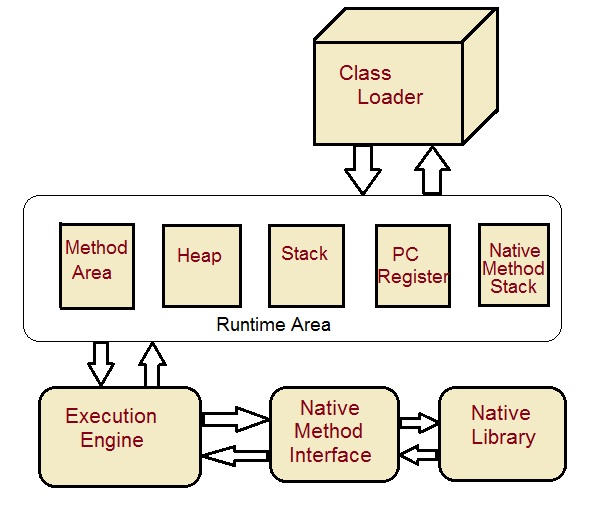
**What is JVM?**

Java virtual Machine(JVM) is a virtual Machine that provides runtime environment to execute java byte code. The JVM doesn't understand Java typo, that's why you compile your \*.java files to obtain \*.class files that contain the bytecodes understandable by the JVM.

JVM control execution of every Java program. It enables features such as automated exception handling, Garbage-collected heap.

**JVM Architecture**



**Class Loader :** Class loader loads the Class for execution.

**Method area :** Stores pre-class structure as constant pool.

**Heap :** Heap is in which objects are allocated.

**Stack :** Local variables and partial results are store here. Each thread has a private JVM stack created when the thread is created.

**Program register :** Program register holds the address of JVM instruction currently being executed.

**Native method stack :** It contains all native used in application.

**Executive Engine :** Execution engine controls the execute of instructions contained in the methods of the classes.

**Native Method Interface :** Native method interface gives an interface between java code and native code during execution.

**Native Method Libraries :** Native Libraries consist of files required for the execution of native code.

**Difference between JDK and JRE**

**JRE** : The Java Runtime Environment (JRE) provides the libraries, the Java Virtual Machine, and other components to run applets and applications written in the Java programming language. JRE does not contain tools and utilities such as compilers or debuggers for developing applets and applications.



**JDK** : The JDK also called Java Development Kit is a superset of the JRE, and contains everything that is in the JRE, plus tools such as the compilers and debuggers necessary for developing applets and applications.



**Type Casting**

Assigning a value of one type to a variable of another type is known as **Type Casting**.

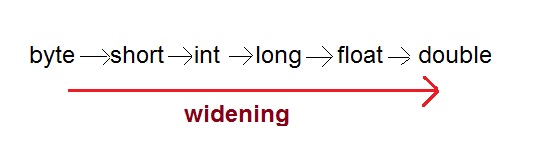
**Example :**

int x = 10;

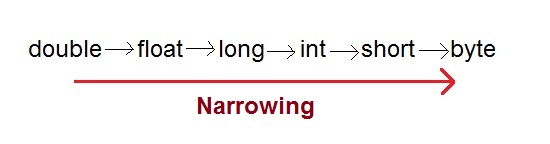
byte y = (byte)x;

In Java, type casting is classified into two types,

* Widening Casting(Implicit)



* Narrowing Casting(Explicitly done)



**Widening or Automatic type converion**

Automatic Type casting take place when,

* the two types are compatible
* the target type is larger than the source type

**Example :**

public class Test

{

public static void main(String[] args)

{

int i = 100;

long l = i; **//no explicit type casting required**

float f = l; **//no explicit type casting required**

System.out.println("Int value "+i);

System.out.println("Long value "+l);

System.out.println("Float value "+f);

}

}

Output :

Int value 100

Long value 100

Float value 100.0

**Narrowing or Explicit type conversion**

When you are assigning a larger type value to a variable of smaller type, then you need to perform explicit type casting.

**Example :**

public class Test

{

public static void main(String[] args)

{

double d = 100.04;

long l = (long)d; **//explicit type casting required**

int i = (int)l; **//explicit type casting required**

System.out.println("Double value "+d);

System.out.println("Long value "+l);

System.out.println("Int value "+i);

}

}

Output :

Double value 100.04

Long value 100

Int value 100

**this keyword**

* **this** keyword is used to refer to current object.
* **this** is always a reference to the object on which method was invoked.
* **this** can be used to invoke current class constructor.
* **this** can be passed as an argument to another method.

*Example* :

class Box

{

Double width, weight, dept;

Box (double w, double h, double d)

{

this.width = w;

this.height = h;

this.depth = d;

}

}

Here the **this** is used to initialize member of current object.

**The this is used to call overloaded constructor in java**

class Car

{

private String name;

public Car()

{

this("BMW"); //oveloaded constructor is called.

}

public Car(Stting n)

{

this.name=n; //member is initialized using this.

}

}

**The this is also used to call Method of that class.**

public void getName()

{

System.out.println("Studytonight");

}

public void display()

{

this.getName();

System.out.println();

}

**this is used to return current Object**

public Car getCar()

{

return this;

}

**Aggregation (HAS-A)**

HAS-A relationship is based on usage, rather than inheritance. In other words, class A *has-a* relationship with class B, if code in class A has a reference to an instance of class B.

**Example**

class Student

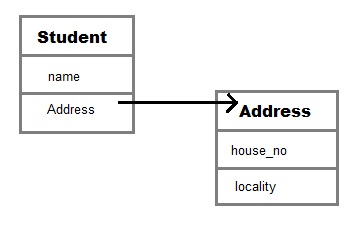
{

String name;

Address ad;

}

Here you can say that **Student** has-a **Address**.



**Student** class has an instance variable of type Address. Student code can use Address reference to invoke methods on the **Address**, and get **Address** behavior.

Aggregation allow you to design classes that follow good Object Oriented practices. It also provide code reusability.

**Example of Aggregation**

class Author

{

String authorName;

int age;

String place;

Author(String name,int age,String place)

{

this.authorName=name;

this.age=age;

this.place=place;

}

public String getAuthorName()

{

return authorName;

}

public int getAge()

{

return age;

}

public String getPlace()

{

return place;

}

}

class Book

{

String name;

int price;

Author auth;

Book(String n,int p,Author at)

{

this.name=n;

this.price=p;

this.auth=at;

}

public void showDetail()

{

System.out.println("Book is"+name);

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

System.out.println("Author is "+auth.getAuthorName());

}

}

class Test

{

public static void main(String args[])

{

Author ath=new Author("Me",22,"India");

Book b=new Book("Java",550,ath);

b.showDetail();

}

}

**Output:**

Book is Java.

price is 550.

Author is me.

**Q. What is Composition in java?**

Composition is restricted form of Aggregation. For example a class **Car** cannot exist without **Engine**.

class Car

{

private Engine engine;

Car(Engine en)

{

engine = en;

}

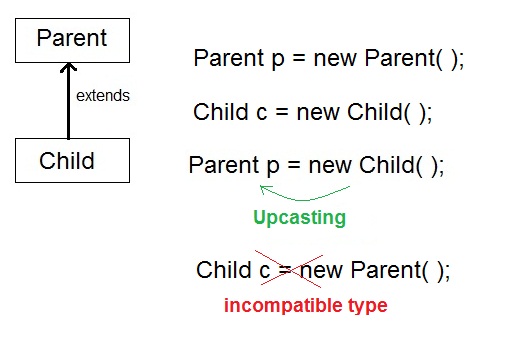
}

**Q. When to use Inheritance and Aggregation?**

When you need to use property and behaviour of a class without modifying it inside your class. In such case**Aggregation** is a better option. Whereas when you need to use and modify property and behaviour of a class inside your class, its best to use **Inheritance**.

**Runtime Polymorphism or Dynamic method dispatch**

Dynamic method dispatch is a mechanism by which a call to an overridden method is resolved at runtime. This is how java implements runtime polymorphism. When an overridden method is called by a reference, java determines which version of that method to execute based on the type of object it refer to. In simple words the type of object which it referred determines which version of overridden method will be called.



**Upcasting**

When **Parent** class reference variable refers to **Child** class object, it is known as **Upcasting**

**Example**

class **Game**

{

public void type()

{ System.out.println("Indoor & outdoor"); }

}

Class **Cricket** extends **Game**

{

public void type()

{ System.out.println("outdoor game"); }

public static void main(String[] args)

{

Game gm = new Game();

Cricket ck = new Cricket();

gm.type();

ck.type();

gm=ck; **//gm refers to Cricket object**

gm.type(); **//calls Cricket's version of type**

}

}

**Output:**

Indoor & outdoor

Outdoor game

Outdoor game

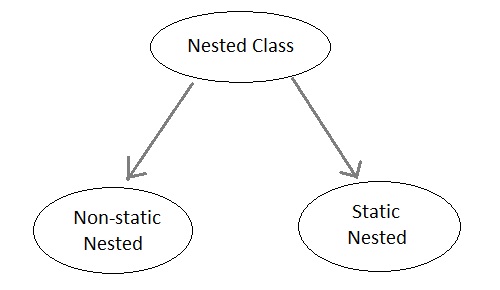
Notice the last output. This is because of **gm = ck**; Now gm.type() will call Cricket version of type method. Because here gm refers to cricket object.

**Q. Difference between Static binding and Dynamic binding in java ?**

Static binding in Java occurs during compile time while dynamic binding occurs during runtime. Static binding uses type(Class) information for binding while dynamic binding uses instance of class(Object) to resolve calling of method at run-time. Overloaded methods are bonded using static binding while overridden methods are bonded using dynamic binding at runtime.

**Nested Class**

A class within another class is known as Nested class. The scope of the nested is bounded by the scope of its enclosing class.



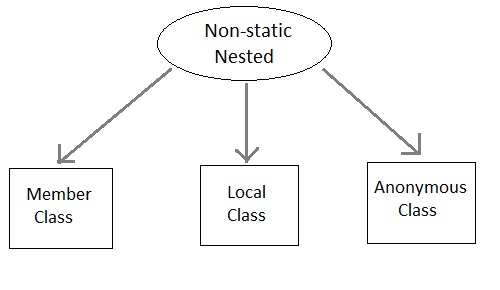
**Static Nested Class**

A satic nested class is the one that has **static** modifier applied. Because it is static it cannot refer to non-static members of its enclosing class directly. Because of this restriction static nested class is seldom used.

**Non-static Nested class**

Non-static Nested class is most important type of nested class. It is also known as **Inner** class. It has access to all variables and methods of **Outer** class and may refer to them directly. But the reverse is not true, that is,**Outer** class cannot directly access members of **Inner** class.

One more important thing to notice about an **Inner** class is that it can be created only within the scope of**Outer** class. Java compiler generates an error if any code outside **Outer** class attempts to instantiate **Inner**class.



**Example of Inner class**

class Outer

{

public void display()

{

Inner in=new Inner();

in.show();

}

class Inner

{

public void show()

{

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

}

}

}

class Test

{

public static void main(String[] args)

{

Outer ot=new Outer();

ot.display();

}

}

**Output:**

Inside inner

**Example of Inner class inside a method**

class Outer

{

int count;

public void display()

{

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

{

class Inner //Inner class defined inside for loop

{

public void show()

{

System.out.println("Inside inner "+(count++));

}

}

Inner in=new Inner();

in.show();

}

}

}

class Test

{

public static void main(String[] args)

{

Outer ot=new Outer();

ot.display();

}

}

**Output:**

Inside inner 0

Inside inner 1

Inside inner 2

Inside inner 3

Inside inner 4

**Example of Inner class instantiated outside Outer class**

class Outer

{

int count;

public void display()

{

Inner in=new Inner();

in.show();

}

class Inner

{

public void show()

{

System.out.println("Inside inner "+(++count));

}

}

}

class Test

{

public static void main(String[] args)

{

Outer ot=new Outer();

Outer.Inner in= ot.new Inner();

in.show();

}

}

**Output**

Inside inner 1

**Annonymous class**

A class without any name is called Annonymous class.

interface Animal

{

void type();

}

public class ATest {

public static void main(String args[])

{

Animal an = new Animal(){ //Annonymous class created

public void type()

{

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

}

};

an.type();

}

}

**Output**

Annonymous animal

Here a class is created which implements **Animal** interace and its name will be decided by the compiler. This annonymous class will provide implementation of **type()** method.

**User defined Exception subclass**

You can also create your own exception sub class simply by extending java **Exception** class. You can define a constructor for your Exception sub class (not compulsory) and you can override the **toString()** function to display your customized message on catch.

class MyException extends Exception

{

private int ex;

MyException(int a)

{

ex=a;

}

public String toString()

{

return "MyException[" + ex +"] is less than zero";

}

}

class Test

{

static void sum(int a,int b) throws MyException

{

if(a<0)

{

throw new MyException(a);

}

else

{

System.out.println(a+b);

}

}

public static void main(String[] args)

{

try

{

sum(-10, 10);

}

catch(MyException me)

{

System.out.println(me);

}

}

}

**Points to Remember**

1. Extend the Exception class to create your own ecxeption class.
2. You don't have to implement anything inside it, no methods are required.
3. You can have a Constructor if you want.
4. You can override the toString() function, to display customized message.

**Method Overriding with Exception Handling**

There are few things to remember when overriding a method with exception handling. If super class method does not declare any exception, then sub class overriden method cannot declare checked exception but it can declare unchecked exceptions.

**Example of Subclass overriden Method declaring Checked Exception**

import java.io.\*;

class Super

{

void show() { System.out.println("parent class"); }

}

public class Sub extends Super

{

void show() throws IOException //Compile time error

{ System.out.println("parent class"); }

public static void main( String[] args )

{

Super s=new Sub();

s.show();

}

}

As the method **show()** doesn't throws any exception while in Super class, hence its overriden version can also not throw any checked exception.

**Example of Subclass overriden Method declaring Unchecked Exception**

import java.io.\*;

class Super

{

void show(){ System.out.println("parent class"); }

}

public class Sub extends Super

{

void show() throws ArrayIndexOutOfBoundsException //Correct

{ System.out.println("child class"); }

public static void main(String[] args)

{

Super s=new Sub();

s.show();

}

}

Output : child class

Because *ArrayIndexOutOfBoundsException* is an unchecked exception hence, overrided **show()** method can throw it.

**More about Overriden Methods and Exceptions**

If Super class method throws an exception, then Subclass overriden method can throw the same exception or no exception, but must not throw parent exception of the exception thrown by Super class method.

It means, if Super class method throws object of **NullPointerException** class, then Subclass method can either throw same exception, or can throw no exception, but it can never throw object of **Exception** class (parent of NullPointerException class).

**Example of Subclass overriden method with same Exception**

import java.io.\*;

class Super

{

void show() throws Exception

{ System.out.println("parent class"); }

}

public class Sub extends Super {

void show() throws Exception //Correct

{ System.out.println("child class"); }

public static void main(String[] args)

{

try {

Super s=new Sub();

s.show();

}

catch(Exception e){}

}

}

Output : child class

**Example of Subclass overriden method with no Exception**

import java.io.\*;

class Super

{

void show() throws Exception

{ System.out.println("parent class"); }

}

public class Sub extends Super {

void show() //Correct

{ System.out.println("child class"); }

public static void main(String[] args)

{

try {

Super s=new Sub();

s.show();

}

catch(Exception e){}

}

}

Output : child class

**Example of Subclass overriden method with parent Exception**

import java.io.\*;

class Super

{

void show() throws ArithmeticException

{ System.out.println("parent class"); }

}

public class Sub extends Super {

void show() throws Exception //Cmpile time Error

{ System.out.println("child class"); }

public static void main(String[] args)

{

try {

Super s=new Sub();

s.show();

}

catch(Exception e){}

}

}

**Chained Exception**

Chained Exception was added to Java in JDK 1.4. This feature allow you to relate one exception with another exception, i.e one exception describes cause of another exception. For example, consider a situation in which a method throws an **ArithmeticException** because of an attempt to divide by zero but the actual cause of exception was an I/O error which caused the divisor to be zero. The method will throw only**ArithmeticException** to the caller. So the caller would not come to know about the actual cause of exception. Chained Exception is used in such type of situations.

Two new constructors and two methods were added to **Throwable** class to support chained exception.

1. **Throwable**( *Throwable cause* )
2. **Throwable**( *String str*, *Throwable cause* )

In the first form, the paramter **cause** specifies the actual cause of exception. In the second form, it allows us to add an exception description in string form with the actual cause of exception.

**getCause()** and **initCause()** are the two methods added to **Throwable** class.

* **getCause()** method returns the actual cause associated with current exception.
* **initCause()** set an underlying cause(exception) with invoking exception.

**Example**

import java.io.IOException;

public class ChainedException

{

public static void divide(int a, int b)

{

if(b==0)

{

ArithmeticException ae = new ArithmeticException("top layer");

ae.initCause( new IOException("cause") );

throw ae;

}

else

{

System.out.println(a/b);

}

}

public static void main(String[] args)

{

try {

divide(5, 0);

}

catch(ArithmeticException ae) {

System.out.println( "caught : " +ae);

System.out.println("actual cause: "+ae.getCause());

}

}

}

**output**

caught:java.lang.ArithmeticException: top layer

actual cause: java.io.IOException: cause

**Synchronization**

At times when more than one thread try to access a shared resource, we need to ensure that resource will be used by only one thread at a time. The process by which this is achieved is called **synchronization**. The synchronization keyword in java creates a block of code referred to as critical section.

Every Java object with a critical section of code gets a lock associated with the object. To enter critical section a thread need to obtain the corresponding object's lock.

**General Syntax :**

synchronized (object)

{

//statement to be synchronized

}

**Why we use Syncronization ?**

If we do not use syncronization, and let two or more threads access a shared resource at the same time, it will lead to distorted results.

Consider an example, Suppose we have two different threads **T1** and **T2**, T1 starts execution and save certain values in a file *temporary.txt* which will be used to calculate some result when T1 returns. Meanwhile, T2 starts and before T1 returns, T2 change the values saved by T1 in the file temporary.txt (temporary.txt is the shared resource). Now obviously T1 will return wrong result.

To prevent such problems, synchronization was introduced. With synchronization in above case, once T1 starts using *temporary.txt* file, this file will be **locked**(LOCK mode), and no other thread will be able to access or modify it until T1 returns.

**Using Synchronized Methods**

Using Synchronized methods is a way to accomplish synchronization. But lets first see what happens when we do not use synchronization in our program.

**Example with no Synchronization**

class First

{

public void display(String msg)

{

System.out.print ("["+msg);

try

{

Thread.sleep(1000);

}

catch(InterruptedException e)

{

e.printStackTrace();

}

System.out.println ("]");

}

}

class Second extends Thread

{

String msg;

First fobj;

Second (First fp,String str)

{

fobj = fp;

msg = str;

start();

}

public void run()

{

fobj.display(msg);

}

}

public class Syncro

{

public static void main (String[] args)

{

*First* ***fnew*** *= new First();*

Second ss = new second(**fnew**, "welcome");

Second ss1= new second (**fnew**,"new");

Second ss2 = new second(**fnew**, "programmer");

}

}

**Output :**

[welcome [ new [ programmer]

]

]

In the above program, object **fnew** of class First is shared by all the three running threads(ss, ss1 and ss2) to call the shared method(*void* **display**). Hence the result is unsynchronized and such situation is called **Race condition**.

**Synchronized Keyword**

To synchronize above program, we must *serialize* access to the shared **display()** method, making it available to only one thread at a time. This is done by using keyword **synchronized** with display() method.

**synchronized** *void* **display** (String msg)

**Using Synchronised block**

If you have to synchronize access to object of a class that has no synchronized methods, and you cannot modify the code. You can use synchronized block to use it.

class First

{

public void display(String msg)

{

System.out.print ("["+msg);

try

{

Thread.sleep(1000);

}

catch(InterruptedException e)

{

e.printStackTrace();

}

System.out.println ("]");

}

}

class Second extends Thread

{

String msg;

First fobj;

Second (First fp,String str)

{

fobj = fp;

msg = str;

start();

}

public void run()

{

**synchronized(fobj)** //Synchronized block

{

*fobj.display(msg)*;

}

}

}

public class Syncro

{

public static void main (String[] args)

{

First fnew = new First();

Second ss = new second(fnew, "welcome");

Second ss1= new second (fnew,"new");

Second ss2 = new second(fnew, "programmer");

}

}

**Output :**

[welcome]

[new]

[programmer]

Because of synchronized block this program gives the expected output