time Error : Unreachable Code      }  } |

* finally block overrides any return values from try and catch blocks.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | public class ReturnValueFromTryCatchFinally  {      public static void main(String[] args)      {          System.out.println(methodReturningValue());    //Output : 50      }        static int methodReturningValue()      {          try          {              return 10;          }          catch (Exception e)          {              return 20;          }          finally          {              return 50;    //This method returns 50 not 10 or 20          }      }  } |

* finally block will be always executed even though try and catch blocks are returning the control.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | public class ReturnValueFromTryCatchFinally  {      public static void main(String[] args)      {          System.out.println(methodReturningValue());    //Output : 10      }        static int methodReturningValue()      {          try          {              return 10;    //control will not be passed to main() method here          }          catch (Exception e)          {              return 20;    //Control will not be passed to main() method here          }          finally          {              System.out.println("finally block is always executed");                //Control will be passed to main() method after executing this block          }      }  } |

* Go through the following examples and try to analyse their output.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | public class ReturnValueFromTryCatchFinally  {      public static void main(String[] args)      {          System.out.println(methodReturningValue());      }        static String methodReturningValue()      {          String s = null;          try          {              s = "return value from try block";              return s;          }          catch (Exception e)          {              s = s + "return value from catch block";              return s;          }          finally          {              s = s + "return value from finally block";          }      }  } |

OUTPUT : return value from try block

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | public class ReturnValueFromTryCatchFinally  {      public static void main(String[] args)      {          System.out.println(methodReturningValue());      }        static int methodReturningValue()      {          int i = 0;            try          {              i = 1;              return i;          }          catch (Exception e)          {              i = 2;              return i;          }          finally          {              i = 3;          }      }  } |

OUTPUT : 1

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | public class ReturnValueFromTryCatchFinally  {      public static void main(String[] args)      {          System.out.println(methodReturningValue());      }        static int methodReturningValue()      {          int i = 0;            try          {              i = 1;              return i;          }          catch (Exception e)          {              i = 2;          }          finally          {              i = 3;          }            return i;      }  } |

OUTPUT : 1

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | public class ReturnValueFromTryCatchFinally  {      public static void main(String[] args)      {          System.out.println(methodReturningValue());      }        static int methodReturningValue()      {          try          {              int i = Integer.parseInt("123");              return 20;          }          finally          {              return 50;          }      }  } |

OUTPUT : 50

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | public class ReturnValueFromTryCatchFinally  {      public static void main(String[] args)      {          System.out.println(methodReturningValue());      }        static int methodReturningValue()      {          try          {              int i = Integer.parseInt("abc");   //This statement throws NumberFormatException              return 20;          }          finally          {              return 50;          }      }  } |

OUTPUT : 50

# 6. Hierarchy Of Exceptions In Java:

let’s see what is checked and unchecked exceptions in brief. (we will discuss about them in detail later).

## **Checked Exceptions**

Checked exceptions are known to compiler i.e they are the exceptions that are checked at compile time. Checked exceptions are also called compile time exceptions, because they can be known during compile time.

## **Unchecked Exceptions**

Unchecked exceptions are not known to compiler.  They are the exceptions that are not checked at compile time, because they occur only at run time.That’s why these exceptions are also called run time exceptions.

Now, come to hierarchy of exceptions in java.

## **java.lang.Throwable :**

java.lang.Throwable is the super class of all errors and exceptions in java. Throwable class extends java.lang.Object class. The only argument of catch block must be it’s type or it’s sub class type. You can check the documentation of Throwable class [here](http://docs.oracle.com/javase/7/docs/api/java/lang/Throwable.html). It has two sub classes.

1)java.lang.Error

2)java.lang.Exception

## **1) java.lang.Error :**

java.lang.Error is the super class for all types of errors in java. You can follow the documentation of Error class [here](http://docs.oracle.com/javase/7/docs/api/java/lang/Error.html). Some of the common errors are,

* [java.lang.VirtualMachineError](http://docs.oracle.com/javase/7/docs/api/java/lang/VirtualMachineError.html) : The most common virtualMachineErrors are [StackOverFlowError](http://docs.oracle.com/javase/7/docs/api/java/lang/StackOverflowError.html) and [OutOfMemoryError.](http://docs.oracle.com/javase/7/docs/api/java/lang/OutOfMemoryError.html)
* [java.lang.AssertionError](http://docs.oracle.com/javase/7/docs/api/java/lang/AssertionError.html)
* [java.lang.LinkageError](http://docs.oracle.com/javase/7/docs/api/java/lang/LinkageError.html)  : The common LinkageError are [NoClassDefFoundError](http://docs.oracle.com/javase/7/docs/api/java/lang/NoClassDefFoundError.html) and subclasses of [IncompatibleClassChangeError](http://docs.oracle.com/javase/7/docs/api/java/lang/IncompatibleClassChangeError.html). The most frequent IncompatibleClassChangeErrors are NoSuchMethodError, NoSuchFieldError, AbstractMethodError, IllegalAccessError and InstantiationError.

All sub classes of Error class are unchecked type of exceptions. i.e They occur during run time only.

## **2) java.lang.Exception :**

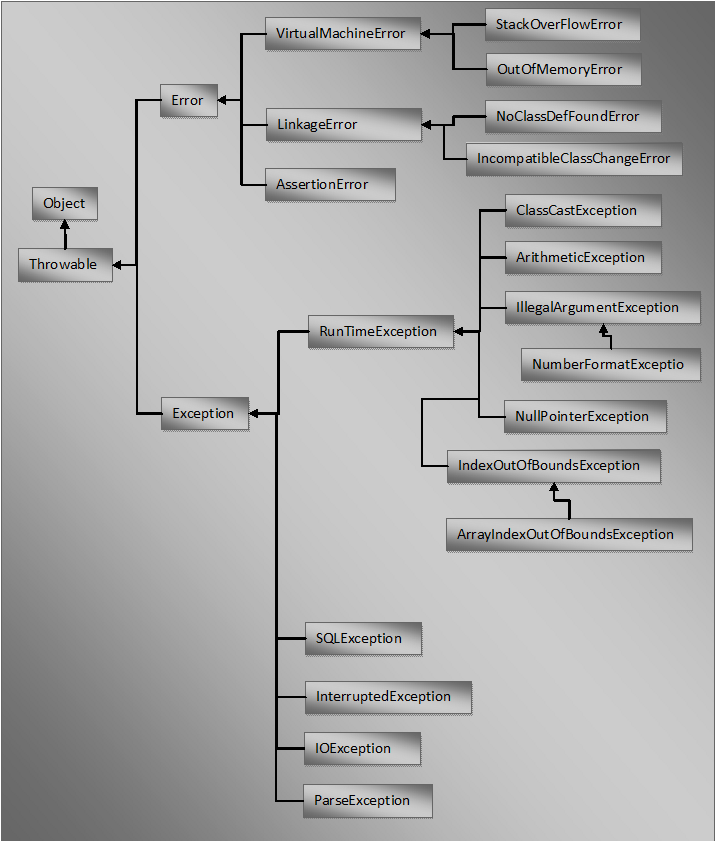
java.lang.Exception is the super class for all types of Exceptions in java. You can follow the documentation of Exception class [here](http://docs.oracle.com/javase/7/docs/api/java/lang/Exception.html). All sub classes of Exception class except sub classes of RunTimeException are checked type of exceptions. Some of the common sub classes of Exception are,

* [java.lang.RunTimeException](http://docs.oracle.com/javase/7/docs/api/java/lang/RuntimeException.html)

All sub classes of RunTimeException are unchecked type of exceptions. i.e They occur during run time only. Some common RunTimeException are [ArithmeticException](http://docs.oracle.com/javase/7/docs/api/java/lang/ArithmeticException.html), [NumberFormatException](http://docs.oracle.com/javase/7/docs/api/java/lang/NumberFormatException.html), [NullPointerException](http://docs.oracle.com/javase/7/docs/api/java/lang/NullPointerException.html), [ArrayIndexOutOfBoundsException](http://docs.oracle.com/javase/7/docs/api/java/lang/ArrayIndexOutOfBoundsException.html) and [ClassCastException](http://docs.oracle.com/javase/7/docs/api/java/lang/ClassCastException.html).

* [java.lang.InterruptedException](http://docs.oracle.com/javase/7/docs/api/java/lang/InterruptedException.html)
* [java.lang.IOException](http://docs.oracle.com/javase/7/docs/api/java/io/IOException.html)
* [java.lang.SQLException](http://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html)
* [java.lang.ParseException](http://docs.oracle.com/javase/7/docs/api/java/text/ParseException.html)

All above classes are placed in java.lang package. Click on class to follow the documentation of that class. The above hierarchy can be represented as,



# 7. Throwing And Re-Throwing An Exception In Java:

Till now, we are catching the exceptions which are thrown by Java Run Time System. Today, we will learn how to throw and re-throw exceptions explicitly.

## **Throwing An Exception :**

We all know that Throwable class is super class for all types of errors and exceptions. An object to this Throwable class or it’s sub classes can be created in two ways. First one is using an argument of catch block. In this way, Throwable object or object to it’s sub classes is implicitly created and thrown by java run time system. Second one is using new operator. In this way, Throwable object or object to it’s sub classes is explicitly created and thrown by the code.

An object to Throwable or to it’s sub classes can be explicitly created and thrown by using **throw** keyword. The syntax for using throw keyword is,

**throw InstanceOfThrowableType;**

where, InstanceOfThrowableType must be an object of type Throwable or subclass of Throwable.

Such explicitly thrown exception must be handled some where in the program, otherwise program will be terminated.

For example,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | public class ExceptionHandling  {      public static void main(String[] args)      {          methodWithThrow();      }        static void methodWithThrow()      {          try          {              NumberFormatException ex = new NumberFormatException();    //Creating an object to NumberFormatException explicitly                throw ex;        //throwing NumberFormatException object explicitly using throw keyword          }          catch(NumberFormatException ex)          {              System.out.println("explicitly thrown NumberFormatException object will be caught here");          }      }  } |

It is not compulsory that explicitly thrown exception must be handled by immediately following try-catch block. It can be handled by any one of it’s enclosing try-catch blocks.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | public class ExceptionHandling  {      public static void main(String[] args)      {          try          {              methodWithThrow();          }          catch(NumberFormatException ex)          {              System.out.println("NumberFormatException object thrown in methodWithThrow() method will be handled here");          }      }        static void methodWithThrow()      {          try          {              NumberFormatException ex = new NumberFormatException("This is an object of NumberFormatException");                throw ex;        //throwing NumberFormatException object explicitly using throw keyword          }          catch(ArithmeticException ex)          {              System.out.println("Explicitly thrown NumberFormatException object will not be caught here");          }      }  } |

## **Re-throwing An Exception :**

We all know that exceptions occurred in the try block are caught in catch block. Thus caught exceptions can be re-thrown using **throw** keyword. Re-thrown exception must be handled some where in the program, otherwise program will terminate abruptly. For example,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | public class ExceptionHandling  {      public static void main(String[] args)      {          try          {              methodWithThrow();          }          catch(NullPointerException ex)          {              System.out.println("NullPointerException Re-thrown in methodWithThrow() method will be handled here");          }      }        static void methodWithThrow()      {          try          {              String s = null;              System.out.println(s.length());   //This statement throws NullPointerException          }          catch(NullPointerException ex)          {              System.out.println("NullPointerException is caught here");                throw ex;     //Re-throwing NullPointerException          }      }  } |

# 8. throws Keyword In Java:

If a method is capable of throwing an exception that it could not handle, then it should specify that exception using throws keyword. It helps the callers of that method in handling that exception. The syntax for using throws keyword is,

|  |  |
| --- | --- |
| 1  2  3  4 | return\_type method\_name(parameter\_list) throws exception\_list  {       //some statements  } |

where, exception\_list is the list of exceptions that method may throw. Exceptions must be separated by commas.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | public class ExceptionHandling  {      public static void main(String[] args)      {          try          {              methodWithThrows();          }          catch(NullPointerException ex)          {              System.out.println(&quot;NullPointerException thrown by methodWithThrows() method will be caught here&quot;);          }      }        static void methodWithThrows() throws NullPointerException      {          String s = null;          System.out.println(s.length());   //This statement throws NullPointerException      }  } |

Let’s see some of the points-to-remember about throws keyword.

* Multiple exceptions can be declared using throws keyword separated by commas.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | public class ExceptionHandling  {      static void methodWithThrows() throws NumberFormatException, NullPointerException      {          int i = Integer.parseInt(&quot;abc&quot;);   //This statement throws NumberFormatException            String s = null;            System.out.println(s.length());    //This statement throws NullPointerException      }        public static void main(String[] args)      {          try          {              methodWithThrows();          }          catch(Exception ex)          {              System.out.println(&quot;This block can handle all types of exceptions&quot;);          }      }  } |

* The main use of throws keyword in java is that an exception can be propagated through method calls.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | public class ExceptionHandling  {      static void methodOne() throws NumberFormatException      {          int i = Integer.parseInt(&quot;abc&quot;);   //This statement throws NumberFormatException      }        static void methodTwo() throws NumberFormatException      {          methodOne();     //NumberFormatException is propagated here      }        static void methodThree() throws NumberFormatException      {          methodTwo();    //NumberFormatException is propagated here      }        public static void main(String[] args)      {          try          {              methodThree();          }          catch(NumberFormatException ex)          {              System.out.println(&quot;NumberFormatException will be caught here&quot;);          }      }  } |

* Even constructor can use throws keyword.For this, object creation statement must be enclosed in try-catch blocks.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | class A  {      int i;        public A(String s) throws NumberFormatException      {          i = Integer.parseInt(s);    //This statement throws NumberFormatException      }  }    public class ExceptionHandling  {      public static void main(String[] args)      {          try          {              A a = new A(&quot;abc&quot;);    //Object creation statement enclosed in try-catch block          }          catch (NumberFormatException ex)          {              System.out.println(&quot;NumberFormatException will be caught here&quot;);          }      }  } |

* When a method is throwing unchecked type of exceptions, then you need not to mention it using throws keyword. But for a method throwing checked type of exceptions, you must declare it with throws keyword or enclose the statement which is throwing an exception in try-catch block. (We will discuss about this in detail while covering checked and unchecked exceptions).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | public class ExceptionHandling  {      //method throwing Unchecked Exception declared without throws clause        static void methodThrowingUncheckedException()      {          int i = Integer.parseInt(&quot;abc&quot;);            //Above statement throws NumberFormatException which is unchecked type of exception      }        //method throwing checked Exception declared with throws clause        static void methodThrowingCheckedException() throws ClassNotFoundException      {          Class.forName(&quot;AnyClassName&quot;);            //Above statement throws ClassNotFoundException which is checked type of exception      }        public static void main(String[] args)      {          try          {              methodThrowingUncheckedException();          }          catch(NumberFormatException ex)          {              System.out.println(&quot;NumberFormatException will be caught here&quot;);          }            try          {              methodThrowingCheckedException();          }          catch (ClassNotFoundException e)          {              System.out.println(&quot;ClassNotFoundException will be caught here&quot;);          }      }  } |

# 9. Method Overriding With throws Clause:

Let’s discuss some rules need to follow when overriding a method with **throws** clause.

* If super class method is not throwing any exceptions, then it can be overrided with any unchecked type of exceptions, but can not be overrided with checked type of exceptions.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | class SuperClass  {      void methodOfSuperClass()      {          System.out.println("Super class method is not throwing any exceptions");      }  }    class SubClass extends SuperClass  {      @Override      void methodOfSuperClass() throws ArrayIndexOutOfBoundsException      {          System.out.println("can be overrided with any unchecked Exception");      }  }    class SubClassOne extends SuperClass  {      @Override      void methodOfSuperClass() throws NumberFormatException, NullPointerException, RuntimeException      {          System.out.println("Can be overrided with any number of Unchecked Exceptions");      }  }    class SubClassTwo extends SuperClass  {      @Override      void methodOfSuperClass() throws SQLException      {          //Compile time error          //Can not be overrided with checked exception      }  } |

* If a super class method is throwing unchecked exception, then it can be overrided in the sub class with same exception or any other unchecked exceptions but can not be overrided with checked exceptions.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | class SuperClass  {      void methodOfSuperClass() throws ArrayIndexOutOfBoundsException      {          System.out.println("Super class method is throwing Unchecked exception");      }  }    class SubClass extends SuperClass  {      @Override      void methodOfSuperClass() throws ArrayIndexOutOfBoundsException      {          System.out.println("Can be Overrided with same unchecked exception");      }  }    class SubClassOne extends SuperClass  {      @Override      void methodOfSuperClass() throws NumberFormatException, NullPointerException, RuntimeException      {          System.out.println("Can be overrided with any other Unchecked Exceptions");      }  }    class SubClassTwo extends SuperClass  {      @Override      void methodOfSuperClass() throws IOException      {          //Compile time error          //Can not be overrided with checked exception      }  } |

* If super class method is throwing checked type of exception, then it can be overrided with same exception or with it’s sub class exceptions i.e you can decrease the scope of the exception, but can not be overrided with it’s super class exceptions i.e you can not increase the scope of the exception.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | class SuperClass  {      void methodOfSuperClass() throws IOException      {          System.out.println("Super class method is throwing checked exception");      }  }    class SubClass extends SuperClass  {      @Override      void methodOfSuperClass() throws IOException      {          System.out.println("Can be Overrided with same checked exception");      }  }    class SubClassOne extends SuperClass  {      @Override      void methodOfSuperClass() throws FileNotFoundException      {          System.out.println("Can be overrided with checked Exception with lesser scope");      }  }    class SubClassTwo extends SuperClass  {      @Override      void methodOfSuperClass() throws NullPointerException, ArrayIndexOutOfBoundsException, FileNotFoundException      {          System.out.println("Can be overrided with any unchecked exceptions and checked exception with lesser scope");      }  }    class SubClassThree extends SuperClass  {      @Override      void methodOfSuperClass() throws Exception      {          //Compile time error          //Can not be overrided with checked exception with higher scope      }  } |

* One more example with overriding a method throwing both checked and unchecked exceptions.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | class SuperClass  {      void methodOfSuperClass() throws IOException, ClassNotFoundException, NumberFormatException      {          System.out.println("Super class method is throwing both checked and unchecked exceptions");      }  }    class SubClass extends SuperClass  {      @Override      void methodOfSuperClass() throws IOException, ClassNotFoundException      {          System.out.println("Can be Overrided with same checked exceptions");      }  }    class SubClassOne extends SuperClass  {      @Override      void methodOfSuperClass() throws FileNotFoundException      {          System.out.println("Can be overrided with checked Exception with lesser scope");      }  }    class SubClassTwo extends SuperClass  {      @Override      void methodOfSuperClass() throws NullPointerException, ArrayIndexOutOfBoundsException, FileNotFoundException      {          System.out.println("Can be overrided with any other unchecked exceptions and checked exception with lesser scope");      }  }    class SubClassThree extends SuperClass  {      @Override      void methodOfSuperClass() throws Exception      {          //Compile time error          //Can not be overrided with checked exception with higher scope      }  } |

# 10. Checked Vs Unchecked Exceptions In Java:

An exception is an abnormal condition which occurs during the execution of a program and disturbs the normal flow of a program. There are two types of exceptions in Java – one is checked exceptions and another one is unchecked exceptions. Checked exceptions are the exceptions which are checked during compilation itself and those exceptions which are not checked during compilation are called unchecked exceptions. Let’s see checked Vs unchecked exceptions in Java in detail.

## **Checked Exceptions In Java :**

Checked exceptions are the exceptions which are checked during compilation itself. They are also called compile time exceptions. Compiler is aware of these exceptions and immediately throws the error wherever it sees the statements which may throw checked exceptions.

All sub classes of java.lang.Exception (except sub classes of RunTimeException) are checked exceptions. For example, FileNotFoundException, IOException, SQLException, ClassNotFoundException etc…

These exceptions must be handled either using try-catch blocks or using throws clause. If not handled properly, they will give compile time error.

For example,

Below code throws ClassNotFoundException which is a checked exception. But it is not handled, so it gives compile time error.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class CheckedException  {      public static void main(String[] args)      {          Class.forName("AnyClassName");            //Compile time error because          //above statement throws ClassNotFoundException which is a checked exception          //this statement must be enclosed within try-catch block or declare main method with throws clause      }  } |

Below are the correct ways to write the above code.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public class CheckedException  {      public static void main(String[] args)      {          try          {              Class.forName("AnyClassName");          }          catch (ClassNotFoundException ex)          {              System.out.println("ClassNotFoundException will be caught here");          }      }  } |

**OR**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class CheckedException  {      public static void main(String[] args) throws ClassNotFoundException      {          Class.forName("AnyClassName");      }  } |

## **Unchecked Exceptions In Java :**

Unchecked exceptions are the exceptions which are not checked during compilation. Compiler is not aware of these exceptions. These exceptions occur only at run time. That’s why they are also called Run Time Exceptions.

All the sub classes of java.lang.RuntimeException are unchecked exceptions. For example, NullPointerException, ArithmeticException, ClassCastException, ArrayIndexOutOfBoundsException etc…

If any statement in the program throws unchecked exceptions and you are not handling them either using try-catch blocks or throws clause, then it does not give compile time error. Compilation will be successful but program may fail at run time. Therefore, to avoid the premature termination of the program, you have to handle them properly.

For example,

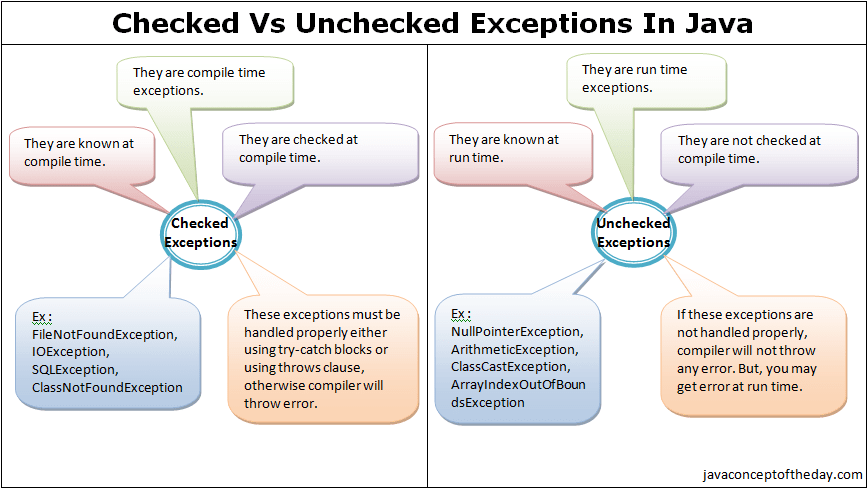
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public class UncheckedException  {      public static void main(String[] args)      {            int i = Integer.parseInt("Unchecked Exception");              //Above statement throws NumberFormatException which is an unchecked exception      }  } |

Above program throws NumberFormatException at run time, but it is not handled properly. So, program will terminate abruptly. To avoid this, you have to handle this exception using try-catch blocks as below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public class UncheckedException  {      public static void main(String[] args)      {          try          {              int i = Integer.parseInt("Unchecked Exception");                //Above statement throws NumberFormatException which is an unchecked exception          }          catch (NumberFormatException e)          {              System.out.println("NumberFormatException will be caught here");          }      }  } |

## **Checked Vs Unchecked Exceptions In Java :**

|  |  |
| --- | --- |
| **Checked Exceptions** | **Unchecked Exceptions** |
| They are known at compile time. | They are known at run time. |
| They are checked at compile time. | They are not checked at compile time. |
| They are compile time exceptions. | They are run time exceptions. |
| These exceptions must be handled properly either using try-catch blocks or using throws clause, otherwise compiler will throw error. | If these exceptions are not handled properly, compiler will not throw any error. But, you may get error at run time. |
| All the sub classes of java.lang.Exception (except sub classes of java.lang.RunTimeException) are checked exceptions. | All the sub classes of java.lang.RunTimeException are unchecked exceptions. |
| Ex : FileNotFoundException, IOException, SQLException, ClassNotFoundException | Ex : NullPointerException, ArithmeticException, ClassCastException, ArrayIndexOutOfBoundsException |

[](https://i0.wp.com/javaconceptoftheday.com/wp-content/uploads/2022/04/CheckedAndUncheckedExceptionsNew.png?ssl=1)

# 11. User Defined Exceptions In Java:

In java, we can define our own exception classes as per our requirements. These exceptions are called **user defined exceptions in java OR Customized exceptions**. User defined exceptions must extend any one of the classes in the hierarchy of exceptions.

Let’s see how to use user defined exceptions in java.

Consider the following example. This example throws an exception when user enters negative age.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | public class ExceptionHandling  {      public static void main(String[] args)      {            Scanner sc = new Scanner(System.in);  //Declaring Scanner variable to take input from user              System.out.println("Enter Your Age");              int age = sc.nextInt();         //Taking input from user              try            {                if(age < 0)                {                    throw new Exception();    //throws an Exception if age is negative                }            }            catch(Exception ex)            {                System.out.println(ex);     //Prints exception description            }      }  } |

When user enters negative value in the above example, it throws an exception and prints exception description which user may not understand. So, Let’s make this program more user friendly. Modify above example so that user can understand why the exception has occurred. To do this, create one sub class to Exception class and override toString() method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | //Defining Our own exception by extending Exception class    class AgeIsNegativeException extends Exception  {      String errorMessage;        public AgeIsNegativeException(String errorMessage)      {          this.errorMessage = errorMessage;      }        //Modifying toString() method to display customized error message        @Override      public String toString()      {          return errorMessage;      }  } |

Above defined exception is called **user defined exception or customized exception.**Now throw this customized exception when user enters negative value.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | public class ExceptionHandling  {      public static void main(String[] args)      {            Scanner sc = new Scanner(System.in);  //Declaring Scanner variable to take input from user              System.out.println("Enter Your Age");              int age = sc.nextInt();         //Taking input from user              try            {                if(age < 0)                {                    throw new AgeIsNegativeException("Age can not be negative");    //throws AgeIsNegativeException if age is negative                }            }            catch(AgeIsNegativeException ex)            {                System.out.println(ex);    //Output : Age can not be negative            }      }  } |

Now, this prints “Age can not be negative” when user enters a negative value. This makes the user understand easily why the error has occurred.

One more example to show user defined exceptions in java.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49 | //Defining Our own exception class by extending ArithmeticException class    class InvalidWithdrawlMoneyException extends ArithmeticException  {      //Overriding toString() method of ArithmeticException as per our needs        @Override      public String toString()      {          return "You don't have that much of money in your account";      }  }    //Using above customized ArithmeticException  public class ExceptionHandling  {      public static void main(String[] args)      {          int balance = 5000;            //Initializing the balance            Scanner sc = new Scanner(System.in);     //Scanner variable to take input from user            System.out.println("Enter Withdrawl Money");            int withdrawlMoney = sc.nextInt();      //taking input from the user            try          {              //checking withdrawl money with the balance              //if withdrawl money is more than the balance,              //then it throws Exception                if(withdrawlMoney > balance)              {                  throw new InvalidWithdrawlMoneyException();              }              else              {                  System.out.println("Transaction Successful");              }          }          catch(InvalidWithdrawlMoneyException ex)          {              //InvalidWithdrawlMoneyException will be caught here                System.out.println(ex);          }      }  } |

We can throw modified exception using anonymous inner class also. Whenever exception occurs, create anonymous inner class, override toString() method and throw the exception. No need to define exception class separately. Above example can be written using anonymous inner classs as,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | public class ExceptionHandling  {      public static void main(String[] args)      {          int balance = 5000;            //Initializing the balance            Scanner sc = new Scanner(System.in);     //Scanner variable to take input from user            System.out.println("Enter Withdrawl Money");            int withdrawlMoney = sc.nextInt();      //taking input from the user            try          {              //checking withdrawl money with the balance              //if withdrawl money is more than the balance,              //then it throws Exception                if(withdrawlMoney > balance)              {                  //throwing exception using anonymous inner class                    throw new ArithmeticException()                  {                      @Override                      public String toString()                      {                          return "You don't have that much of money in your account";                      }                  };              }              else              {                  System.out.println("Transaction Successful");              }          }          catch(ArithmeticException ex)          {              System.out.println(ex);          }      }  } |

# 12. Chained Exceptions In Java:

In an application, one exception throws many exceptions. i.e one exception causes another exception and that exception causes another exception thus forming chain of exceptions. It is better to know where the actual cause of the exception lies. This is possible with chained exceptions feature of the Java.

Chained exceptions are introduced from JDK 1.4. To implement chained exceptions in java, two new constructors and two new methods are added in the Throwable class. They are,

## **Constructors Of Throwable class Which support chained exceptions in java :**

1) Throwable(Throwable cause)    —-> where cause is the exception that causes the current exception.

2) Throwable(String msg, Throwable cause)   —-> where msg is the exception message and cause is the exception that causes the current exception.

## **Methods Of Throwable class Which support chained exceptions in java :**

1) getCause() method : This method returns actual cause of an exception.

2) initCause(Throwable cause) method : This method sets the cause for the calling exception.

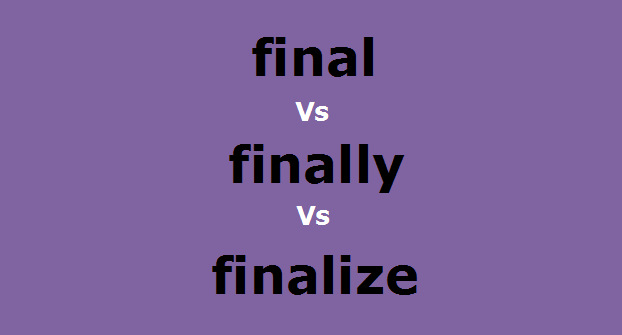
Let’s see one example for how to set and get the actual cause of an exception.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | public class ExceptionHandling  {      public static void main(String[] args)      {          try          {              //creating an exception                NumberFormatException ex = new NumberFormatException("Exception");                //setting a cause of the exception                ex.initCause(new NullPointerException("This is actual cause of the exception"));                throw ex;          }          catch(NumberFormatException ex)          {              System.out.println(ex);     //displaying the exception                System.out.println(ex.getCause());    //getting the actual cause of the exception          }      }  } |

# 13. Difference Between final, finally and finalize In Java:

## **What is the difference between final, finally and finalize in java?**

This is one of the most favorite question of many interviewer for Java freshers as well as for java experienced professionals. They ask this question to confuse the candidate because they spell very similar. That is the only similarity between them. They are totally different things conceptually.



In this post, we will see how final keyword, finally block and finalize() method differ from each other.

## **final keyword :**

final is a keyword which is used to make a variable or a method or a class as “**unchangeable**“. In simple terms,

A variable which is declared as final, it’s value can not be changed once it is initialized.

|  |  |
| --- | --- |
| 1  2  3 | final int i = 10;    //final variable    i = 20;      //Compile time error, Value can not be changed |

A method declared as final can not be overridden or modified in the sub class.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | class SuperClass  {      final void methodOfSuperClass()      {          System.out.println("final Method");      }  }    class SubClass extends SuperClass  {      void methodOfSuperClass()      {          //Compile time error, final method can not be overridden.      }  } |

A class declared as final can not be extended.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | final class SuperClass  {      //final class  }    class SubClass extends SuperClass  {      //Compile time error, can not create a sub class to final class  } |

You can read more about final keyword [here](https://javaconceptoftheday.com/final-keyword-in-java/) and [here](https://javaconceptoftheday.com/final-keyword-in-java-2/).

## **finally Block :**

**finally** is a block which is used for exception handling along with try and catch blocks. finally block is always executed whether exception is raised or not and raised exception is handled or not. Most of time, this block is used to close the resources like database connection, I/O resources etc.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | try  {      //checking the code for exceptions  }  catch(Exception ex)  {      //Catching the exceptions  }  finally  {      //This block is always executed  } |

You can read more about finally block [here](https://javaconceptoftheday.com/try-catch-finally-blocks-java/).

## **finalize() Method :**

**finalize() method** is a protected method of **java.lang.Object** class. It is inherited to every class you create in java. This method is called by garbage collector thread before an object is removed from the memory. finalize() method is used to perform some clean up operations on an object before it is removed from the memory.

|  |  |
| --- | --- |
| 1  2  3  4 | protected void finalize() throws Throwable  {      //Clean up operations  } |

You can read more about finalize() method [here](https://javaconceptoftheday.com/garbage-collection-finalize-method-java/).

## **finally Vs finalize() :**

But, there is one similarity between **finally block** and **finalize() method**. Both are used to close the resources used by the program. finally block is used to close the resources soon after their use. finalize() method is used to close the resources before an object is removed from the memory. That means if you use finalize() method to close the resources, they will remain open until an object,  which is using them, is garbage collected.

But, using finalize() method to close the resources is less recommended as it is not guaranteed that garbage collector will always call finalize() method on an object before it is removed from the memory. If it is not called, the resources will remain open. Therefore, it is always good to close the resources soon after their use using finally block.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | try  {      //Open the resources        //Use the resources  }  catch(Exception ex)  {      //Catching the exceptions raised in the try block  }  finally  {      //Close the resources here only  } |

# 14. Difference Between throw, throws and Throwable In Java:

## **What is the difference between throw, throws and Throwable in java?**

This is another most confusing java interview question asked to java freshers. Interviewer ask this type of questions to confuse the candidate because all three spell similar. But, all three serve different purpose in java. In this article, we will see the differences between [**throw**](https://javaconceptoftheday.com/throwing-rethrowing-exception-java/), [**throws**](https://javaconceptoftheday.com/throws-keyword-java/) and [**Throwable**](https://javaconceptoftheday.com/hierarchy-exceptions-java/) in java.

## **throw In Java :**

**throw** is a keyword in java which is used to throw an exception manually. Using throw keyword, you can throw an exception from any method or block. But, that exception must be of type **java.lang.Throwable** class or it’s sub classes. Below example shows how to throw an exception using throw keyword.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | class ThrowAndThrowsExample  {      void method() throws Exception      {          Exception e = new Exception();            throw e;            //throwing an exception using 'throw'      }  } |

## **throws In Java :**

**throws** is also a keyword in java which is used in the method signature to indicate that this method may throw mentioned exceptions. The caller to such methods must handle the mentioned exceptions either using try-catch blocks or using throws keyword. Below is the syntax for using throws keyword.

|  |  |
| --- | --- |
| 1  2  3  4 | return\_type method\_name(parameter\_list) throws exception\_list  {       //some statements  } |

Below is the example which shows how to use throws keyword.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | class ThrowsExample  {      void methodOne() throws SQLException      {          //This method may throw SQLException      }        void methodTwo() throws IOException      {          //This method may throw IOException      }        void methodThree() throws ClassNotFoundException      {          //This method may throw ClassNotFoundException      }  } |

## **Throwable In Java :**

**Throwable** is a super class for all types of errors and exceptions in java. This class is a member of **java.lang** package. Only instances of this class or it’s sub classes are thrown by the java virtual machine or by the throw statement. The only argument of catch block must be of this type or it’s sub classes. If you want to create your own customized exceptions, then your class must extend this class. Click [here](https://javaconceptoftheday.com/hierarchy-exceptions-java/) to see the hierarchy of exception classes in java.

Below example shows how to create customized exceptions by extending **java.lang.Throwable** class.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class MyException extends Throwable  {             //Customized Exception class  }    class ThrowAndThrowsExample  {      void method() throws MyException      {          MyException e = new MyException();            throw e;      }  } |

# 15. Difference Between Error Vs Exception In Java:

Both **java.lang.Error** and **java.lang.Exception** classes are sub classes of **java.lang.Throwable**class**,** but there exist some significant differences between them. **java.lang.Error** class represents the errors which are mainly caused by the environment in which application is running. For example, **OutOfMemoryError** occurs when JVM runs out of memory or **StackOverflowError** occurs when stack overflows. Where as **java.lang.Exception** class represents the exceptions which are mainly caused by the application itself. For example, **NullPointerException** occurs when an application tries to access null object or **ClassCastException** occurs when an application tries to cast incompatible class types. In this post, we will discuss the differences between Error Vs Exception in Java.

## **Differences Between Error Vs Exception In Java :**

1) Recovering from **Error** is not possible. The only solution to errors is to terminate the execution. Where as you can recover from **Exception** by using either try-catch blocks or throwing exception back to caller.

2) You will not be able to handle the **Errors** using try-catch blocks. Even if you handle them using try-catch blocks, your application will not recover if they happen. On the other hand, **Exceptions** can be handled using try-catch blocks and can make program flow normal if they happen.

3) **Exceptions** in java are divided into two categories – checked and unchecked. Where as all **Errors** belongs to only one category i.e unchecked.

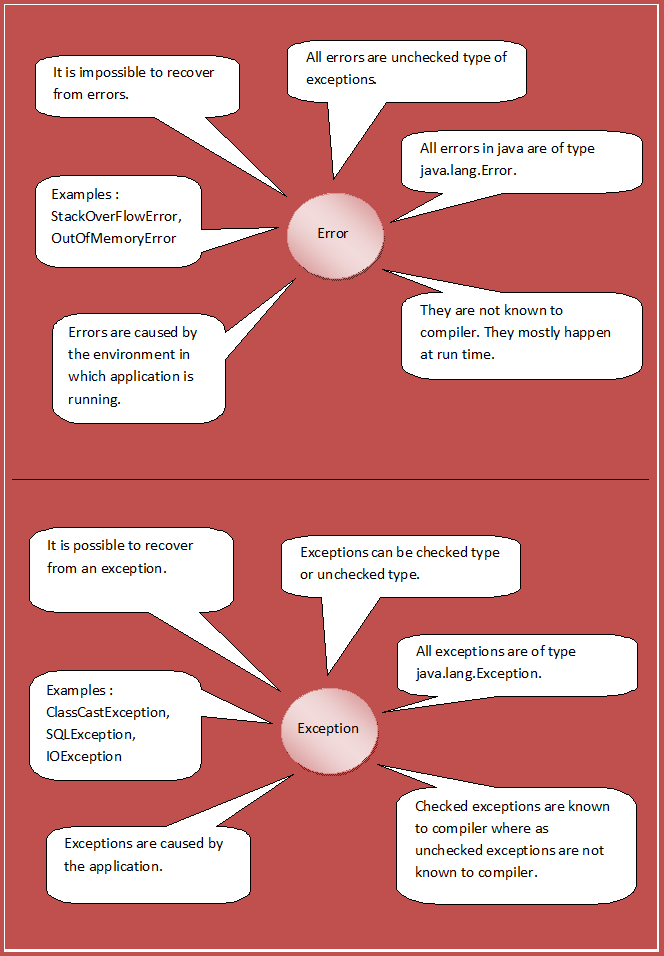
Click [here](https://javaconceptoftheday.com/checked-unchecked-exceptions-java/) for more info on Checked and Unchecked Exceptions.

4) Compiler will not have any knowledge about unchecked exceptions which include **Errors** and sub classes of RunTimeException because they happen at run time. Where as compiler will have knowledge about checked **Exceptions**. Compiler will force you to keep try-catch blocks if it sees any statements which may throw checked exceptions.

5) **Exceptions** are related to application where as **Errors** are related to environment in which application is running.

Below is the quick recap of above points.

|  |  |
| --- | --- |
| Errors | Exceptions |
| Errors in Java are of type java.lang.Error. | Exceptions in Java are of type java.lang.Exception. |
| All errors in java are unchecked type. | Exceptions include both checked as well as unchecked type. |
| Errors happen at run time. They will not be known to compiler. | Checked exceptions are known to compiler where as unchecked exceptions are not known to compiler because they occur at run time. |
| It is impossible to recover from errors. | You can recover from exceptions by handling them through try-catch blocks. |
| Errors are mostly caused by the environment in which application is running. | Exceptions are mainly caused by the application itself. |
| Examples : java.lang.StackOverflowError, java.lang.OutOfMemoryError | Examples : Checked Exceptions : SQLException, IOException Unchecked Exceptions : ArrayIndexOutOfBoundException, ClassCastException, NullPointerException |



# 16. ClassNotFoundException Vs NoClassDefFoundError In Java:

In Java, both **ClassNotFoundException** and **NoClassDefFoundError** occur when a particular class is not found at run time. But, they occur at different scenarios. **ClassNotFoundException** is an exception which occurs when you try to load a class at run time using **Class.forName()** or **loadClass()** methods and mentioned classes are not found in the classpath. On the other hand, **NoClassDefFoundError** is an error which occurs when a particular class is present at compile time but it is missing at run time. In this post, we will see the differences between ClassNotFoundException Vs NoClassDefFoundError in Java and when they occur.

## **ClassNotFoundException In Java :**

ClassNotFoundException is a checked exception which is thrown when an application tries to load a class at run time using **Class.forName()** or **loadClass()** or **findSystemClass()** methods and the class with specified name are not found in the classpath. For example, you may have come across this exception when you try to connect to MySQL or Oracle databases and you have not updated the classpath with required JAR files. In most of time, this exception occurs when you try to run an application without updating the classpath with required JAR files.

For example, below program will throw ClassNotFoundException if the mentioned class **“oracle.jdbc.driver.OracleDriver”** is not found in the classpath.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | public class MainClass  {      public static void main(String[] args)      {          try          {              Class.forName("oracle.jdbc.driver.OracleDriver");          }          catch (ClassNotFoundException e)          {              e.printStackTrace();          }      }  } |

If you run the above program without updating the classpath with required JAR files, you will get the exception like below,

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | java.lang.ClassNotFoundException: oracle.jdbc.driver.OracleDriver      at java.net.URLClassLoader.findClass(Unknown Source)      at java.lang.ClassLoader.loadClass(Unknown Source)      at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)      at java.lang.ClassLoader.loadClass(Unknown Source)      at java.lang.Class.forName0(Native Method)      at java.lang.Class.forName(Unknown Source)      at pack1.MainClass.main(MainClass.java:17) |

## **NoClassDefFoundError In Java :**

NoClassDefFoundError is an error which is thrown when Java Runtime System tries to load the definition of a class and class definition is no longer available. The required class definition was present at compile time but it is missing at run time. For example, compile the below program.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | class A  {    }    public class B  {      public static void main(String[] args)      {          A a = new A();      }  } |

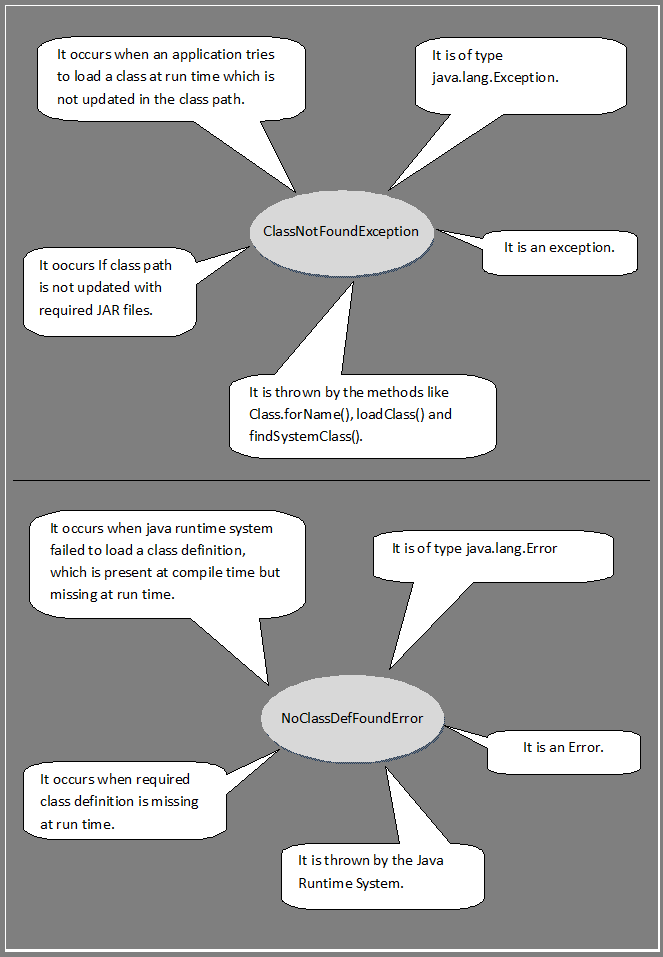
When you compile the above program, two .class files will be generated. One is **A.class** and another one is **B.class**. If you remove the **A.class** file and run the **B.class**file, Java Runtime System will throw NoClassDefFoundError like below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | Exception in thread "main" java.lang.NoClassDefFoundError: A          at MainClass.main(MainClass.java:10)  Caused by: java.lang.ClassNotFoundException: A          at java.net.URLClassLoader.findClass(URLClassLoader.java:381)          at java.lang.ClassLoader.loadClass(ClassLoader.java:424)          at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:331)          at java.lang.ClassLoader.loadClass(ClassLoader.java:357) |

Below is the quick recap of above findings.

## **ClassNotFoundException Vs NoClassDefFoundError In Java :**

|  |  |
| --- | --- |
| **ClassNotFoundException** | **NoClassDefFoundError** |
| It is an exception. It is of type java.lang.Exception. | It is an error. It is of type java.lang.Error. |
| It occurs when an application tries to load a class at run time which is not updated in the classpath. | It occurs when Java runtime system doesn’t find a class definition, which is present at compile time, but missing at run time. |
| It is thrown by the application itself. It is thrown by the methods like Class.forName(), loadClass() and findSystemClass(). | It is thrown by the Java Runtime System. |
| It occurs when classpath is not updated with required JAR files. | It occurs when required class definition is missing at run time. |



# 17. ClassCastException In Java:

ClassCastException in java is a run time error it occurs when an object can not be casted to another type.

An object is automatically upcasted to its super class type. You need not to mention class type explicitly. But, when an object is supposed to be downcasted to its sub class type, then you have to mention class type explicitly. In such case, there is a possibility of occurring class cast exception. In most of time, it occurs when you are trying to downcast an object explicitly to its sub class type.

Try to run below program.

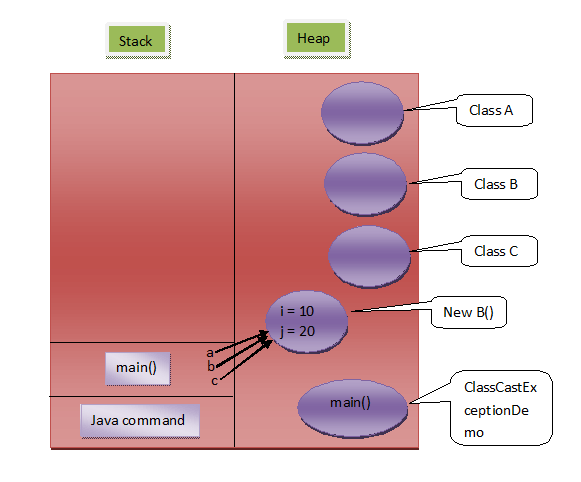
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | package com;  class A  {      int i = 10;  }    class B extends A  {      int j = 20;  }    class C extends B  {      int k = 30;  }    public class ClassCastExceptionDemo  {      public static void main(String[] args)      {          A a = new B();   //B type is auto up casted to A type          B b = (B) a;     //A type is explicitly down casted to B type.          C c = (C) b;    //Here, you will get class cast exception          System.out.println(c.k);      }  } |

You will get ClassCastException. Below is the sample of the error.

**Exception in thread “main” java.lang.ClassCastException: com.B cannot be cast to com.C**  
**at com.ClassCastExceptionDemo.main(ClassCastExceptionDemo.java:23)**

In the above example, Class B extends Class A and Class C extends Class B. In the main method, Class B-type object is created (Line 21). It will be having two non-static fields. one field (int i) is inherited from class A and another one is its own field (int j). ‘a’ is Class A-type reference variable which will be pointing to this newly created object. In the next statement (Line 22), reference variable ‘a’ is assigned to ‘b’ which is Class B-type reference variable. After execution of this statement, ‘b’ will also be pointing to the same object to which ‘a’ is pointing. In the third statement, ‘b’ is assigned to ‘c’ which is Class C-type reference variable. So, ‘c’ will also be pointing to same object to which ‘a’ and ‘b’ are pointing. While executing this statement, you will get run time exception called Class Cast Exception.

The memory allocation of above program can be diagrammatically represented as,



**Why you got this exception?**

Every sub class extends its super class. i.e every child class will have some additional properties along with some inherited properties from its parent class. In the above example, Class A has one property (int i). Class B has two properties, one is it’s own and another one is inherited. Class C has three properties. one is it’s own and two are inherited. In this example, Class C-type reference variable is referring to Class B-type object. Class B-type object will be having only two properties. But, through Class C-type reference variable, you can access Class C’s own property (int k) like in the line 24. But, actually this property does not exist in Class B-type object. This creates the confusion. Class B-type can not be casted to Class C-type. That’s why, you will get class cast exception.

Put ClassCastException in simple terms. ClassCastException occurs when code has attempted to cast an object to a type of which it is not an object. In the above example, Class B is a Class A type but Class B is not a Class C type. Therefore, you are getting ClassCastException.

Consider one more case of ClassCastException.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public class ClassCastExceptionDemo  {      public static void main(String[] args)      {          Object o = new String();          Integer i = (Integer) o;      }  } |

We all know that every class in java is a sub class of java.lang.Object class. String is also a subclass of Obeject class and Integer is also a subclass of Object class. In the above example, String object is created and it is automatically up casted to Object type. Further, this object is explicitly downcasted to Integer type. This causes ClassCastException, because, String object is not an Integer type.

# 18. 12 Frequently Occurring Exceptions In Java With Examples:

### **12 Most Frequently Occurring Exceptions In Java With Examples :**

**1) java.lang.NullPointerException**

NullPointerException is a RunTimeException which occurs when your application tries to access null object. It happens if you don’t initialize the reference variable and it is pointing to null instead of actual object. Using such reference variable will cause NullpointerException.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | package pack1;    class A  {      static String s;  }    public class MainClass  {      public static void main(String[] args)      {         System.out.println(A.s.length());      }  } |

In the above code, we are trying to find the length of string filed **‘s’** of class **A**. But, it is not initialized. It is pointing to null. This will throw java.lang.NullPointerException like below.

|  |  |
| --- | --- |
| 1  2 | Exception in thread "main" java.lang.NullPointerException      at pack1.MainClass.main(MainClass.java:12) |

**2) java.lang.ArrayIndexOutOfBoundsException**

ArrayIndexOutOfBoundsException is also one of the frequently occurring exception in java. It occurs when you try to access an array element with an invalid index i.e index greater than the array length or with a negative index.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | package pack1;    public class MainClass  {      public static void main(String[] args)      {         String s[] = new String[] {"ZERO", "ONE", "TWO", "THREE", "FOUR"};           System.out.println(s[5]);    //This will throw ArrayIndexOutOfBoundsException           System.out.println(s[-1]);   //This will also throw ArrayIndexOutOfBoundsException      }  } |

In the above example, string array **‘s’** contains only 5 elements. But, we are trying to access sixth element. This will result in java.lang.ArrayIndexOutOfBoundsException.

|  |  |
| --- | --- |
| 1  2 | Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5      at pack1.MainClass.main(MainClass.java:9) |

**3) java.lang.NumberFormatException**

NumberFormatException is thrown when you are trying to convert a string to numeric value like integer, float, double etc…, but input string is not a valid number. NumberFormatException is also one of IllegalArgumentException.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | package pack1;    public class MainClass  {      public static void main(String[] args)      {         int i = Integer.parseInt("abc");      }  } |

In the above code, we are trying to convert a string **“abc”** to number, which is not possible. This will throw java.lang.NumberFormatException.

|  |  |
| --- | --- |
| 1  2  3  4  5 | Exception in thread "main" java.lang.NumberFormatException: For input string: "abc"      at java.lang.NumberFormatException.forInputString(Unknown Source)      at java.lang.Integer.parseInt(Unknown Source)      at java.lang.Integer.parseInt(Unknown Source)      at pack1.MainClass.main(MainClass.java:7) |

**4) java.lang.ClassNotFoundException**

ClassNotFoundException is a checked type of exception. It is thrown when an application tries to load a class at run time using Class.forName() or loadClass() or findSystemClass() methods, but the class with specified name is not found in the classpath. This frequently occurs when you try to run your application without updating the class path with required *JAR* files.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | package pack1;    public class MainClass  {      public static void main(String[] args)      {         try         {             Class.forName("oracle.jdbc.driver.OracleDriver");         }         catch (ClassNotFoundException e)         {             e.printStackTrace();         }      }  } |

The above code will throw java.lang.ClassNotFoundException if you don’t update the classpath with Oracle JDBC driver class.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | java.lang.ClassNotFoundException: oracle.jdbc.driver.OracleDriver      at java.net.URLClassLoader.findClass(Unknown Source)      at java.lang.ClassLoader.loadClass(Unknown Source)      at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)      at java.lang.ClassLoader.loadClass(Unknown Source)      at java.lang.Class.forName0(Native Method)      at java.lang.Class.forName(Unknown Source)      at pack1.MainClass.main(MainClass.java:9) |

**5) java.lang.ArithmeticException**

ArithmeticException is also a RunTimeException which is thrown when an abnormal arithmetic condition arises in an application. For example, divide by zero exception.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | package pack1;    public class MainClass  {      public static void main(String[] args)      {         int i = 10/0;           System.out.println(i);      }  } |

The above code will throw java.lang.ArithmeticException like below.

|  |  |
| --- | --- |
| 1  2 | Exception in thread "main" java.lang.ArithmeticException: / by zero      at pack1.MainClass.main(MainClass.java:7) |

**6) java.sql.SQLException**

SQLException is thrown when an application encounters with an error while interacting with the database. For example, passing the wrong username or password, passing the wrong URL of the database, passing invalid column name or column index etc. SQLException is also a checked exception.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | import java.sql.\*;    public class MainClass  {      static      {          try          {              Class.forName("oracle.jdbc.driver.OracleDriver");          }          catch (ClassNotFoundException e)          {              System.out.println("Unable To Load The Driver class");          }      }        public static void main(String[] args)      {          Connection con = null;            String URL = "jdbc:oracle:thin:@localhost:1521:XE";            String username = "username";            String password = "password";            try          {              con = DriverManager.getConnection(URL, username, password);          }          catch (SQLException e)          {              e.printStackTrace();          }      }  } |

In the above program, we are trying to connect to the database by passing wrong username and password. This will throw java.sql.SQLException.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | java.sql.SQLException: ORA-01017: invalid username/password; logon denied        at oracle.jdbc.driver.DatabaseError.throwSqlException(DatabaseError.java:112)      at oracle.jdbc.driver.T4CTTIoer.processError(T4CTTIoer.java:331)      at oracle.jdbc.driver.T4CTTIoer.processError(T4CTTIoer.java:283)      at oracle.jdbc.driver.T4CTTIoer.processError(T4CTTIoer.java:278)      at oracle.jdbc.driver.T4CTTIoauthenticate.receiveOauth(T4CTTIoauthenticate.java:785)      at oracle.jdbc.driver.T4CConnection.logon(T4CConnection.java:362)      at oracle.jdbc.driver.PhysicalConnection.<init>(PhysicalConnection.java:414)      at oracle.jdbc.driver.T4CConnection.<init>(T4CConnection.java:165)      at oracle.jdbc.driver.T4CDriverExtension.getConnection(T4CDriverExtension.java:35)      at oracle.jdbc.driver.OracleDriver.connect(OracleDriver.java:801)      at java.sql.DriverManager.getConnection(Unknown Source)      at java.sql.DriverManager.getConnection(Unknown Source)      at MainClass.main(MainClass.java:31) |

**7) java.lang.ClassCastException**

ClassCastException occurs when an object of one type can not be casted to another type. While casting, an object must satisfy “IS-A” relationship. If it doesn’t satisfy then JVM will throw java.lang.ClassCastException.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | package pack1;    class A  {    }    class B extends A  {    }    public class MainClass  {      public static void main(String[] args)      {         A a = new A();           B b = (B)a;      }  } |

In the above example, we are trying to cast an object of type A to type B. But, it doesn’t satisfy “IS-A” relationship. i.e A is not of type B. This will throw java.lang.ClassCastException.

|  |  |
| --- | --- |
| 1  2 | Exception in thread "main" java.lang.ClassCastException: pack1.A cannot be cast to pack1.B      at pack1.MainClass.main(MainClass.java:19) |

**8) java.io.IOException**

IOException occurs when an IO operation fails in your application. IOException is a checked type of exception. This exception is the super class for all type of IO exceptions. Some of the popular IO exceptions are FileNotFoundException, SocketException, SSLException etc.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | package pack1;    import java.io.BufferedReader;  import java.io.FileNotFoundException;  import java.io.FileReader;  import java.io.IOException;    public class MainClass  {      public static void main(String[] args)      {         String filePath = "C:\\Users\\Bablad\\Desktop\\Open.txt";           BufferedReader reader = null;           try         {             reader = new BufferedReader(new FileReader(filePath));         }         catch (FileNotFoundException e)         {             e.printStackTrace();         }         finally         {             try             {                 if(reader != null)                 {                     reader.close();                 }             }             catch (IOException e)             {                 e.printStackTrace();             }         }      }  } |

In the above example, we are trying to read a file which doesn’t exist in the path specified. It causes java.io.FileNotFoundException.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | java.io.FileNotFoundException: C:\Users\Bablad\Desktop\Open.txt (The system cannot find the file specified)      at java.io.FileInputStream.open0(Native Method)      at java.io.FileInputStream.open(Unknown Source)      at java.io.FileInputStream.<init>(Unknown Source)      at java.io.FileInputStream.<init>(Unknown Source)      at java.io.FileReader.<init>(Unknown Source)      at pack1.MainClass.main(MainClass.java:18) |

**9) java.lang.InterruptedException**

You may have come across this exception if you have worked on multithreaded programming. InterruptedException is thrown when a sleeping thread or waiting thread is interrupted. The methods like sleep(), wait() and join() methods throw InterruptedException. InterruptedException is also a checked exception.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | public class MainClass  {      public static void main(String[] args)      {          Thread t = new Thread()          {              public void run()              {                  try                  {                      Thread.sleep(10000);   //Thread sleeps for 10s                  }                  catch (InterruptedException e)                  {                      e.printStackTrace();                  }              }          };            t.start();            try          {              Thread.sleep(3000);   //main thread sleeping for 3s          }          catch (InterruptedException e)          {              e.printStackTrace();          }            t.interrupt();    //interrupting thread t      }  } |

In the above example, main thread interrupts thread ***‘t’*** while it is sleeping. This causes java.lang.InterruptedException to arise.

|  |  |
| --- | --- |
| 1  2  3 | java.lang.InterruptedException: sleep interrupted      at java.lang.Thread.sleep(Native Method)      at MainClass$1.run(MainClass.java:11) |

**10) java.lang.SecurityException**

SecurityException is thrown by the security manager if an application violates the security rules. For example, changing a thread name or thread priority to which you don’t have access or using a package name which is already used. SecurityException indicates that application has violated the security rules and it can not continue the execution.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | package java.lang;    public class MainClass  {      public static void main(String[] args)      {          System.out.println("Java Concept Of The Day");      }  } |

The above code will throw java.lang.SecurityException. Because, we are using package name “java.lang” which is already used in JDK.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | Exception in thread "main" java.lang.SecurityException: Prohibited package name: java.lang      at java.lang.ClassLoader.preDefineClass(Unknown Source)      at java.lang.ClassLoader.defineClass(Unknown Source)      at java.security.SecureClassLoader.defineClass(Unknown Source)      at java.net.URLClassLoader.defineClass(Unknown Source)      at java.net.URLClassLoader.access$100(Unknown Source)      at java.net.URLClassLoader$1.run(Unknown Source)      at java.net.URLClassLoader$1.run(Unknown Source)      at java.security.AccessController.doPrivileged(Native Method)      at java.net.URLClassLoader.findClass(Unknown Source)      at java.lang.ClassLoader.loadClass(Unknown Source)      at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)      at java.lang.ClassLoader.loadClass(Unknown Source)      at sun.launcher.LauncherHelper.checkAndLoadMain(Unknown Source) |

**11) java.lang.StackOverflowError**

StackOverflowError is a run time error which occurs when stack overflows. This happens when you keep calling the methods recursively.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public class MainClass  {      static void methodOne()      {          methodTwo();      }        static void methodTwo()      {          methodOne();      }        public static void main(String[] args)      {          methodOne();      }  } |

The above code will throw java.lang.StackOverflowError, because methodOne() and methodTwo() are calling each other recursively.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | Exception in thread "main" java.lang.StackOverflowError      at MainClass.methodTwo(MainClass.java:10)      at MainClass.methodOne(MainClass.java:5)      at MainClass.methodTwo(MainClass.java:10)      at MainClass.methodOne(MainClass.java:5)      at MainClass.methodTwo(MainClass.java:10)      at MainClass.methodOne(MainClass.java:5)      at MainClass.methodTwo(MainClass.java:10)      at MainClass.methodOne(MainClass.java:5)  ......  ......  ...... |

**12) java.lang.NoClassDefFoundError**

NoClassDefFoundError is thrown when Java Runtime System tries to load the definition of a class which is no longer available. The required class definition was present at compile time but it was missing at run time.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | class A  {    }    public class B  {      public static void main(String[] args)      {          A a = new A();      }  } |

When you compile the above program, two .class files will be generated. One is A.class and another one is B.class. If you run the B.class file after deleting the A.class file, java.lang.NoClassDefFoundError will be thrown.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | Exception in thread "main" java.lang.NoClassDefFoundError: A          at MainClass.main(MainClass.java:10)  Caused by: java.lang.ClassNotFoundException: A          at java.net.URLClassLoader.findClass(URLClassLoader.java:381)          at java.lang.ClassLoader.loadClass(ClassLoader.java:424)          at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:331)          at java.lang.ClassLoader.loadClass(ClassLoader.java:357) |

# clone() Method Of java.lang.Object Class:

clone() method is a non-static **protected** method of java.lang.Object class. This method is used to create a clone or copy of the given object. It throws **CloneNotSupportedException** if an object is not clone-able. Here is the method signature of clone() method.

**protected Object clone() throws CloneNotSupportedException**

Not all the objects in java are clone-able. In order to make an object clone-able, the class of that object must implement **Cloneable** interface. Cloneable interface is a marker interface. It does not have any methods or fields in it. It is just used to provide a marker for cloning operation.

CloneNotSupportedException is a checked type of exception. Therefore, you have to keep calling statement to clone() method in try-catch blocks or specify it using throws clause.

clone() method is a protected method. So, you can’t use it outside the class without overriding it.

Below example shows how to create a clone of an object using clone() method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56 | //Class A implementing Cloneable interface    class A implements Cloneable  {      int i;        int j;        public A(int i, int j)      {          this.i = i;            this.j = j;      }        //Overriding clone() method        @Override      protected Object clone() throws CloneNotSupportedException      {          return super.clone();      }  }    public class CloneMethodDemo  {     public static void main(String[] args)     {         //Creating an instance of Class A           A a1 = new A(10, 20);           //Declaring reference variable of Class A and assigning null to it           A a2 = null;           //enclosing a1.clone() in try-catch blocks           //as clone() throws CloneNotSupportedException           try         {             //Creating a clone of a1 and assigning it to a2               a2 = (A) a1.clone();         }         catch (CloneNotSupportedException e)         {             System.out.println("Object is not clone-able");         }           System.out.println(a2.i);   //Output : 10           System.out.println(a2.j);   //Output : 20     }  } |

## **Shallow Copy And Deep Copy :**

If a cloned object and original objects are not 100% disjoint, then it is called **shallow copy**. In shallow copy operation, any changes made to clone will be reflected in the original or vice-versa. This happens when an object has reference variables as fields.

For example, in the below program Class B has reference variable ‘a’ of type class A. This will be pointing to an object of type class A. When you create a clone ‘b2’ of object ‘b1’ of type Class B, that clone will also have this reference variable pointing to same object. Any changes you make to this object through clone will be reflected in the original object.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64 | class A  {      int i;        public A(int i)      {          this.i = i;      }  }    class B implements Cloneable  {      int j;        A a;        public B(int j, A a)      {          this.j = j;            this.a = a;      }        @Override      protected Object clone() throws CloneNotSupportedException      {          return super.clone();      }  }    public class CloneMethodDemo  {     public static void main(String[] args)     {         A a = new A(10);           B b1 = new B(20, a);           B b2 = null;           try         {             //Creating clone of b1 and assigning it to b2                b2 = (B) b1.clone();         }         catch (CloneNotSupportedException e)         {             System.out.println("Onject is not clone-able");         }           //Printing value of b1.a.i           System.out.println(b1.a.i);        //Output : 10           //Changing the value of b2.a.i to 100           b2.a.i = 100;           //This change will be reflected in original object 'b1'           System.out.println(b1.a.i);       //Output : 100     }  } |

The default version of clone() method implements the shallow copy. Therefore, you need to override clone() method so that any changes made to clone should not be reflected in original or vice-versa. This type of cloning is called deep copy.

If a cloned object and original objects are 100% disjoint, then it is called **deep copy**. In deep copy operation, any changes made to cloned object will not be reflected in original object or vice-versa.

In the below example, clone() method is overrided to implement the deep copy operation.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76 | class A implements Cloneable  {      int i;        public A(int i)      {          this.i = i;      }        @Override      protected Object clone() throws CloneNotSupportedException      {          return super.clone();      }  }    class B implements Cloneable  {      int j;        A a;        public B(int j, A a)      {          this.j = j;            this.a = a;      }            //Overriding clone method to implement deep copy        @Override      protected Object clone() throws CloneNotSupportedException      {          B b = (B) super.clone();            b.a = (A) a.clone();            return b;      }  }    public class CloneMethodDemo  {     public static void main(String[] args)     {         A a = new A(10);           B b1 = new B(20, a);           B b2 = null;           try         {             //Creating clone of b1 and assigning it to b2                b2 = (B) b1.clone();         }         catch (CloneNotSupportedException e)         {             System.out.println("Onject is not clone-able");         }           //Printing value of b1.a.i           System.out.println(b1.a.i);        //Output : 10           //Changing the value of b2.a.i to 100           b2.a.i = 100;           //Now, this change will not effect the original object           System.out.println(b1.a.i);       //Output : 10     }  } |

## **Some Extra Points About clone() Method :**

* If you try to clone an object of the class that does not implement Cloneable interface, you will get **CloneNotSupportedException** at run time.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | class A  {      int i = 10;        @Override      protected Object clone() throws CloneNotSupportedException      {          return super.clone();      }  }    public class CloneMethodDemo  {     public static void main(String[] args)     {         A a1 = new A();           try         {             A a2 = (A) a1.clone();   //This statement will throw CloneNotSupportedException               //Because, Class A does not implement Cloneable interface         }         catch (CloneNotSupportedException e)         {             e.printStackTrace();         }     }  } |

* As clone method is **protected**, you must override it to use outside the class or else use it inside a class like below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | class A  {      int i = 10;        public static void main(String[] args)      {          A a1 = new A();            A a2 = null;            try          {              //Using clone() method directly as we are using it inside the class                a2 = (A) a1.clone();            }          catch (CloneNotSupportedException e)          {              e.printStackTrace();          }      }  } |

* The default implementation of clone() method does shallow copy of an object. This type of copying an object is dangerous and unsecured when an object contains the references to other objects. So. try to avoid the shallow copy by overriding the clone() method.
* Cloning is an unsecured operation. That why whenever Java run time sees cloning, it expects one marker from the developer. We are providing that marker in the form of Cloneable interface.
* While cloning, copy of the object is created by **field-by-field** assignment. No constructor is called while cloning.

# How To Launch External Applications Through Java Program:

For every java application, there is a one and only one **java.lang.Runtime** object associated with it. This Runtime objcet is used to interact with the environment in which application is running. Application itself can’t create an instance of Runtime. But, You can retrieve Runtime object associated with the appliaction using **getRuntime()** method of java.lang.Runtime class. Like below,

**Runtime runtime = Runtime.getRuntime();**

You can use thus obtained Runtime object to interact with runtime environment of that application.

There is one method in java.lang.Runtime class called **exec()** method. This method executes the specified system command in a separate process. You can use this method to launch external applications like notepad, browser or any media player through your java program. For example if you pass **“notepad.exe”** to this method, it opens new instance of notepad application.

There are total 6 versions of exec() method available in Runtime class. You can follow all those methods [here](http://docs.oracle.com/javase/7/docs/api/java/lang/Runtime.html). We will discuss two widely used exec() methods in this post. They are,

**1) public Process exec(String command) throws IOException**

—>This method takes system command in the form of string.

**2) public Process exec(String[] command) throws IOException**

—> This method takes system command in the form of string array.

Here is the java program which opens a new instance of notepad using exec() method which takes system command as a string.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | public class LaunchingExternalApps  {      public static void main(String[] args)      {          Runtime runtime = Runtime.getRuntime();     //getting Runtime object            try          {              runtime.exec("notepad.exe");        //opens new notepad instance                //OR runtime.exec("notepad");          }          catch (IOException e)          {              e.printStackTrace();          }      }  } |

You can also open a particular file in notepad using the same exec() method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public class LaunchingExternalApps  {      public static void main(String[] args)      {          Runtime runtime = Runtime.getRuntime();     //getting Runtime object            try          {              runtime.exec("notepad I:\\sample.txt");        //opens "sample.txt" in notepad          }          catch (IOException e)          {              e.printStackTrace();          }      }  } |

Please notice that file name is specified along with it’s path (**I:\sample.txt**).

You can also open an URL in any browser using exec() method which takes command as string array. First element of string array must be the path of .exe file of installed browser and second element must be URL to open.

Here is a java program which opens “**https://javaconceptoftheday.com/**” in chrome browser.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | public class LaunchingExternalApps  {      public static void main(String[] args)      {          Runtime runtime = Runtime.getRuntime();     //getting Runtime object            String[] s = new String[] {"C:\\Program Files (x86)\\Google\\Chrome\\Application\\chrome.exe", "<https://javaconceptoftheday.com/>"};            try          {              runtime.exec(s);        //opens "<https://javaconceptoftheday.com/>" in chrome browser          }          catch (IOException e)          {              e.printStackTrace();          }      }  } |

Here is an example which opens “**sample.mp3**” file in VLC Media Player.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class LaunchingExternalApps  {      public static void main(String[] args) throws Exception      {          Runtime runtime = Runtime.getRuntime();     //getting Runtime object            String[] s = new String[] {"C:\\Program Files\\VideoLAN\\VLC\\vlc.exe", "F:\\sample.mp3"};            Process process = runtime.exec(s);        //opens "sample.mp3" in VLC Media Player      }  } |

You can also close the launched applications using **destroy()** method of Process class. Just call the destroy() method on the process object returned by the exec() method.

Here is the java program which opens the notepad instance and closes it after 5 seconds.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class LaunchingExternalApps  {      public static void main(String[] args) throws Exception      {          Runtime runtime = Runtime.getRuntime();     //getting Runtime object            Process process = runtime.exec("notepad I:\\sample.txt");        //opens "sample.txt" in notepad            Thread.sleep(5000);            process.destroy();      }  } |

**­**