卐

Hi Friends,

Just go through the following small story.. (taken from **"*You Can Win* "** - by Shiv Khera)

Once upon a time a very strong woodcutter asks for a job in a timber merchant, and he got it. The paid was really good and so were the work conditions. For that reason, the woodcutter was determined to do his best. His boss gave him an axe and showed him the area where he was supposed to work. The first day, the woodcutter brought 18 trees "Congratulations," the boss said. "Go on that way!" Very motivated for the boss' words, the woodcutter try harder the next day, but he only could bring 15 trees. The third day he try even harder, but he only could bring 10 trees. Day after day he was bringing less and less trees. The woodcutter thought that "I must be losing my strength". He went to the boss and apologized, saying that he could not understand what was going on.

The boss asked, "When was the last time you sharpened your axe?"

"Sharpen? I had no time to sharpen my axe. I have been very busy trying to cut trees.

If we are just busy in applying for jobs & work, when we will sharpen our skills to chase the job selection process?

My aim is to provide good and quality content to readers to understand easily. All contents are written and practically tested by me before publishing. If you have any query or questions regarding any article feel free to leave a comment or you can get in touch with me on venus.kumaar@gmail.com.

This is just a start, I have achieved till now is just 0.000001n% .

With Warm Regards

Venu Kumaar.S
venus.kumaar@gmail.com

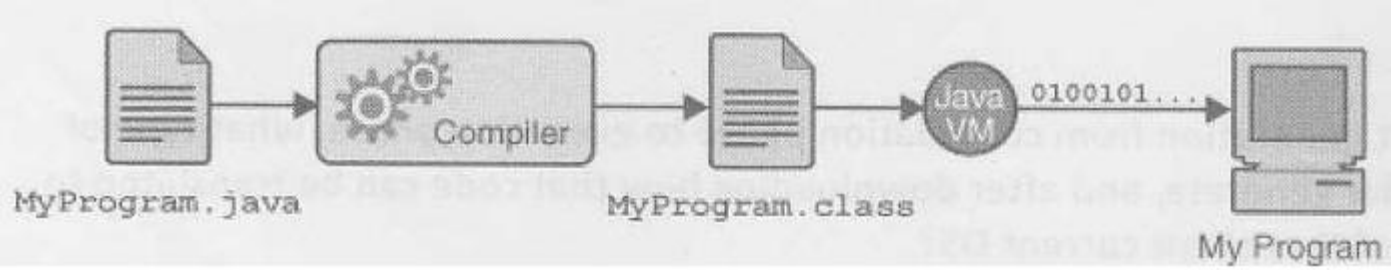
ByteCode → is an intermediate format that can only be understandable by JVM.

JVM → **Java Virtual Machine** - is a software that executes Java byteCode s by converting. bytecodes into machine language of the current operating system's understandable format.

All source code is first written in "**plain text**" ending with the "**.java**" extension. Those source files are then compiled into "**.class**" files by the **javac** compiler. A "**.class**" file does not contain code that is native to out processor; it instead contains "**ByteCode**" - the **native language** of the **JVM**.

The java launcher tool then runs our application with an instance of the **JVM**.

Java execution process as internet application: User downloads "**.class**" file from server system and executes with the **JVM** installed in that **client** computer. Because the **JVM** is available on many different operating systems, the same "**.class**" files are capable of running on Windows, Linux, Solaris or Mac OS.



Java compiler → is responsible to convert java **source code** into **bytecode** and storing these bytecode in a separate file with extension "**.class**".

JVM → is responsible to **execute** this **bytecode**. JVM executes this bytecode by converting them into the **Machine Language** of **current OS**.

Java class **bytecode** run in **all** operating systems, **irrespective** of the **OS** in which the java program is **compiled**.

Java compiler is **OS independent**, because it takes java source code and generates java bytecode, both input and output files are java related files.

JIT compiler → stands for **Just In-Time** compiler, it is the part of the JVM which increases the speed of the execution of the program.

Types of Java softwares:

1) **JDK** → Java Development Kit
2) **JRE** → Java Runtime Environment

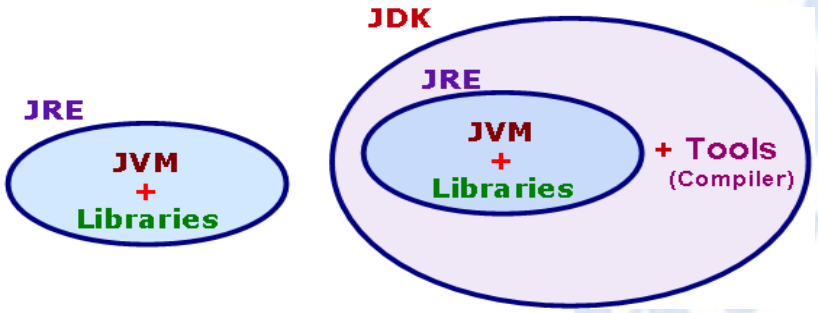
JDK & JRE are called principle products of Java software.

Difference Between JDK, JRE & JVM:

JVM is a **subset** of **JRE**, and **JRE** is the subset of **JDK**.
When we install JDK, JRE is also installed automatically.
JRE software is available as a separate pack, so we can install JRE alone.

→ **JDK has both compiler and JVM**. So using JDK we can develop, compile and execute new java applications and also we can modify, compile and execute already developed applications.

→ **JRE has only JVM**, hence, using JRE we can only execute already developed applications.



Different Environments existed in real time projects development :

1)**Development environment** - Here developers will work to develop new programs(class). Hence we should install JDK in development environment.

2)**Testing environment** - Here testers will work to test the project, means they just execute the project. Hence, JRE installation is enough.

3)**Production environment** - Here end-users will work to use the project, means they just execute the project to complete their transactions. Hence, JRE installation is enough.

****

Types of JVMs:

The Java 2 SDK, SE, contains two implementations of the Java Virtual Machine.

Java HotSpot Client VM : It is the default virtual machine of the Java 2 SDK and Java 2 Run-time Environment. As its name implies, it is tuned for best performance when running applications in a client environment by reducing application start-up time and memory footprint.

Java HotSpot Server VM : It is designed for maximum program execution speed for applications running in server environment. The Java HotSpot Server VM is invoked by using the "-server" command-line option when launching an application.

Syntax: java -server class_name

Some of the features of Java HotSpot technology, common to both VM implementations are as follows:

Adaptive compiler:

→ Applications are launched using a standard interpreter, but the code is then analyzed as it runs to detect performance "bottlenecks", or "hot spots".

→ The Java HotSpot VM compile those performance-critical portions of the code for a boost in performance, while avoiding unnecessary compilation of seldom-used code(most of the program).

→ The Java HotSpot VMs also uses the adaptive compiler to decide, on the fly, how best to optimize compiled code with techniques such as in-lining.

→ The runtime analysis performed by the compiler allows it to eliminate guesswork in determining which optimizations will yield the largest performance benefit.

Rapid memory allocation and Garbage collection: Java HotSpot technology provides for rapid memory allocation for objects, and it has a fast, efficient, state-of-the-art garbage collector.

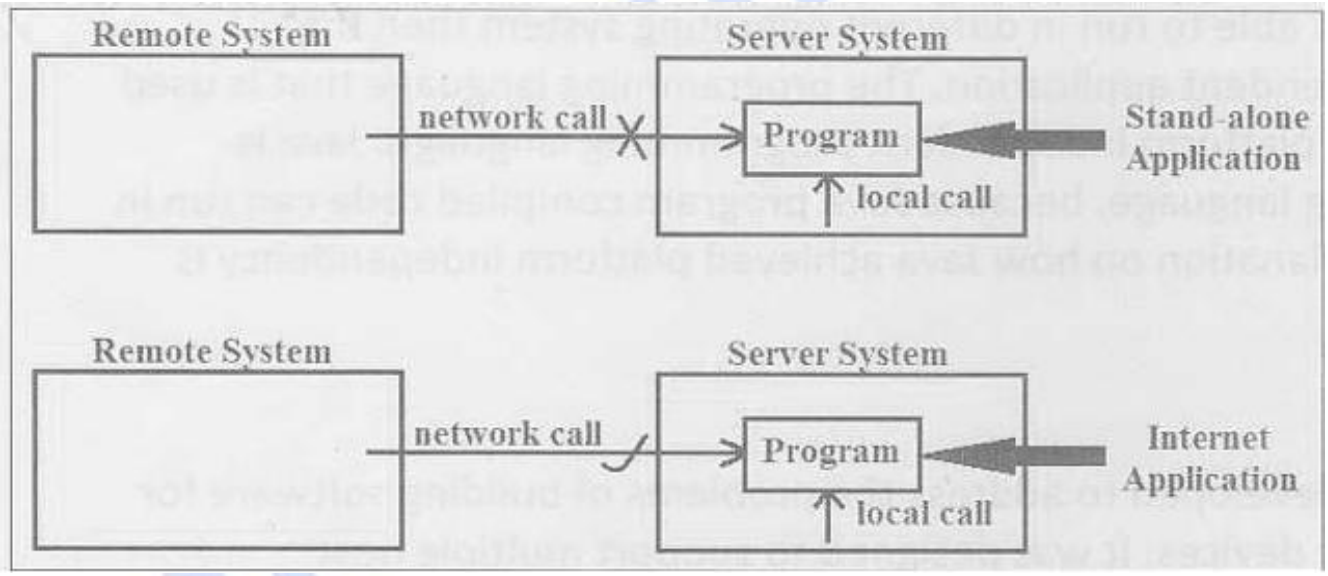
Thread synchronization: The java programming language allows for use of multiple, concurrent paths of program execution, called Threads.

Java HotSpot technology provides a thread-handling capability that is designed to scale readily for use in large, shared-memory multiprocessor servers.

Types of applications: Based on way of execution of programs, all available applications are divided into two types.

1) **Stand-alone applications:** An application that can only executed in local system with local call.

2) **Internet application:** An application that can executed in local system with local call and also from remote computer via network call(request).



****

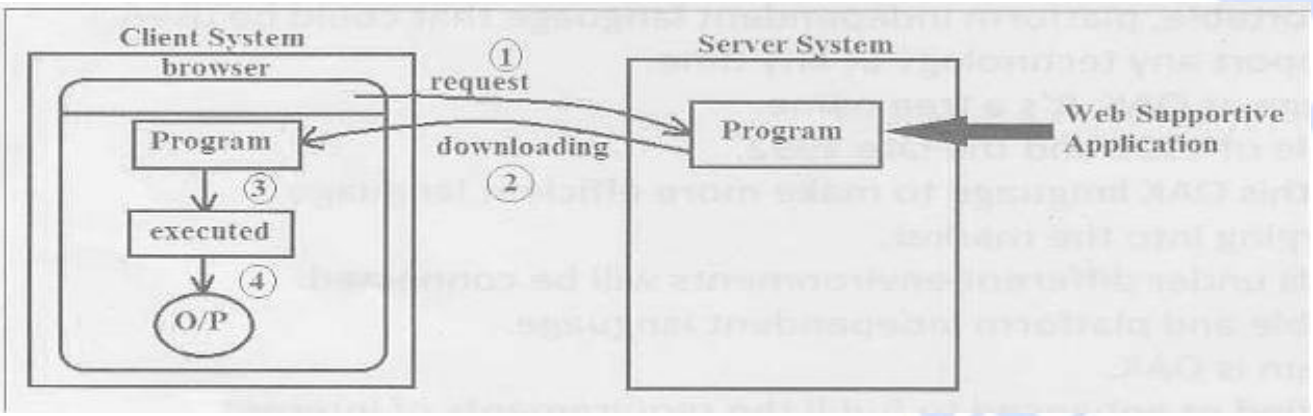
Types of Internet Applications: We have two types of internet applications:

1) Web Supportive Application → executed in client computer
2) Web Application → executed in server computer

1) Web Supportive Application : An application that resides in server system and that is downloaded and executed in client computer via network call is called web supportive application.

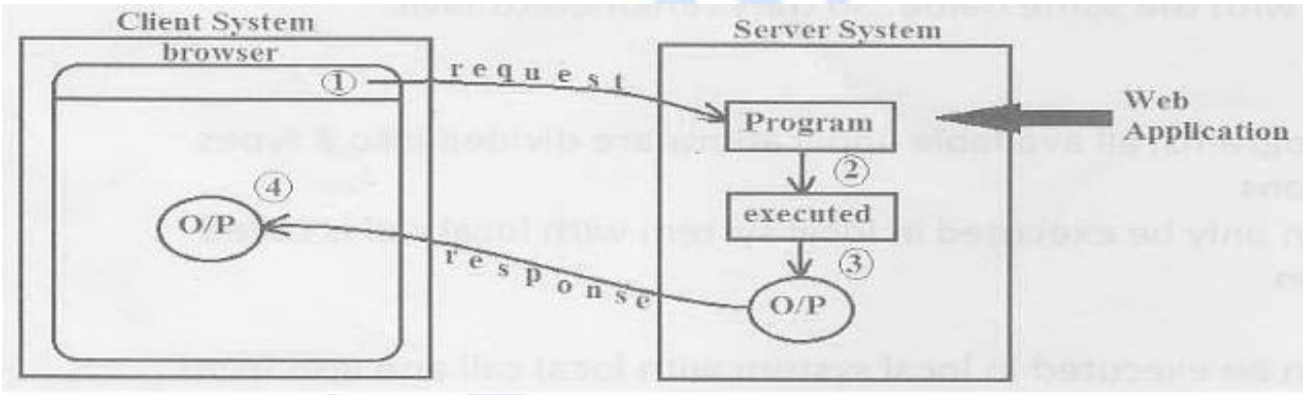
To develop web support application we are using **Applets**. **Applet** is executed in client system browser.

Initially HTML is invented to support Web Supportive Applications. In java, Applet is replacement of HTML.

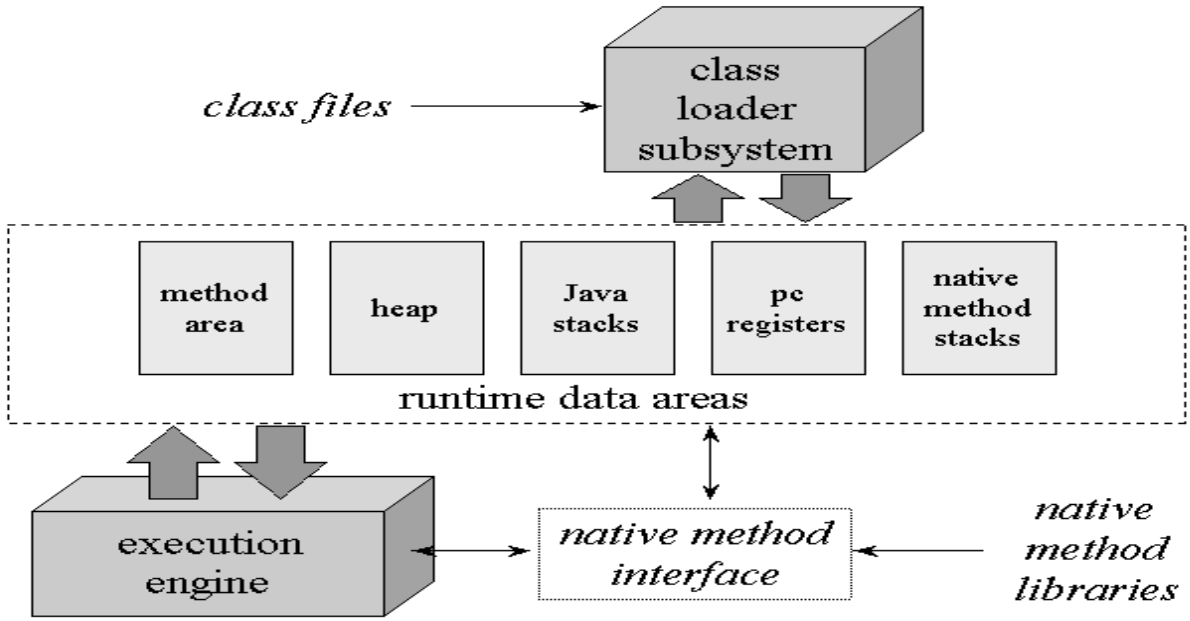


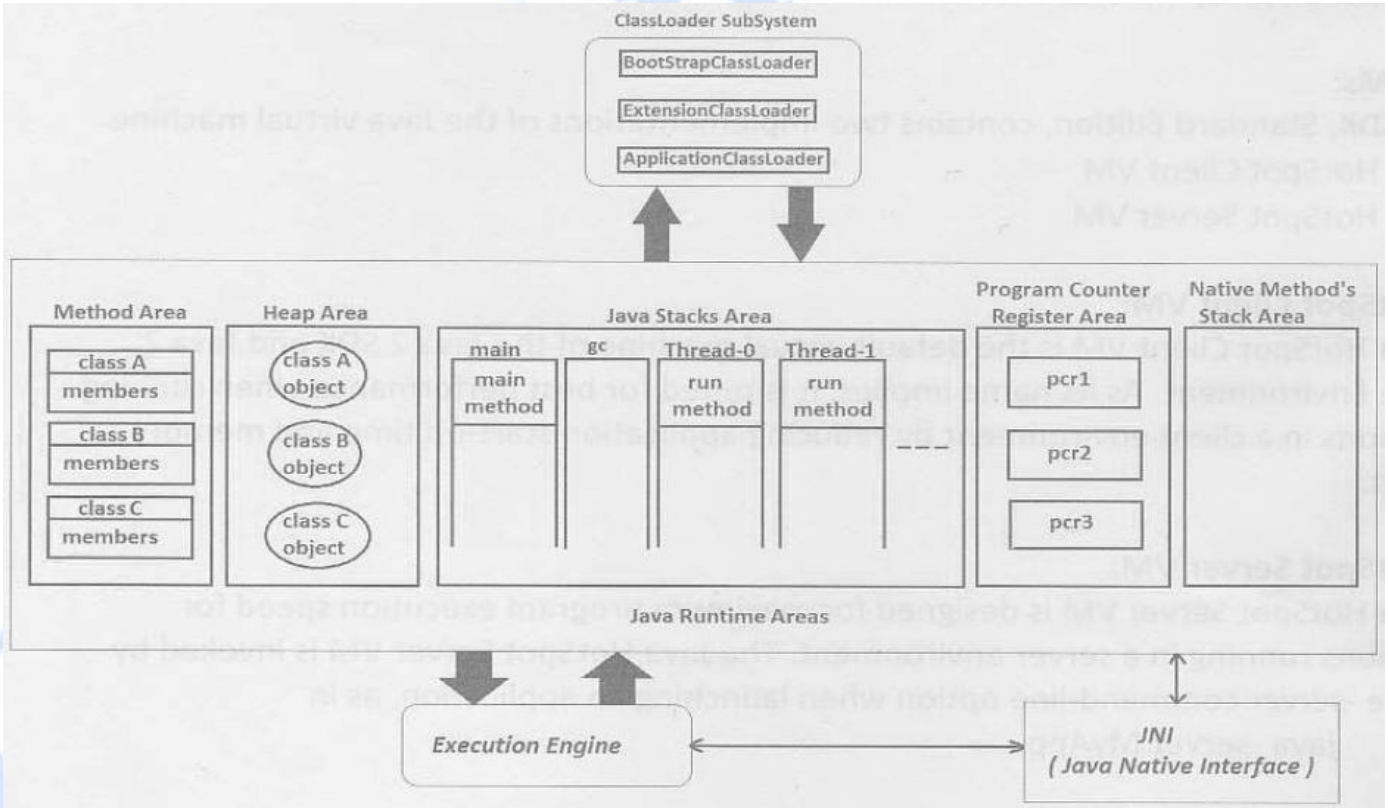
2) Web Application: An application that resides in server system and that is executed directly in server system via network call and sending response(output) back to client is called web application.

To develop web application we are using **Servlet**s and **JSP**s, these are executed in server system.



JVM Architecture





ClassLoader Subsystem: ClassLoader is a class that is responsible to load classes into JVM's method area.

We have basically three types of class loaders:

1) ApplicationClassLoader
2) ExtensionClassLoader
3) BootstrapClassLoader

→ **ApplicationClassLoader** is responsible to load classes from Application Classpath (current working directory). Basically uses Classpath environment variable to locate the class's ".class" file.

→ **ExtensionClassLoader** is responsible to load classes from Extension Classpath, ("%JAVA_HOME%\jre\lib\ext" folder).

→ **BootStrapClassLoader** is responsible to load classes from BootStrap Classpath, ("%JAVA_HOMR%\jre\lib\rt.jar"). These classes are predefined classes.

Loading, Linking and Initialization:

The class loader subsystem is responsible for more than just locating and importing the binary data for classes. It must also verify the correctness of imported classes, allocate and initialize memory for class variables, and assist in the resolution of symbolic references. These activities are performed in a strict order:

1.Loading: finding and importing the binary data for a type
2.Linking: performing verification, preparation, and (optionally) resolution
    2.1)Verification: ensuring the correctness of the imported type
    2.2)Preparation: allocating memory for class variables and initializing the memory to default values
    2.3)Resolution: transforming symbolic references from the type into direct references.
3.Initialization: invoking Java code that initializes class variables to their proper starting values.

ClassLoader Working procedure:

→ When JVM come across a type, it check for that class byte-codes in method area.

→ If it is already loaded it makes use of that type.

→ If it is not yet loaded, it requests class loader subsystem to load that class's byte-codes in method area form that class respective Class path.

→ Then ClassLoader subsystem, first handovers this request to Application Class Loader, then application loader will search for that class in the folders configured in Classpath environment variable.

→ If class is not found, it forwards this request to ExtentionClassLoader, Then it searches that class in ExtenstionClasspath.

→ If class is not found, it forwards this request to **BootStrapClassLoader**, Then it searches that class in **BootStrapClasspath**.

→ If here also class not found, JVM throws an exception
"***java.lang.NoclassDefFoundError***" or "***java.lang.ClassNotFoundException***"

→ If class is found in any one of the class paths, the respective ClassLoader loads that class into JVM's method area.

→ Then JVM uses that loaded class byte-codes to complete the execution.

****

Java Runtime Areas:

Whenever we execute a class by specifying its corresponding class name by using the command java <classname>, the java launcher, java, immediately initiates the Java Run-time Environment(JRE) for the class execution as a layer on top of OS, and further the entire setup is divided in to 5 java Run-time Areas named as

1) Method Area
2) Heap Area
3) Java Stacks Area
4) Program counter registers area
5) Native Methods Stacks area

1) Method Area : All classes' byte-code is loaded and stored in this run-time area, and all static variables are created in this run-time area.

2) Heap Area:
→ It is the main memory of JVM. All objects of classes - non-static variables memory - are created in this run-time area. This run-time area memory is a finite memory.

→ This area can be configured at the time of setting up of run-time environment using non standard options like → java -xms <size> class-name

→ This area can be expandable by its own, depending on the objects creation.

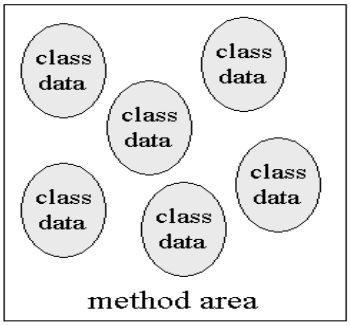
→ Method area and Heap area both are sharable memory areas.

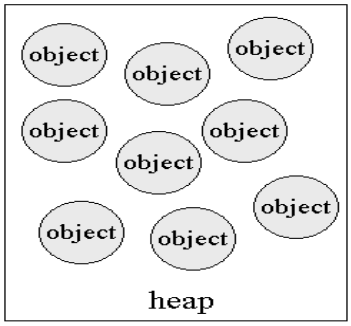
→ Whenever a class instance or array is created in a running Java application, the memory for the new object is allocated from a single heap. As there is only one heap inside a Java virtual machine instance, all threads share it.

→ The Java virtual machine has an instruction that allocates memory on the heap for a new object, but has no instruction for freeing that memory. Usually, a Java virtual machine implementation uses a *garbage collector* to manage the heap.

Runtime data areas shared among all threads

Each instance of the Java virtual machine has one *method area* and *one heap*. These areas are shared by all threads running inside the virtual machine. When the virtual machine *loads a class file*, it *parses information about a type from the binary data contained in the class file*. It places this type information into the *method area*. As *the program runs*, the virtual machine *places all objects the program instantiates onto the heap*.





3)Java Stacks area (JSA):

→ In this run-time area all java methods are executed.

→ In this run-time JVM by default creates two thread, those are:
main thread
garbage collector thread

→ **main thread:** is responsible to execute java methods starts with main method, also responsible to create objects in heap are if it finds "new" keyword in any method logic.

→ **Garbage collector thread:** is responsible to destroy all unused objects from heap area.
Like in c++, in Java we do not have destructors to destroy objects.

→ For each method execution JVM creates separate block in main thread. Technically this block is called Stack Frame. This stack frame is created when method is called and is destroyed after method execution.

Java operations are called "stack based operations(sequential)", because every method is executed only in stack.

4)Program Counter Register Area:

→ In this run-time area, a separate program counter register is created for every thread for tracking that thread execution by storing its instruction address.

5)Native Methods Stacks Area:

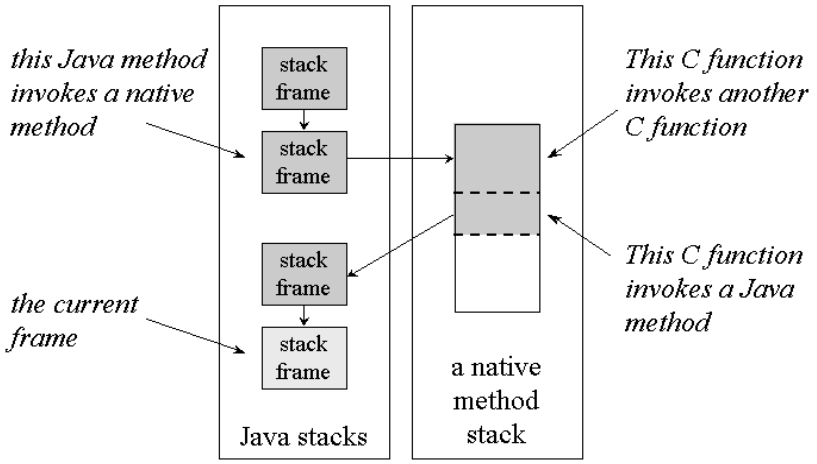
→ In Native Methods Area all java native methods are executed.

→ **Native Method:** It is the java method that has logic in c, c++ is called native method. To create native method, we must place native keyword in its prototype and it should not have body.

```
class NativeTest {

        public static native int add(int x, int y);

        public static void main(String args[]) {
                System.out.println(add(10, 20));
        }
}
```

→ The execution of above program leads **RE:"*java.lang.UnsatisfiedLinkError*"**. Because, we have declared a native method, but we have not defined it in required other language and not linked.

→ The stack for a thread that invokes Java and native methods representation is as follows:



→ **Execution engine:** All executions happening in JVM are controlled by Execution Engine.

→ At the core of any Java virtual machine implementation is its execution engine. In the Java virtual machine specification, the behavior of the execution engine is defined in terms of an instruction set.

****

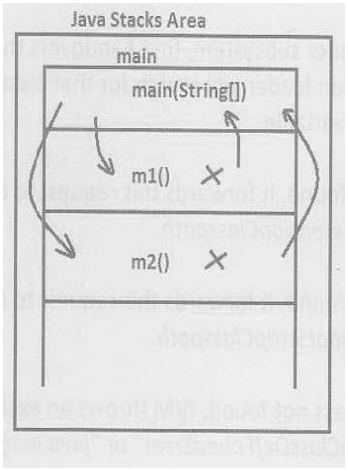
Thread & Stack Frame Architecture:

→ Thread is an independent sequential flow of execution created in **JSA**.

→ Stack Frame is a sub block created inside a thread for executing a method or block, and is destroyed automatically after the completion of that method execution.

→ If this method has any local variables, they are all created inside that method's stack frame, and are destroyed automatically when Stack Frame is destroyed.

```
class ThreadSfa {
	static void m1() {
		System.out.println("m1");
	}

	static void m2() {
		System.out.println("m2");
	}

	static void m3() {
		System.out.println("m3");
	}

	public static void main(String args[]) {
		System.out.println("main");
		m1();
		m2();
	}
}

/*Output:
main
```



```
m1
m2
*/
```

****

StackFrame Architecture:

StackFrame is internally divided into three blocks to create that method's local variables, to store instructions, and to execute its logic. Those are:

1) Variables storage area
2) Instructions storage area
3) Method stack

→ All local variables are created in variables storage area.

→ All method instructions are stored in Instructions storage area

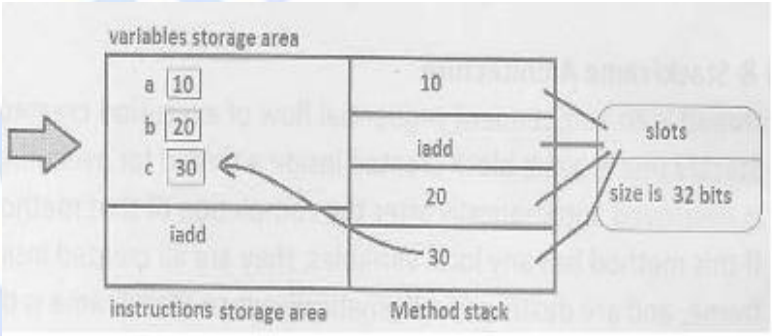
→ Method logic is executed in method stack. To execute method logic, method stack is divided into number of blocks each block is called slot. This slot is 32 bits(4 bytes - int/float data-type depending on the variable type).

→ Due to the slot size - byte, short, char data-types are automatically promoted to int data-type, when they are used in an expression.

→ So, the minimum result type coming out from an expression is int.

Ex1:

```
static void add() {
        int a = 10;
        int b = 10;
        int c = a + b;
System.out.println("Result .." + c);
}
```



Ex2:

```
class Example2 {

   public static int runClassMethod(int i, long l, float f,
      double d, Object o, byte b) {

      return 0;
   }

   public int runInstanceMethod(char c, double d, short s,
      boolean b) {

      return 0;
   }
}
```

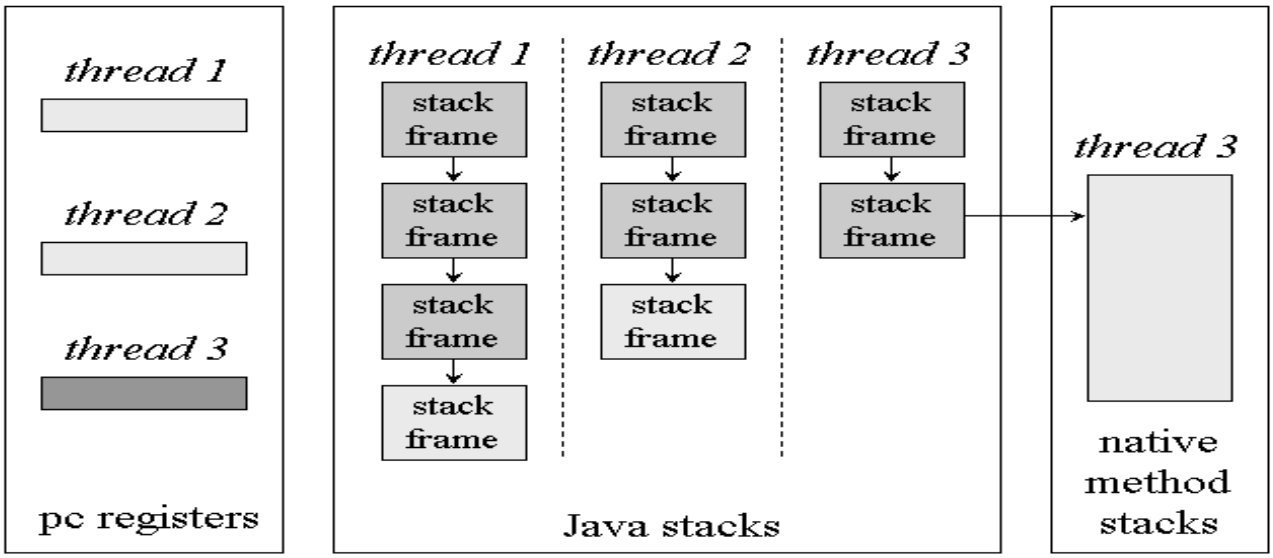
runClassMethod()

index	type	parameter
0	int	int i
1	long	long l
3	float	float f
4	double	double d
6	reference	Object o
7	int	byte b

runInstanceMethod()

index	type	parameter
0	reference	hidden this
1	int	char c
2	double	double d
4	int	short s
5	int	boolean b

The Java virtual machine creates the memory areas for each thread. These areas are private to the owning thread. No thread can access the program counter(pc)register or Java stack of another thread.



Runtime data areas exclusive to each thread is as follows:

****

Conclusion:

→ Complete class byte-codes are stored in method area with **java.lang.Class** object and all static variables get memory in method area.

→ all non-static variables i.e. objects, are created in Heap area when object is created.

→ All java method, blocks and constructors executed in java stack's area in main thread by creating separate stack frame.

→ So, all local variables are created in it's method stack frame.

When we execute java command the following things happened:

→ When "java" command is executed, JVM is created as a layer on top of OS, and is divided in 5 run-time areas.

→ For executing the requested classes, JVM internally performs below three phases.
Those are:

1. **Classloading:** JVM requests class loader to load the class from its respective Classpath. Then class is loaded in method area by using java.lang.Class object memory.

2. **Bytecode verification phase:** After Classloading, BytecodeVerifier verifies the loaded byte-code's internal format. If those bytecodes are not in the JVM understandable format, it terminates execution process by throwing exception "***java.lang.ClassFormatError***". If loaded bytecodes are valid, it allows interpreter to execute those bytecodes.

3. **Execution/interpretation phase:** Then interpreter converts bytecodes into current OS understandable format.

Finally, JVM generates output with the help of OS.

→ If a class is declared as public, file name should be the same as the public class name, else its name can be user defined.

Ex: file: B.java

```
class A{
}
```

javac B.java → valid

file: B.java

```
public  A{
}
```

javac B.java → **CE: class A is public, should be declared in a file named A.java**

In a java file, we can define only one public class, and multiple non-public classes.

→ JVM developer doesn't know what methods should be executed, when they should be executed and in which order.

Hence, it is the developer responsibility to inform about the methods those must be executed by JVM. For this purpose there should be a common method to call these methods, that method should be known to both JVM and developer,and that method prototype should be given by the JVM software designer as it must be called from JVM. That method is **main()** method.

Every full implementation of the java platform gives the following features:

1) **Development Tools:** It provides everything you'll need for compiling, running, monitoring, debugging, and documenting the applications.

2) **API (Application Programming Interface)** : It is the core functionality of the java programming language. It offers a wide array of useful classes ready for use in our own applications.

3) **Deployment Technologies:** The JDK software provides standard mechanisms such as the Java Web Start software and Java Plug-In software for deploying our applications to end users.

4) **User Interface Tool-kits:** The Swing and Java 2D tool-kits make it possible to create sophisticated GUI.

5) **Integration Libraries:** Integration libraries such as the java IDL API, JDBC API, JNDI(Java Naming and Directory Interface) API, Java RMI (Remote Method Invocation), and Java RMI-IIOP(RMI over Internet Inter-ORB(Object Request Broker) Protocol Technology) enable database access and manipulation of remote objects.

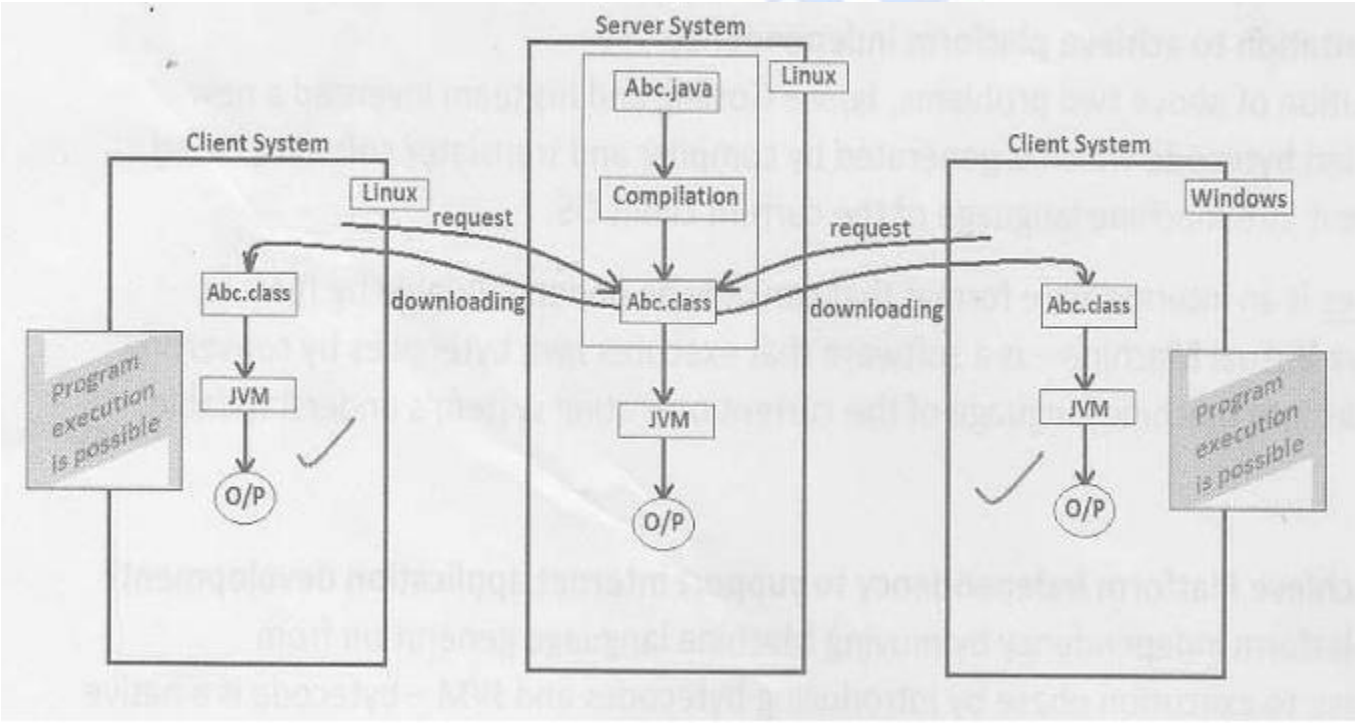
"**java.lang.Object**" is the super class of all types of classes.

Java Platform Architecture:

1)Java Source file **.java**
2)Java **.class** file format
3)Java **API**
4)**JVM**

JVM and **API** both together are called **Java Platform**. It is a software based platform.

Java program Execution process:



****

→ **The main() methods usage:**

public static void main(String[] args) OK

public static void main(String []args) OK

public static void main(String args[]) OK

static public void main(String[] args) OK

public static void main(String... args) OK

public static void main([]String args) CE: Illegal start of type

public static void main(String[] guru) CE: Expected ;

public static int main(String[] args) CE: missing return statement

public static void main(String[5] args) CE:]expected

public static void main(int[] args) RE:Exception in thread "main"
java.lang.NoSuchMethodError: main

public static void main(String args) RE:Exception in thread "main"
java.lang.NoSuchMethodError: main

public static void main() RE:Exception in thread "main"
java.lang.NoSuchMethodError: main

static void main(String[] args) RE:Main method not public.

public void main(String[] args) RE:Exception in thread "main"
java.lang.NoSuchMethodError: main

void main(String[] args) RE:Exception in thread "main"
java.lang.NoSuchMethodError: main

****

```
class First {
        public static void main(String args[]) {
                System.out.println("Hello!");
        }
}
```

→ In JVM we have **First.main(new Sting[0])**

To call main method → main(new String[0]); → in place of 0 we can place any +ve number.

Ex:

```
class A1 {
        public static void main(String args[]) {
                System.out.println("A main");
        }
}
class B {
        public static void main(String args[]) {
```

```
                System.out.println("B main");
                A1.main(new String[0]);
        }
}

/* java A
 Output:
 A main
*/
/* java B
Output:
B main
A main
*/
```

****

→ When we call **main()** in the same class we are not getting CE, but it leads to "***java.lang.StackOverflowError***"

```
class B {
        public static void main(String args[]) {
                System.out.println("B main");
                B.main(new String[0]);
        }
}

B main()
B main()
B main()
.
.
.
Exception in thread "main" java.lang.StackOverflowError
```

```
class B {
        public static void main(String args[]) {
                System.out.println("B main()");
                m1();
        }
        static void m1()
        {
                System.out.println("B m1()");
                B.main(new String[0]);
        }
}
```

Output:

```
B main()
B m1()
B main()
```

```
B m1()
B main()
B m1()
.
.
.
```
Exception in thread "main" ***java.lang.StackOverflowError***

→ **We can override main() method.**

Ex:

```
class B {
	public static void main(String args) {
		System.out.println(" main(String)");
	}

	public static void main(String args[]) {
		System.out.println(" main(String[])");
	}

	public static void main(int args[]) {
		System.out.println("main(int[])");
	}
}
/*Output:
main(String[])
*/
```

Ex:

```
class B {
	public static void main(String args[]) {
		System.out.println(" main(String[])");
		main("Hi");
		main(new int[] { 10 });
	}
	public static void main(String args) {
		System.out.println(" main(String)");
	}
	public static void main(int args[]) {
		System.out.println("main(int[])");
	}
}
/*Output:
main(String[])
main(String)
main(int[])
*/
```

Object class

```
/**@author:VenukumarS  @date:Oct 6, 2012
 * @fileName:HelloWorld.java
```

```
 */
package edu.core;
public class HelloWorld {
        public static void main(String args[]) {
                System.out.println("Hello!");
        }
}
```

javac HelloWorld

→ In the code level the class is not extending any class, while compiling the code it will extend "Object" class.

java HelloWorld

Hello!

javap HelloWorld

→ compiler extended Object class to HelloWorld

javap HelloWorld

Output:

```
Compiled from "HelloWorld.java"

public class HelloWorld extends java.lang.Object {
   public HelloWorld();
   public static void main(java.lang.String[]);
}
```

1) **wait(), notify() & notifyAll()** methods are required for any java class because, if we are not creating any Thread still JVM will create a **Thread** for **main()**... means to start & stop a Thread the logic is required for all the class so these methods are there in "**Object**" class.

wait() → Causes current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.

notify() → Wakes up a single thread that is waiting on this object's monitor.

notifyAll() → Wakes up all threads that are waiting on this object's monitor.

2) **hashCode()** is required for all the program's so **hashCode()** is required. It's a native method.
Returns a hash code value for the object.
To execute the native methods first it should be registered, so to register it will call **registerNatives()** methods in a static block →

```
static{
 registerNatives();
}
```

Program:

```
/**@author:VenukumarS  @date:Oct 6, 2012
 * @fileName:HelloWorld.java
 */
```

```
package edu.core;
 class HelloWorld {
	public static void main(String args[]) {
		HelloWorld helloWorld=new HelloWorld();
		System.out.println(".helloWorld.hashCode()..="+helloWorld.hashCode());
	}
}
/*Output:
java HelloWorld
.helloWorld.hashCode()..=4072869
*/
```

3) **toString():** Returns a string representation of the object.

Program:

```
/**@author:VenukumarS  @date:Oct 6, 2012
 * @fileName:HelloWorld.java
 */
package edu.core;
 class HelloWorld {
	public static void main(String args[]) {
		HelloWorld helloWorld=new HelloWorld();
		System.out.println(".HelloWorld."+helloWorld);
	}
}
/*Output:
java HelloWorld
.HelloWorld.HelloWorld@3e25a5
*/
```

If we print reference variable then it executes Object class toString();

```
/**@author:VenukumarS  @date:Oct 6, 2012
 * @fileName:HelloWorld.java
*/
package edu.core;
class HelloWorld {

	public String toString(){
		return getClass().getName()+"@"+Integer.toHexString(hashCode());
		}

	public static void main(String args[]) {
		HelloWorld helloWorld=new HelloWorld();
		System.out.println("HelloWorld...= "+helloWorld);
	}

}
/*Output:
java HelloWorld
HelloWorld...= HelloWorld@3e25a5
*/
		String s = "Guru";
```

System.out.println(s); → it overrides **toString()**;

4) We can get the state of object copy by using **clone()** & it's required for all classes.

5) To compare two objects we required **equals()**, it required for all classes.

→ equals() is used in most collections to determine if a collection contains a given element.

→ It is the **String.equals()**implementation that determines if two strings are equal.

6) While destroying the object to implement any logic it can be done using **finalize()**.

finalize() → Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.

7) To know the complete information about class then we need to create class object & it's required **getClass()**.Returns the runtime class of an object.

1) To create a class object

Class c1= String.class → if we know the class at compile time then we call ".class".

2) If we know at runtime...

String className = get it from properties file

Class class1 = Class.forName(className);

3) String s = "Guru"; → if object is already there & we want create class object for that

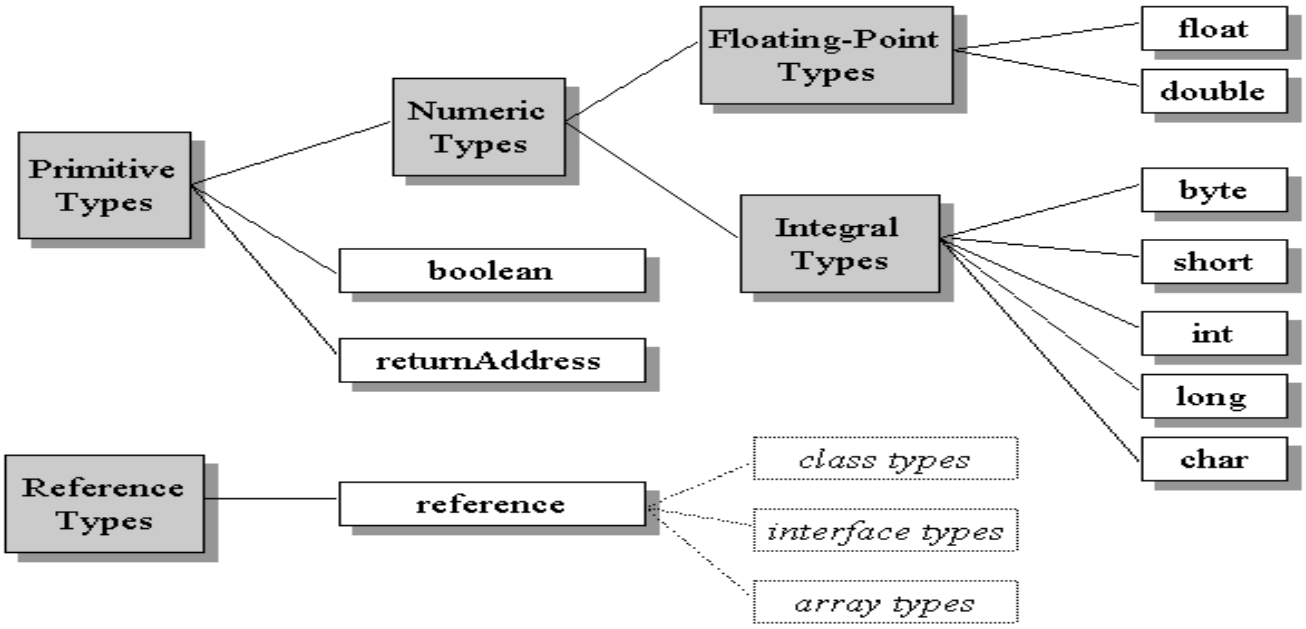
Class c1 = s.getClass();

Program:

```
/**@author:VenukumarS  @date:Oct 6, 2012
 * @fileName:HelloWorld.java
 */
package edu.core;
 class HelloWorld {
	public static void main(String args[]) {
		String s = "Guru";
		Class c1 = s.getClass();

		HelloWorld helloWorld = new HelloWorld();
		Class c2 = helloWorld.getClass();

		System.out.println("s ..Class name = " + c1);
		System.out.println("helloWorld ..Class name = " + c2);
	}

}
/*Output:
 java HelloWrold
s ..Class name = class java.lang.String
helloWorld ..Class name = HelloWorld
*/
```

→ Some common functionality is required for all the objects & those methods are there in Object class.

Data-types



Primitive Types:

The primitive types of the Java programming language other than boolean form the numeric types of the Java virtual machine. The numeric types are divided between the *integral types*: byte, short, int, long, and char, and the floating- point types: float and double. As with the Java programming language, the primitive types of the Java virtual machine have the same range everywhere. A long in the Java virtual machine always acts like a 64-bit signed twos complement number, independent of the underlying host platform.

boolean qualifies as a primitive type of the Java virtual machine, the instruction set has very limited support for it. When a compiler translates Java source code into bytecodes, it uses ints or bytes to represent booleans. In the Java virtual machine, false is represented by integer zero and true by any non-zero integer. Operations involving boolean values use ints. Arrays of boolean are accessed as arrays of byte, though they may be represented on the heap as arrays of byte or as bit fields.

The returnAddress type is used to implement finally clauses of Java programs. It is unavailable to the Java programmer:

Reference Types: Values of type type reference The *class type*, the *interface type,* and the *array type*. All three types have values that are references to dynamically created objects.

The *class type*'s values are references to class instances.

The *array type*'s values are references to arrays, which are full-fledged objects in the Java virtual machine.

The *interface type*'s values are references to class instances that implement an interface. One other reference value is the null value, which indicates the reference variable doesn't refer to any object.

Type	Range
byte	8-bit signed two's complement integer (-2^{7} to $2^{7}-1$, inclusive)
short	16-bit signed two's complement integer (-2^{15} to $2^{15}-1$, inclusive)
int	32-bit signed two's complement integer (-2^{31} to $2^{31}-1$, inclusive)
long	64-bit signed two's complement integer (-2^{63} to $2^{63}-1$, inclusive)
char	16-bit unsigned Unicode character (0 to $2^{16}-1$, inclusive)
float	32-bit IEEE 754 single-precision float
double	64-bit IEEE 754 double-precision float
returnAddress	address of an opcode within the same method
reference	reference to an object on the heap, or null

Primitive datatypes range, bytes, default values

Data Type Name	Size [byte(s)]	Range	Default Value
byte	1	-128 to 127	0
short	2	-32,768 to 32,767	0
int	4	-2,147,483,648 to 2,147,483,647	0
long	8	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807	0
float	4	1.40129846432481707e-45 to 3.40282346638528860e+38	0.0
double	8	4.94065645841246544e-324 to 1.79769313486231570e+308	0.0
char	2	0 to 65,535	One white space
boolean	1	false or true	false
Reference	Depends on PDT	Depends on primitive data types	null

Types of referenced type conversion

Java supports two types of reference type conversions

1) **Upcasting** // automatic conversion
2) **Downcasting** //casting

1) **Upcasting:** the process of "storing subclass object reference in super class reference varia-

ble" is called upcasting. It is the implicit reference type conversion.

Type of Literal	Datatype	default value	Sample values
Integral	int	0	10, 20, 30, 40, ...
	long	0	10L, 20L,30L, 40L, ...
Floating-Point	float	0.0	10.0f, 20.0f, 30.34f, 40.3f, ...
	double	0.0	10.0, 20.0, 30.34, 40.3, ...
Character	char	one space	'a', 'b', 'c', '1', '@', '#', '\n', ...
Boolean	boolean	false	true, false
String	java.lang.String	null	"abc", "bbc", "a", "10", "@#$", ...

```
class A {
}

class B extends A {
}

class C extends B {
}

class D {
}
```

Ex: A a = **new** B();

***Note:** It is not possible to store super class object in sub-class reference variable, it leads to "**in-compatible types**" compile-time error(**CE**).

```
Ex:   A a = new A(); // OK
      B b = a          //CE:incompatible types
      B b = new A() //CE:incompatible types
```

2) **Down-casting:** Retrieving subclass obj ref from super class referenced variable and storing in the same sub-class referenced variable is called down-casting. It is the explicit reference type conversion, casting.

```
Ex:   A a = new B();
      B b = (B) a;
```

In this casting we are informing to compiler that the object stored in "a" is "B" type object. Hence, compiler allows compiling the program as they are having IS-A relation.

Rule in using cast operator:

The cast operator type and source should have inheritance relation else it leads to **CE: "inconvertible types"**.

Ex:
```
A a = new A();
D d = (D) a // CE:inconvertible types
```

java.land.ClassCastException: In casting the object coming from source variable, if it is not compatible with cast operator type **JVM throws** run-time exception(**RE**) **java.lang.ClassCastException**.

In casting **compiler cannot identify the object coming from the source variable**, because it checks only source variable type and cast operator type has IS-A relation or not. So when the source variable is super class type and cast operator is subclass type compiler always compiles this casting. But if the object coming from the refrenced variable is sibling type of cast operator type then JVM throws **java.lang.ClassCastException**.

Ex:
```
Object obj = new A(); //Ok
D d = (D) obj; //RE: ClassCastException
```

From the above **ex: D d = (D) obj**; compiler only checks **obj** and **D** has inheritance relation or not, it does not check the object stored in **obj** variable because compiler checks only type of the variable. since both types have inheritance relation compiler allows above conversion. But, at run-time JVM identifies the object coming from **obj** variable is of type **A** which is not compatible with **D**, hence JVM terminates program execution by throwing **ClassCastException**.

Compiler checks the source variable type & cast operator type has inheritance relation or not.

→ first checks **obj** type "**IS-A**" **D** ==> NO

→ then it checks **D** "**IS-A**" **obj** type ==> YES

Then it compiles this casting statement.

JVM checks the source type object "**IS-A**" cast operator type or not. from the ex. source type object is "**A**" and cast operator type is "**D**", they are siblings so **JVM throws ClassCastException**.

To Over come this CCE(ClassCastException) we should use instanceof operator. :

instanceof operator: This operator returns boolean value by checking source type object with the given class. It works exactly as like cast operator, the only difference is for siblings comparison cast operator throws ClassCastException where as **instanceof** operator return false.

To solve CCE before down-casting we should check object type using **instanceof** operator.

Syntax: referenced variable instanceof classname

referenced variable is the source variable and **class name** is the cast operator type name. It returns **true**, if referenced variable contains object **IS-A** class type, else returns **false**.

source variable type and class name type should have IS-A relation else compiler throws **CE:inconvertiable** types.

Ex:
```
Object obj = new A();
```

```
if(obj instanceof D) {
        D d = (D) obj;
}
```

We do not get CCE because JVM does not execute casting since **instanceof** operator returns false because the source variable object is **A**.

→ "**instanceof**" operator checks whether the given obj is of a particular type or not.

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
  * @fileName:TestInstanceOf.java
 */

class TestInstanceOf {
        public static void main(String[] args) {

                Thread t = new Thread(); // Thread is a child of Object and it impls the
                                                                        // Runnable

                System.out.println("t instanceof Object.....:" + (t instanceof Object));
                System.out.println("t instanceof Thread.....:"
                                + ((t instanceof Thread)));
                System.out.println("t instanceof Runnable.....:"
                                + (t instanceof Runnable));

                // System.out.println("t instanceof String.....:"+(t instanceof
                // String));

        }
}

/*OutPut:
t instanceof Object.....:true
t instanceof Thread.....:true
t instanceof Runnable.....:true
*/
/*for the following statement it shows CE:"incovertible types"

System.out.println("t instanceof String.....:"+(t instanceof String));

TestInstanceOf.java:13: inconvertible types
found   : java.lang.Thread
required: java.lang.String
System.out.println("t instanceof String.....:"+(t instanceof String));
                                   ^
1 error*/
```

Need of Up-casting: In projects up-casting is implemented to develop loosely coupled run-time polymorphic applications means to store all subclasses objects into a single referenced variable and further to execute the invoked method from different subclasses based on the object stored in the referenced variable.

Limitation : In this compiler allows us to invoke only super class members, we cannot invoke subclass specific members with super class type variable. It leads to CE:cannot find symbol, because, compiler checks only variable type but not object stored in super class reference variable.

Ex:

```
class Example {
	void m1() {
		System.out.println("..m1");
	}
}

class Sample extends Example {
	void m2() {
		System.out.println("..m2");
	}
}

class TestCast1 {
	public static void main(String args[]) {
		Example e = new Sample(); // up-casting
		e.m1(); // o/p: ..m1

		e.m2();  // CE:cannot find symbol, because m2() definition is not
				availabe in superclass
	}
}
```

Need of Down-casting: We should implement down-casting to invoke subclass specific members, because when we store subclass object in super-class referenced variable we cannot invoke that subclass specific member using super class referenced variable.

```
class Example {
	void m1() {
		System.out.println("..m1");
	}
}

class Sample extends Example {
	void m2() {
		System.out.println("..m2");
	}
}

class TestCast {
	public static void main(String args[]) {
		Example e = new Sample(); // upcasting
		e.m1(); // o/p: ..m1

		Sample s = e;

		// Sample s = (Sample) e; //downcasting
		s.m2(); // o/p: ..m2
```

```
        }
    }

    /* java TestCast
    ..m1
    ..m2
    */
```

Type Casting Related:

Program:

```
    /**@author:VenukumarS  @date:Oct 6, 2012
     * @fileName:TypeCast.java
     */
    package edu.core;

    public class TypeCast {
        public static Long WhyObjectMehtod1(Long sno) {
            return sno;
        }

        public static String WhyObjectMehtod2(String name) {
            return name;
        }

        public static Object WhyObjectMehtod(Object object) {
            return object;
        }

        public static void main(String args[]) {
            Object obj = new Object();
            Long sno = WhyObjectMehtod1(new Long(1));
            String name = WhyObjectMehtod2("Guru");

            Object snoObj = new Long(1);
            Object nameObj = new String("Guru");

            String nameRef = (String) nameObj; // it's ref type is obj but we are
                                               // able to typecast to String
                                               // because it is pointing to String.

            System.out.println("obj.. = .className." + obj.getClass().getName());

            System.out.println("Name = " + name + " -> .Name length()."
                    + nameRef.length());

        }
    }

    /*Output:
     java TypeCast
```

```
	obj.. = .className.java.lang.Object
	Name = Guru -> .Name length().4
	*/
```

Program:

```
	/**@author:VenukumarS  @date:Oct 6, 2012
	 * @fileName:TypeCast.java
	 */
	package edu.core;
	public class TypeCast {
		public static Long WhyObjectMehtod1(Long sno) {
			return sno;
		}

		public static String WhyObjectMehtod2(String name) {
			return name;
		}

		public static Object WhyObjectMehtod(Object object) {
			return object;
		}

		public static void main(String args[]) {
			Object obj = new Object();
			Long sno = WhyObjectMehtod1(new Long(1));
			String name = WhyObjectMehtod2("Guru");

			Object snoObj = new Long(1);
			Object nameObj = new String("Guru");

			String nameRef = (String) nameObj; // it's ref type is obj but we are
			                                   // able to typecast to String
			                                   // because it is pointing to String.

			System.out.println("obj.. = .className." + obj.getClass().getName());

			System.out.println("Name = " + name + " -> .Name length()."
					+ nameRef.length());

			String nameRef1=(String)obj; //Runtime Exception because we can't
					//assign Object reference to any other reference.

			 System.out.println(".Name length()."+nameRef1.length());

		}
	}

/*Output:
 java TypeCast

obj.. = .className.java.lang.Object
Name = Guru -> .Name length().4
Exception in thread "main" java.lang.ClassCastException: java.lang.Object cannot
 be cast to java.lang.String
```

```
*/
```

Auto Boxing and Unboxing:

Converting **primitive type to Wrapper class object automatically** is called **Auto-Boxing**.

```
Ex: int a = 90;
        Integer i = a; //compiler converts int to Integer object as "Integer i = Integer.valueOf(a)"
    Integer i1 = 50;
```

Converting **Wrapper class object to primitive type automatically** is called **Auto-Unboxing**.

```
        Ex: int i = new Integer(90); //compiler converts Integer object 90 to int value 90
                                    int i1 = new Integer(90).intValue();
```

IS-A Relationship: -It is also known as Inheritance.

→ By using extends keyword we can implement **IS-A** relationship

→ The main advantage of **IS-A** relationship is Re-usability of the code.

```
class P {
        void m1() {
        }
}

class C extends P {
        void m2() {
        }
}
```

Has-A Relationship: - It is also known as "Composition or Aggregation".

-There is no specific keyword to implement **Has-A** relationship, mostly we are using "new" keyword.

→ The main advantage of **Has-A** relationship is re-usability.

```
class Engine {
}

class Car {
        Engine e = new Engine();
}
```

→ The main disadvantage is it increases dependency between the classes and creates maintenance problems.

Composition Vs Aggregation

Composition: In this case whenever container object is destroyed all contained objects will be destroys automatically. i.e., without existing container object there is no chance of existing contained object i.e. Container & Contained objects having strong association.

Ex: University is composed of several departments. whenever university is closed automatically all departments will be closed.

Aggregation: Whenever container object destroyed, there is no guarantee of destruction of contained objects i.e. without existing Container object there may be a chance of existing contained object. i.e. Container object just maintains references of contained objects. This relationship is called Weak Association.

Ex: Several professors will work in the department. Whenever we are closing the department still there may be a chance of existing professors.

Modifiers

Definition: The keywords which define accessibility permissions are called accessibility modifiers.

Levels of Accessibility permissions: java supports four accessibility levels to define accessibility permissions at different levels:

1) only within the class
2) only within the package
3) outside the package but in subclass by using the same subclass object.
4) from all places of project.

To define above four levels we have the following three keywords:

1) **private:** The class members which have private keyword in its creation statement are called private members. Those members are only accessible with in that class.

2) **protected:** The class members which have protected keyword in its creation statement are called protected members. Those members can be accessible with in package from all classes, but from outside package only in subclass that too only by using subclass object → only for non-static protected members. Static members can be accessible by using same class name or by using subclass name.

3) **public:** The class and its members which have public keyword in its creation statement are called public members. Those members can be accessible from all places of java application.

Note:→ If we do not use any of the above 3 accessibility modifiers, **"package"** level is the default accessibility modifier of class and its members. It means that class and its members are not accessible from outside that package.

Accessibility modifiers for class: → public, default.
Accessibility modifiers for class members: → public, default, protected, private.

Modifiers	private	Protected	public	static	final	abstract	native	volatile	transient	synchronized	strictfp
Local Variable	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
Class Level Variable	✓	✓	✓	✓	✓	✗	✗	✓	✓	✗	✗
Method	✓	✓	✓	✓	✓	✓	✓	✗	✗	✓	✓
class	✗	✗	✓	✗	✓	✓	✗	✗	✗	✗	✓
interface	✗	✗	✓	✗	✗	✓	✗	✗	✗	✗	✓

Class modifiers:

When ever we are writing our own class compulsory we have to provide information about our class to the JVM like

→ Whether our class can accessible from any where or not.

→ Whether child class creation is possible for our class or not.

→ Whether instantiation is possible or not etc.

We can specify this information by declaring with appropriate modifier.

→ The applicable modifiers for top-level classes are:
public, <default>, final, abstract, strictfp

→ If we are using any other modifier for top-level class we will get ***CE:***"*modifier xxxx not allowed here*".

Ex:

```
private class Test {
        public static void main(String args[]) {
                System.out.println("class modifier");
        }
}
```

CE:modifier private not allowed here

→ For the Inner classes the following modifiers are allowed:
public, <default>, final, abstract, strictfp, private, protected, static.

Access specifiers Vs Access modifiers:

→ In old languages like c & c++ public, private, protected, & default are considered as access specifiers & all the remaining like final, static are considered as access modifiers.

→ But in java there is no such type of division all are considered as access modifiers.

public class: If a class declared as the public then we can access that class from any where.

Ex:

```
package p1;
```

```
public class A {
        public void m1() {
                System.out.println("Hello!public class modifier--A");
        }
}

javac -d . A.java

package p2;
import p1.A
class B {
        public static void main(String args[]) {
                System.out.println("class B");
                A a = new A();
                a.m1();
        }
}

javac -d . B.java
java p2.B
```

→ If we are not declaring *class A* as *public*, then we will get CE, while compiling B class, saying "*p1.A is not public in p1; can't be accessed from outside package*".

default classes:

→ If a class declared as default then we can access that class only with in that current package i.e., from outside of the package we can't access.

final modifier: final is the modifier applicable for classes, methods & variables.

→ If a method declared as the final then we are not allowed to override that method in the child class. If we try to override it causes "*CE: xx() in Xxxc cannot override xx() in XxxP; overridden method is final*"

```
class P {
        public final void m1() {
                System.out.println("Hello!class-P");
        }
}

class C extends P {
        public final void m1() {
                System.out.println("Hay!class-C");
        }
}

CE: m1() in C cannot override m1() in P; overridden method is final
 public final void m1() {
                  ^
```

→ If a class declared as the final then we can't create child class. If we tried, it causes "*CE: can't inherit from final xx*"

Ex:

```
final class P {
        public final void m1() {
                System.out.println("Hello!class-P");
        }
}

class C extends P {
        public final void m1() {
                System.out.println("Hay!class-C");
        }
}
```

javac C.java

CE: cannot inherit from final P

```
class C extends P {
            ^
```

→ Every method present inside a final class is always final by default. But every variable present in final class need not be final.

→ The main advantage of final keyword is we can achieve security as no one is allowed to change out implementation.

→ But the main disadvantage of final keyword is we are missing key benefits of Oop's inheritance & Polymorphism(Overriding). Hence, if there is no specific requirement never recommended to use final keyword.

abstract modifier: It applicable for classes & methods but not for variables.

→ abstract method: → Even though we don't know about implementation still we can declare a method with abstract modifier. i.e., abstract methods can have only declaration but not implementation. Hence, every abstract method declaration should compulsory ends with ";".

Ex: public abstract void m1();

→ Child classes are responsible to provide implementation for parent class abstract methods.

→ By declaring abstract methods in parent class we can define guidelines to the child classes which describes the methods those are to be compulsory implemented by child class.

→ abstract modifier never talks about implementation.

various illegal combinations of modifiers for methods:

final, static, synchronized, native, strictfp, private.

abstract class: for any java class if we don't want instantiation then we have to declare that class as abstract. i.e., for abstract classes instantiation(creation of object) is not possible. If we try it causes "***CE:*** *Xxxx is abstract; cannot be instantiated*"

```
abstract class Test {
}

Test t=new Test(); //CE: Test is abstract; cannot be instantiated
```

CE: Test is abstract; cannot be instantiated

strictfp(Strict Floating-Point) modifier:

→ This modifier is applicable for methods & classes but not for variables.

→ If a method declared as strictfp all floatingPoint calculations in that method has to follow IEEE754 standard. So, that we will get platform independent result.

→ strictfp method always talks about implementation where as abstract method never talks about implementation. Hence, strictfp-abstract method combination is illegal combination for methods.

```
public abstract strictfp void m1(); // invalid
```

→ If a class declared as strictfp then every concrete method in that class has to follow IEEE 754 standard so, that we will get platform independent results.

→ abstract-strictfp combination is legal for classes but illegal for methods.

Ex:
```
abstract strictfp class Test{
}
```

****

Member modifiers:

public members: If we declare a member as public then we can access that member from anywhere, but corresponding class should be visible(public) i.e. before checking member visibility we have to check class visibility.

Ex:
```
package p1;
class A {
        public void m1() {
                System.out.println("Hello!");
        }
}
javac -d . A.java

package p2;

import p1.A
class B {
        public static void main(String args[]) {
                System.out.println("class B");
                A a = new A();
                a.m1();
        }
}
```

→ From the above program even though m1() is public, we can't access m1() from outside of p1 because the corresponding class A is not declared as "public", if both are public then only we can access.

Ex2:
```
package p1;
```

```
public class A {
        public void m1() {
                System.out.println("Hello!");
        }
}

javac -d . A.java


package p2;

import p1.A
class B {
        public static void main(String args[]) {
                System.out.println("class B");
                A a = new A();
                a.m1();
        }
}
```

default members: If a member declared as the default, then we can access that member only within the current package & we can't access from outside of the package. Hence, default access is also known as "package level access".

private members: If members declared as private then we can access that member only within the current class.

→ abstract methods should be visible in child classes to provide implementation where as private methods are not visible in child classes. Hence, private-abstract combination is illegal for methods.

protected members: If a member declared as protected then we can access that member with in the current package any where but outside package only in child classes.

Protected = default + child of another package

→ With in the current package we can access protected members either by parent reference or by child reference.

→ But from outside package we can access protected members only by using child reference. If we are trying to use parent reference we will get CE.

****

final modifier:

i)final variables: If the instance variable declared as the final compulsory we should perform initialization whether we using or not otherwise we will get CE: variable x might not have been initialized.

```
1) class Test {
        final int x = 10;
   }
```

2)**CE:** variable x might not have been

```
class Test {
```

```
        final int x = 10; //CE: variable x might not have been
    }
```

→ For the final instance variables we should perform initialization before constructor compilation.

1) at the time of declaration:

```
    class Test {
        final int x = 10;
    }
```

2) inside instance block

```
    class Test {
        final int x;
        {
            x = 10;// instance block
        }
    }
```

3)inside constructor

```
    class Test {
        final int x;

        Test() {
            x = 10;
        }
    }
```

ii)final static variables: for the normal static variables it is not required to perform initialization explicitly, JVM will always provide default values.

But, for final static variables we should perform initialization explicitly, otherwise it causes CE: variable x might not be initialized

Rule:

→ for the final static variables we should perform initialization before class loading compilation.

1) at the time of declaration

```
        class Test {
            final static int x = 10;
        }
```

2) Inside static block

```
        class Test {
            final static int x;
            static {
                x = 10;
            }
        }
```

If we initialize any where else we will get CE:

iii) final local variable: Only applicable modifier for local variables is final. when we other modifiers we will get **CE**.

****

static modifier: it applicable for variables & methods but not for classes(but inner class can be declared as static)

→ If the value of a variable is varied from obj to obj then we should go for instance variable. In the case of instance variable for every object a separate copy will be created.

→ If the value of a variable is same for all objects then we should go for static variables. In the case of static variable only one copy will be created at class level and share that copy for every obj of that class.

→ static members can be accessed from both instance & static areas. Where as instance members can be accessed only form instance area directly. i.e., from static area we can't access instance members directly, otherwise we will get CE:

```
class Test {
        int x = 10;
        static int y = 20;

        public static void main(String args[]) {
                Test t1 = new Test();
                t1.x = 888;
                t1.y = 999;
                System.out.println("t1 obj.." + t1.x + "..." + t1.y);// 888 999

                Test t2 = new Test();
                System.out.println("t2 obj.." + t2.x + "..." + t2.y);// 10 999
        }
}
/*Output:
t1 obj..888...999
t2 obj..10...999
*/
```

→ abstract-static is illegal because implementation should be available for static methods compulsory

→ For static methods overloading concept is applicable. Hence within the same class we can declare two main() with different args.

```
        class P {
                public static void main(String args[]) {
                        System.out.println("String main()");
                }
                public static void main(int args[]) {
                        System.out.println("int main()");
                }
        }

/*Output:
 String main()
```

```
*/
```

→ Inheritance concept is applicable for static methods including main() method, hence while executing child class if the child doesn't contain main method then the parent class main method will be executed.

```
class P {
        public static void main(String args[]) {
                System.out.println("String main()");
        }
}

class C extends P {
}

java C

        /*Output:
         String main()
        */

java P
        /*Output:
         String main()
        */
```

****

native: It is applicable only for methods but not for variables and classes.

→ The native methods are implemented in some other languages like c&c++, hence, native methods also known as "**foreign methods**".

→ The main objectives of native are 1) to improve performance of the system 2) to use already existing legacy non-java code.

Pseudo code:

```
class Native {
        static {
                System_loadLibrary("native Library"); // load native library
        }

        public native void m1();// native method declaration
}

class child {
        public static void main(String args[]) {
                Native n = new Native();
                n.m1(); // invoke a native method
        }
}
```

→ For native methods implementation is already available in other languages and we are not responsible to provide implementation.

→ For native methods implementation is available in some other languages where as for abstract methods implementation should not be available , Hence, abstract-native combination is illegal combination for methods.

→ native-strictfp is also illegal because, there is no guarantee of IEEE754 in other languages.

→ The disadvantage of native is it breaks platform independent nature of java class because we are depending on result of platform dependent languages.

****

synchronized modifier: It applicable for methods & blocks. we can't declare class & variable with this.

→ If a method or block declared as synchronized then at a time only one Thread is allowed to operate on the given object.

→ The main disadvantage of synchronized keyword is we can resolve data inconstancy problems. But the main dis-advantage of synchronized is it increases waiting time of Thread and effects performance of the system. Hence, if there is no specific requirement it is never recommended to use synchronized.

****

transient modifier: it applicable only for variables & we can't apply for methods & classes.

→ At the time of serialization, if we don't want to save the value of a particular variable to meet security constraints, then we should go for transient keyword.

→ At the time of serialization JVM ignores the original value of transient variable & default value will be serialized.

****

volatile modifier: It is applicable for variables, but not for methods & classes.

→ If the value of a variable keep on changing such type of variables we have to declare with volatile modifier.

→ If a variable declared as volatile then for every Thread a separate local copy will be created.

→ Every intermediate modification performed by that Thread will takes place in local copy instead of master copy.

→ Once, the value of finalized just before terminating the Thread the master copy value will be updated with local stable value.

→ The main advantage of volatile is we can resolve data inconsistency problems.

→ But, the main disadvantage of volatile keyword is, creating & maintaining a separate copy for every Thread increases complexity of the programming & effects performance of the system. Hence if there no specific requirement if is never recommended. It is outdated keyword.

→ Volatile variable means it's value keep on changes where as final variable means its value never changes. Hence final-volatile combination is illegal combination for variables.

****

Note:

→ The only applicable modifier for local variables is final.

→ The modifiers which are applicable only for variables but not for classes & methods are volatile & transient.

→ The modifiers which are applicable only for method but not for classes & variables native & synchronized.

→ The modifiers which are applicable for top level classes, methods & variables are public, <default>, final.

static block

→ At the time of class loading if we want to perform any activity we have to define that activity inside static block because static blocks will be executed at the time of class loading.

```
class staticBlockTest {
	static {
		System.out.println("staticBlockTest-->Static Block");
	}

	public static void main(String args[]) {

		System.out.println("staticBlockTest...main()");
	}
}

/*Output:
staticBlockTest-->Static Block
staticBlockTest...main()
*/
```

→ With in a class we can take any number of static blocks but all these static blocks will be executed from top to bottom.

```
class staticBlockTest {
	static {
		System.out.println("Static Block..1");
	}
	static {
		System.out.println("Static Block..2");
	}

	public static void main(String args[]) {

		System.out.println("staticBlockTest...main()");
	}

	static {
		System.out.println("Static Block..3");
	}
}
/*Output:
Static Block..1
Static Block..2
Static Block..3
```

```
staticBlockTest...main()
*/
```

Ex: After loading JDBC Driver class we have to register driver with DriverManager but every Driver class contains a static block to perform this activity at the time of Driver class loading automatically, we are not responsible to perform register explicitly.

```
class Driver {
        static {
         Register this Driver with DriverManager
        }
}
```

→ **Advantage:** At the time of class loading compulsory we have to load the corresponding native libraries hence, we can define this step inside static block.

```
class Native {
        static {
                System.loadLibrary("native Library path");
        }
}
```

Static block in parent-child classes:

```
class Base {
(1)→ static int x = 10; ←(7)
(2)→ static {
                m1();←(8)
                System.out.println("Base-->Static Block");←(10)
        }

(3)→ public static void main(String args[]) {
                m1();
                System.out.println("Base main()");
        }

        public static void m1() {
                System.out.println("y=" + y); ←(9)
        }

(4)→ static int y = 20;←(11)
}

class Derived extends Base {
(5)→ static {
                System.out.println("Derived-->Static Block"); ←(12)
        }

(6)→ public static void main(String args[]) {
                System.out.println("Derived...main()");←(13)
        }
}

/*Output:
```

```
java Derived
y=0
Base-->Static Block
Derived-->Static Block
Derived...main()
*/
```

Process:

→ Whenever we are trying to load child class then automatically parent class will be loaded to make parent class members available to the child class. Hence, whenever we are executing child class the following is the flow with respect to static members step.

1) Identification of static members from parent to child. (1 to 6)
2) Execution of static variable assignments & static blocks from parent to child.(7-12)
3) Execution of only child class main method.(If the child class won't contain main() method then automatically parent class main() method will be executed).(13)

```
class Base {
        static int x = 10;
        static {
                System.out.println("Base-->Static Block Start");
                m1();
                System.out.println("Base-->Static Block End");
        }

        public static void main(String args[]) {
                m1();
                System.out.println("Base main()");
        }

        public static void m1() {
                System.out.println("y=" + y);
        }

        static int y = 20;
}

class Derived extends Base {
        static {
                System.out.println("Derived-->Static Block");
        }

}

/*Output:
java Derived
Base-->Static Block Start
y=0
Base-->Static Block End
Derived-->Static Block
y=20
Base main()
*/
```

Note: Whenever we are loading child class automatically parent class will be loaded. But whenever we are loading parent class child class won't be loaded.

Instance control flow:

```
class InstanceBlock {
  (3)→  int x = 10; ←(9)
  (4)→  {
              m1(); ←(10)
              System.out.println("Instance Block..1"); ←(12)
        }

  (5)→  InstanceBlock() {
              System.out.println("Constructor"); ←(15)
        }

  (1)→ public static void main(String args[]) {
  (2)→       InstanceBlock p = new InstanceBlock();
              System.out.println("Main"); ←(16)
        }

  (6)→  public void m1() {
              System.out.println("y="+y); ←(11)
         }

  (7)→  {
              System.out.println("Instance Block..2"); ←(13)
        }
  (8)→  int y = 20; ←(14)
}

/*Output:
java InstanceBlock
y=0
Instance Block..1
Instance Block..2
Constructor
Main
*/
```

Process: Whenever we are creating an object the following sequence of events will be performed automatically:

1) Identification of instance member from top to bottom. (1 to 8)
2)Execution of instance variable assignments & instance blocks from top to bottom(9 to14)
3)Execution of constructor.(15)

Note: static control flow is only one time activity and it will be performed at the time of class loading. But, instance control flow is not one time activity for every object creation it will be executed.

Instance control flow from parent to child:

```
class Parent {
(3)→  int x = 10;←(15)
(4)→  {
```

```
                m1();←(16)
                System.out.println("Parent Instance Block");←(19)
        }

(5)→    Parent() {
                System.out.println("Parent constructor");←(20)
        }

 (1)→   public static void main(String args[]) {
 (2)→                   Parent p = new Parent();
                System.out.println("Parent...main()"); ←(21)
        }

 (6)→           public void m1() {
                System.out.println("y=" + y); ←(17)
        }

(7)→    int y = 20;←(18)

}

class Child extends Parent {
(10)→ int i = 100; ←(22)
(11)→ {
                m2();←(23)
                System.out.println("Child Instance Block..1 ");←(25)
        }

(12)→ Child() {
                System.out.println("Child..Constructor");←(28)
        }

(8)→    public static void main(String args[]) {
(9)→            Child c = new Child();
                System.out.println("Child...main()");←(29)
        }

(13)→           public void m2() {
                System.out.println("j=" + j);←(24)
        }

(14)→           {
                System.out.println("Child Instance Block..2");←(26)
        }
        int j = 200;←(27)

}
/*Output:
java child
y=0
Parent
Parent Instance Block
```

```
Parent constructor
j=0
Child Instance Block..1
Child Instance Block..2
Child..Constructor
Child...main()
*/
```

Process:

Whenever we are creating child class object the following sequence of events will be performed automatically.

1) Identification of instance member from Parent to Child.
2) Execution of instance variable assignments & instance blocks only in Parent class.
3) Execution of Parent class Constructor.
4) Execution of instance variable assignments & instance blocks only in Child class.
5) Execution of Child class Constructor

Object Oriented Program Features

1) Data Hiding:

→ Hiding of the data, so that outside person can't access our data directly.

→ By using private modifier we can implement Data Hiding.

```
class Account {
        private double balance = 1000;
}
```

→ The main advantage of Data Hiding is we can achieve Security.

****

2) Abstraction:

→ Hiding internal implementation details & just highlight the set of services what we are offering, is called "Abstraction".

Ex: → By bank ATM machine, Bank people will highlight the set services what they are offering without highlighting internal implementation. This concept is nothing but Abstraction.

→ By using interfaces & abstract classes we can achieve abstraction.

The main advantages of abstraction are:

1) We can achieve Security as no one is allowed to know our internal implementation.
2) Without effecting outside person we can change our internal implementation. Hence, enhancement will become very easy.
3) It improves modularity of the application.

****

3) Encapsulation:

→ Encapsulation data & corresponding methods(behaviour) into a single module is called "Encapsulation".

→ If any java class follows Data Hiding & Abstraction such type of class is said to Encapsulated class.

Encapsulation = Data Hiding + Abstraction.

```
class Account {
        private double balance;

        public double getBalance() {
                // validate user
                return balance;
        }

        public void setBalance(double balance) {
                // validate user
                this.balance = balance;
        }
}
```

→ Hiding data behind methods is the central concept of Encapsulation.

→ The main advantages of Encapsulation are :

1) We can achieve Security.
2) Enhancement will become very easy.
3) Improves modularity of the application.

→ The main disadvantage of Encapsulation is it increases the length of the code & slows down execution.

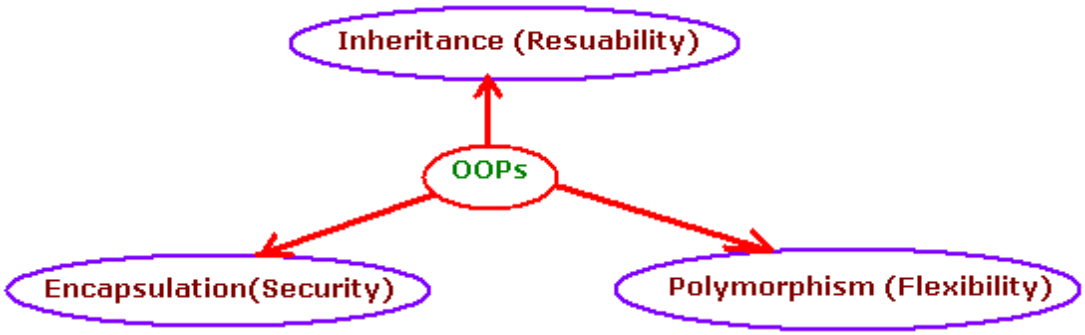
Tightly Encapsulated class:

→ A class is said to be tightly encapsulated iff every data member declared as the private.

→ whether the class contains getter & setter methods or not & whether those methods declared as public or not, these are not required to check.

```
class A {
        private int balance;

        public int getBalance() {
                return balance;
        }
}

class Parent {
        private int x;
}

class Child extends A {
        private int y;
}
```

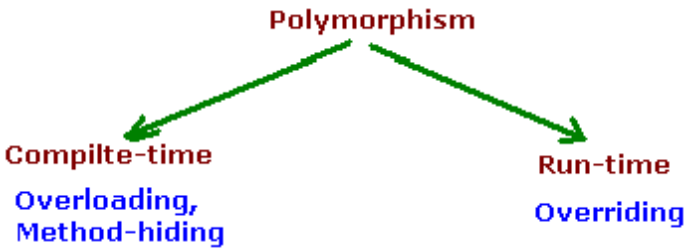
Note: If parent class is not tightly encapsulated then no child class is Tightly encapsulated.



****

4) Polymorphism:

Polymorphism means many forms. In java using Overloading & Overriding concepts we can implement Compile time and /or run-time Polymorphism.



Overloading:

→ Two methods are said to overloaded iff method names are same but arguments are different.

→ In java two methods having the same name with different arguments is allowed & these methods are considered as overloaded methods.

→ Having overloading concept in java simplifies the programing.

→ In overloading more specific version will get highest priority.

→ It is also known as compile-time binding or static binding or early binding.

***In overloading method resolution child-arguments will get more priority than parent. argument.

```
class Test {
        public void m1(Object o) {
                System.out.println("Object Version");
        }

        public void m1(String s) {
                System.out.println("String Version");
        }

        public static void main(String args[]) {
                Test t = new Test();
                t.m1(new Object());// Object version
```

```
                t.m1("Guru");// String version
                t.m1(null); // String version

        }
}

/* Output:
Object Version
String Version
String Version
*/
```

****

Overriding :

→ Overriding is also known as "runtime polymorphism or dynamic polymorphism or Late binding".

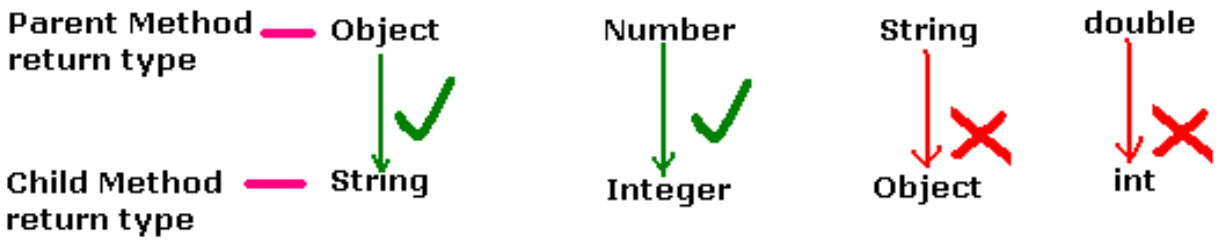
→ Overriding method resolution is also known as "Dynamic method dispatch".

Rules for Overriding:

1) In overriding method names & arguments must be matched i.e. method signature must be matched.

2) In overriding return type must be matched. But, this rule is applicable until 1.4v, from 1.5v onwards co-variant return types are allowed, according to this, child method return type need not be same as parent method return type, its child classes also allowed.

```
class Parent {
        public Object m1() {
                return null;
        }
}

class Child extends Parent {
        public String m1() // It is valid in 1.5v but invalid in 1.4v
        {
                return null;
        }
}
```



→ co-variant return type concept is applicable only for object type but not for primitive types.

3) We can't override parent class final method.

4) Private methods are not visible in child classes, hence overriding concept is not applicable for private methods.

5)Based on our requirement we can declare the same parent class private method in child class also it is valid but it is not overriding.

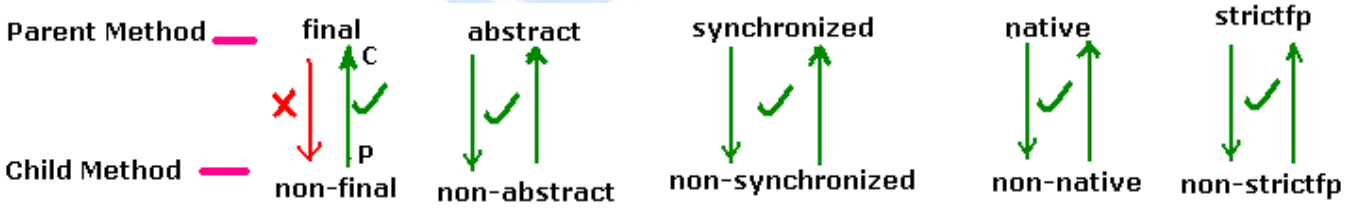
```
class Parent {
        private void m1() {
                System.out.println("Parent-m1()");
        }
}

class Child extends Parent {
        private void m1()// it is not overriding
        {
                System.out.println("Child-m1()");
        }
}
```

6) For parent class abstract methods we should override in child class to provide implementation.

7) We can override parent class non-abstract method as abstract in child class to stop parent class method implementation availability to the child classes.

```
class Parent {
        public void m1() {
                System.out.println("Parent-m1()");
        }
}

abstract class Child extends Parent {
        public abstract void m1();//
}
```

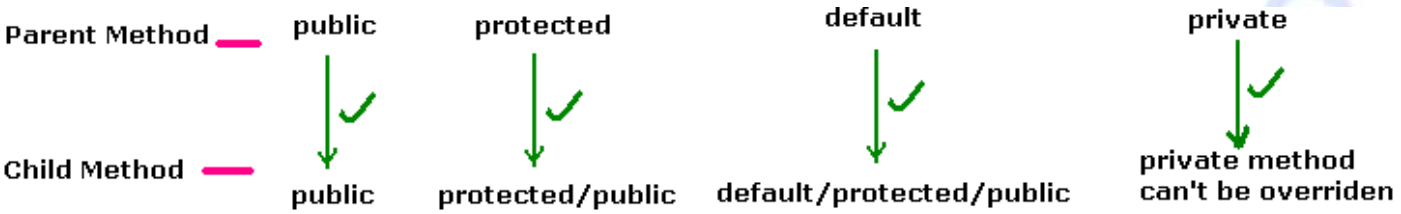
→ The following modifiers won't play any restrictions in overriding:



native, synchronized, strictfp

→ While overriding we can't decrease scope of the modifier , but we can increase. The following are various acceptable overridings.

private < default < protected < public



→ Decreasing the scope of the modifier in Child.

```
class Parent {
        public void m1() {
        }
}

class Child extends Parent {
        protected void m1() { // CE:Cannot reduce the visibility of the
                                 inherited method from Parent
        }
```

CE:Cannot reduce the visibility of the inherited method from Parent

→ Increasing the scope of the modifier in Child.

```
class Parent {
        private void m1() {
        }
}

class Child extends Parent {
        protected void m1() {
        }
}
```

→ This rule is applicable while implementing interface methods also.

→ Whenever we are implementing any interface method compulsory it should be declared as public, because every interface method is public by default.

EX:

```
interface Interf
{
void m1();
}
class Test7 implements Interf
{
void m1(){} //CE:Cannot reduce the visibility of the inherited method
              from Interf
}
```

EX:

```
interface Interf {
        void m1();
}

class Test implements Interf {
```

```
        public void m1() {
        }
}
```

8) If child class method throws some checked exception then compulsory parent class method should throw the same checked exception or it's parent class exception, otherwise we will get CE.

```
class Parent {
        public void m1() {
        }
}

class Child extends Parent {
        public void m1() throws Exception // CE:Exception Exception is not compatible
                                          with throws clause in Parent.m1()
        {
        }
}
```

CE: m1() in Child can't override m1() in Parent; Overridden method doesnot throw Exception.

Valid:

→ 1)Parent: **public void** m1() **throws** IOException
child: **public void** m1()

→ 2)Parent:**public void** m1() **throws** Exception
child:**public void** m1() **throws** IOException

→ 3)Parent:**public void** m1() **throws** IOException
child:**public void** m1() **throws** FileNotFoundException, EOFException

→ 4)Parent:**public void** m1() **throws** IOException
child:**public void** m1() **throws** ArithmeticException, NullPointerException

→ 5)Parent:**public void** m1()
child:**public void** m1() **throws** ArithmeticException, NullPointerException

Invalid:

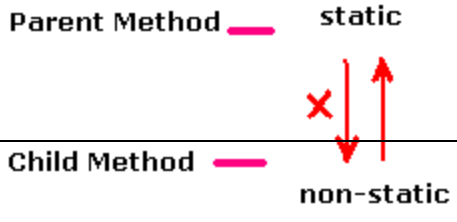
→ 1)Parent:**public void** m1()
child:**public void** m1() **throws** IOException

→ 2)Parent:**public void** m1() **throws** IOException
child:**public void** m1() **throws** Exception

→ 3)Parent:**public void** m1() **throws** IOException
child:**public void** m1() **throws** EOFException, InterruptedException

Overriding with respect to static method:

→ We can't override a static method as non-static.



```
class Parent {
        public static void m1() {
        }
}

class Child extends Parent {
        public void m1() // CE:
        {
        }
}
```

CE: m1() is can't override m1() in Parent; Overridden method is static

→ Similarly, we can't override non-static method as static.

→ If both parent & child class method is static then we won't to get any CE. It seems to be overriding is happen, but it is not overriding. it is "**Method Hiding**".

```
class Parent {
        public static void m1() {
        }
}

class Child extends Parent {
        public static void m1() // Method Hiding
        {
        }
}
}
```

Method Hiding vs Overriding

Method Hiding :

1) Both methods should be static
2) Method resolution takes care by compiler based on reference type
3) It is considered as compile-time polymorphism or static polymorphism or early binding.
4)

Overriding:

1) Both methods should be non-static
2) Method resolution always takes care by JVM based on Runtime object.
3) It is considered as Runtime Polymorphism or dynamic Polymorphism or Late Biding.

```
class Parent {
        public static void m1() {
                System.out.println("Parent-m1()");
        }
}

class Child extends Parent {
        public static void m1() // Method Hiding
```

```
        {
                System.out.println("Child-m1()");
        }
}

class Test {
        public static void main(String args[]) {
                Parent p = new Parent();
                p.m1(); // Parent-m1()

                Child c = new Child();
                c.m1(); // Child-m1();

                Parent p1 = new Child();
                p1.m1(); // Parent
        }
}

/*Output:
 java Test
Parent-m1()
Child-m1()
Parent-m1()
*/
```

→ If *both methods are non-static* then it will *become overriding*:

```
class Parent {
        public void m1() {
                System.out.println("Parent-m1()");
        }
}

class Child extends Parent {
        public void m1() // Method Hiding
        {
                System.out.println("Child-m1()");
        }
}

class Test {
        public static void main(String args[]) {
                Parent p = new Parent();
                p.m1(); // Parent-m1()

                Child c = new Child();
                c.m1(); // Child-m1()

                Parent p1 = new Child();
                p1.m1(); // Child-m1()
        }
}

/*Output:
```

```
java Test
Parent-m1()
Child-m1()
Child-m1()
*/
```

Difference between Overloading and Overriding :

Property	Overloading	Overriding
1) Method Names	must be same	must be same
2) arguments	must be different(at least order)	must be same(including order
3) Method signature	must be different	must be same
4) return type	no restrictions	must be same until 1.4v. But from 1.5v onwards co-variant return types are allowed.
5) private, static, & final methods	can be overloaded	can't be overridden
6) Access modifiers	no restrictions	we can't decrease
7) throws clause	no restrictions	size & level of checked exceptions we can't increase, but we can decrease. But no restriction for unchecked exceptions.
8) Method resolution	always takes care by compiler based on reference type	always takes care by JVM based on runtime object
9) Also known as	Compile-time polymorphism or Static polymorphism or Early binding	Runtime polymorphism or Dynamic polymorphism or Late binding

****

5)Class:

A class is a top level block that is used for grouping variables and methods for developing logic.

Types of classes: In java we have five types of classes.
1.**Interface** 2.**Abstract class** 3. **Concrete class** 4. **Final class** and 5. **enum**

5.1) **interface:** It is a *fully unimplemented class* used for declaring a set of operations of an object. It contains only public static final variables and public abstract methods.

It is always *created as a root class in object creation process*. We must design an object starts with interface if its operations have different implementations.

5.2) **abstract class:** It is a *partially implemented class* used for developing some of the operations of an object which are common for all next level subclasses.

It contains both *abstract, concrete methods* including variables, blocks and constructors.

It is always created *as super class next to interface in object's inheritance hierarchy* for implementing common operations from interface.

5.3) **Concrete class:** It is a *fully implemented class* used for implementing all operations of an object. It contains only concrete methods including *variables, blocks, constructors*. It is always created as sub class next to abstract class or interface in object's inheritance hierarchy.

concrete class modifiers: public, final, abstract, strictfp.

members allowed inside concrete class: static and non-static methods, variables, block, constructors, main()

5.4) **final class:** It is concrete class which is declared as final is called final class. It does not allows subclasses. So it is the last subclass in an object's inheritance hierarchy.

5.5) **enum:** enum is a final class.

→ It is subclass of **java.lang.Enum**

→ It is a abstract class which is the default super class for every enum type classes. Also it is implementing from Comparable and Serializable interfaces.

→ Since every enum type is comparable so we can add its objects to collection TreeSet object, also it can be stored in file as it is serializable type.

→ All named constants created inside enum are referenced type the the current enum type. compiler adds the missing code and

→ These enum type variables are initialized in static block with the enum class object by using(String, int) parameter constructor.

6)Inheritance: A *concrete class* can be derived from another *concrete class* by using "**extends**" keyword.

→ We can derive a *class* from *interface* using "**implements**" keyword. We can derive an *interface* from *interface* using "**extends**" keyword.

In two ways we can access one class members from another class:

1) by using its *class name* or *object* name(with **HAS-A** relation)

2) By using *inheritance*- means by using subclass or object(with **IS-A** relation)

EX: for **HAS-A** relation

```
class Example1 {
        static int a = 10;
        int x = 20;
}

public class HasATest {
        public static void main(String args[]) {
                //accessing static members using class name
                System.out.println("Static Example1.a..= " + Example1.a);

                Example1 e = new Example1();
```

```
                //accessing non-static members using object
                System.out.println("Non-static.. e.x..= " + e.x);
        }
}
/*
Output:
        java HasATest
        Static Example1.a..= 10
        Non-static.. e.x..= 20
*/
```

****

EX: for **IS-A** relation

```
public class IsATest extends Example1 {
        public static void main(String args[]) {
                //accessing static members using class name
                System.out.println("Static Example1.a..= " + Example1.a);

                Example1 e = new Example1();
                //accessing non-static members using object
                System.out.println("Non-static.. e.x..= " + e.x);
        }
}
/*
Output:
        java IsATest
        Static Example1.a..= 10
        Non-static.. e.x..= 20
*/
```

****

compiler and JVM activities in Compiling and Executing a class with inheritance:

Compiler Activities: While compiling a class with inheritance relationship, compiler *first compiles* the *super class* and t*hen it compiles the subclass*. It gets super class name from *extends* keyword.

In *finding method or variable definitions*, compiler always *first search in sub class*. If it is not available in subclass, *it searches in its immediate parent class*, next in its grand parent class. If there also it is not available compiler throws **CE:**"cannot find symbol".

JVM activities: JVM also executes the *invoked members from subclass*, if that member definition is not available in sub class, *JVM executes it from super class where ever it is appeared first*.

When *subclass is loaded its entire super classes are also loaded*, and also when subclass object is created all its super class's non-static variables memory is created in subclass object.

So, the order execution of static and non-static members with inheritance is:

→ class loading order is from super class to sub class

→ static variable and static blocks are identified and then executed from super class to subclass

→ Object creation is also starts from super class to sub class

→ non-static variable, non-static block and invoked constructor are identified and executed from super class to subclass.

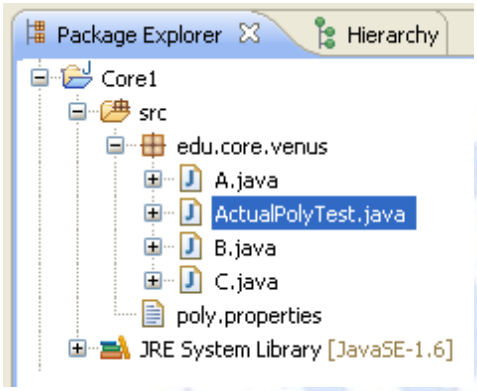
→ For executing method its definition searching is starts from sub class to super class

→ Invoked method is executed from sub class, if it not defined in sub clas, it is executed from super class, but not from both.

→ JVM can load super class when subclass loaded by using extends keyword.

→ JVM can create super class non-static variables memory and initialize it in subclass **object using super().**

****



Actual Polymorphism in coding:

Program:

A.java

```
/**@author:VenukumarS  @date:Oct 10, 2012
 * @fileName:A.java
 */
package edu.core.venus;

abstract class A {
	abstract public void x();
}
```

B.java

```
/**@author:VenukumarS  @date:Oct 10, 2012
 * @fileName:B.java
 */
package edu.core.venus;

public class B extends A {

	public void x() {
		System.out.println(".B.x().");
	}

}
```

C.java

```
/**@author:VenukumarS  @date:Oct 10, 2012
 * @fileName:C.java
 */
package edu.core.venus;

public class C extends A {
	public void x() {
		System.out.println(".C.x().");
	}
}
```

poly.properties

```
className=edu.core.venus.C
```

ActualPolyTest.java

```
/**@author:VenukumarS  @date:Oct 10, 2012
 * @fileName:ActualPolyTest.java
 */
package edu.core.venus;

import java.io.IOException;
import java.util.Properties;

public class ActualPolyTest {
	private static Properties properties = new Properties();
	static {
		try {
		properties.load(edu.core.venus.ActualPolyTest.class
				.getClassLoader().getResourceAsStream("poly.properties"));
		} catch (IOException ie) {
			throw new InstantiationError(ie.getMessage());
		}
	}

	public static void main(String args[]) throws Exception {
		String className = properties.getProperty("className");
		A a = (A) Class.forName(className).newInstance();
		a.x();
	}
}

/* Output:
	java ActualPolyTest
	.C.x().
 */
```

→ →

1) Change **C** to **B** and **B** to **C** in **properties** file.
2) So here it's calling different implementation without changing the java code.

→ The logic is implemented by Container. Container can call an implementation.

A1.java

```
/**@author:VenukumarS  @date:Oct 10, 2012
 * @fileName:A1.java
 */
package com.core.PolyTest1;

public class A1 {

	public void x() {
		System.out.println(".B.x().");
	}

}
```

B1.java

```
/**@author:VenukumarS  @date:Oct 10, 2012
 * @fileName:B1.java
 */
package com.core.PolyTest1;

public class B1 extends A1 {

	public void x() {
		System.out.println(".B1.x().");
	}

}
```

PolyTest1.java

```
/**@author:VenukumarS  @date:Oct 10, 2012
 * @fileName:PolyTest1.java
 */
package com.core.PolyTest1;

public class PolyTest1 {
	public static void main(String args[]) {
		// Technically Correct but understanding wise it's wrong
		A1 a = new B1();
		a.x();
		B1 b = new B1();
		a.x();
		b.x();
	}
}
```

If we know the subclass at runtime then we need to assign the object to super class reference.

→ 2.3) **Compile** → If class is calling any method then first we need to verify the method in the same class, if not then verify in the super class.

→ 2.4) **Runtime** → if class is calling any method then first we need t verify the method in the sub

class (object created class), if not then verify in the super class.

Runtime means resolve the name using "**.properties**" or "**.xml**".

Ex: In Servlets we configure the servlet className in the web.xml, so containers will read the xml & will create the object using Reflection and then will assign the object to servlet request.

From the PolyTest1 Program:

→ 2.1)A1 a = **new** B1(); → compile time it will check x() method is in A1 class & it's super class, if the method is not available then it will throw CE:

→ 2.2)Runtime it will check the method in the object created class(means sub class - B1 class), if it's not available then it will check in super class.

****

Servlet s = (Servlet) Class.forName(*HelloWorldServlet*).new Instance();
s.service();

→ The container will check service method is there in Servlet interface or not at compile time→ But at run-time it will call *HelloWorldServlet* service method, because *HelloWorldServlet* object will created at runtime.

A2.java

```
/**@author:VenukumarS  @date:Oct 10, 2012
 * @fileName:A2.java
 */
package com.core.PolyTest2;

abstract class A2 {

	public void x() {
		System.out.println(".A2.x()");
		y();
	}

	public abstract void y();

	// public void y();
	public void z() {
		System.out.println(".A2.z()");
	}
}
```

B2.java

```
/**@author:VenukumarS  @date:Oct 10, 2012
 * @fileName:B2.java
 */
package com.core.PolyTest2;

class B2 extends A2 {
	public void y() {
		System.out.println(".B2.y()");
```

```
                z();
            }
        }
```

PolyTest2.java

```
        /**@author:VenukumarS  @date:Oct 10, 2012
         * @fileName:PolyTest2.java
         */
        package com.core.PolyTest2;

        public class PolyTest2 {
            public static void main(String args[]) {

                A2 a = new B2();
                a.x();

                // System.out.println("\n...B2 Obj...");
                // B2 b = new B2();
                // difference between a.y() & b.y()
                // a.y();
                // b.y();
            }
        }

        /*Output:
            java PolyTest2
        .A2.x().
        .B2.y().
        .A2.z().
        */
```

→ 1) comment the "**y()**" code in **A2** class & verify, then it will throw **CE:** cannot find symbol
symbol : method y()
location: class A2

→ 2) If we are calling a method then it must be there in the same class or it's super class to compile the code. Here if we comment "y()" method then there in no "y()" signature in A or it's super class

How to call Developer method from API?

→ 1) **API**, they have to declare the method signature in their class & there need to call the method.

→ 2) **Developer:**He has to override the method & at runtime the developer method will be called.

→ API is not calling developer method, but at runtime Developer method will be called because we are overriding it.

```
A2 a = new B2();
a.x();
```

→ 1) Compile time it will check "**x()**" in **A2** & the code will be compiled.

→ 2) runtime it will check "**x()**" in the **B2** class & there is no "**x()**" in the **B2**, so it will verify in super class(**A2**) & it's there then **A2.x()** will be called.

→ 3) from **x()**, it's calling **y()**, so let's verify "**y()**" first in "**B2**" class & it's there so "**B2.y()**" is called.

→ 4) from "**y()**", it's calling "**z()**" so, let's verify "**z()**" first in "**B2**", & it's not there, so "**A2.z()**" will be called.

Polymorphism Implementaion: **ThreadTest** in the **JVM** level, they will create our Thread object(**StudentThread**) & it will be assigned to Runnable Reference so runtime it will call out Thread class run method.

StudentThread .java

```
/**@author:VenukumarS  @date:Oct 10, 2012
 * @fileName:StudentThread.java
 */
package edu.core.PolyTest2;

public class StudentThread extends Thread {
	public void run()
	{
		System.out.println(".StudentThread.run().");
	}
}
```

ThreadTest.java

```
/**@author:VenukumarS  @date:Oct 10, 2012
 * @fileName:ThreadTest.java
 */
package edu.core.PolyTest2;

public class ThreadTest {
	public static void main(String args[]) throws Exception {
		// Developer Thread.start(); start is a Native method & it's going to
		// call StudentThread class run() method JVM.
		// A a=new B();
Runnable runnable = (Runnable) Class.forName("edu.core.PolyTest2.StudentThread")
				.newInstance();

		// a.x();
		runnable.run();

		System.out.println(Thread.currentThread().toString());

	}
}

/*Output:
	java ThreadTest
	.StudentThread.run().
	Thread[main,5,main]
*/
```

PolyJDBCTest:

```
class.forName("oracle.jdbc.driver.oracleDriver");
```

class.forName can load a class & it can't do to the registration.
Registration is done: OracleDriver internal code

```
public class OracleDriver implements Driver {
        static {
                DriverManager.registerDriver();
        }
}
```

→ Once the class is loaded, it will execute static block then it's doing registration in the **DriverManager**.

Program:

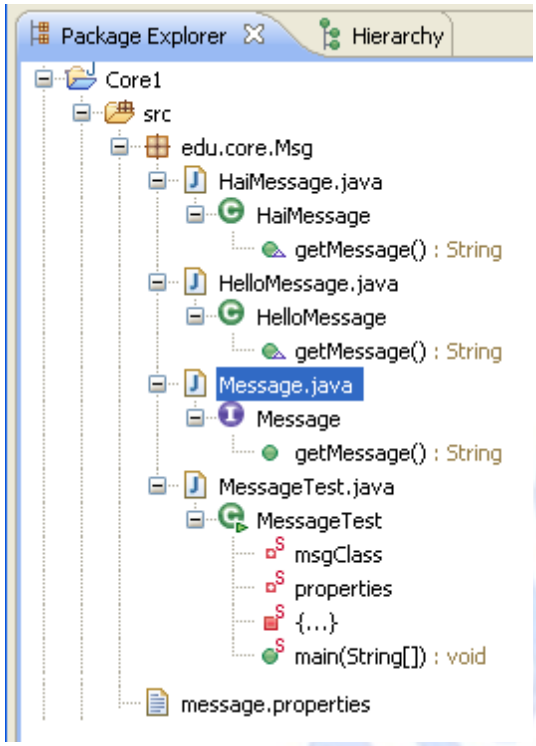
→ Put Hello & Hai message & resolve the code at runtime java code should not be effected.

1) Provide the **contract** [Message interfaces]
   In **Servlet** it is **Servlet** interface.
   This type of interface is given by Architect.
2) **Developers**, they have to implement the interface ..[Hello/Hi]
   In **Servlet** it's
   We need to provide the details using **xml**
   Here, we are providing using "**message.properties**"

3) **Artictect**, they will read **xml** & create the objects using **Reflection** & assign the object reference to contact..
   Here, it's alone using "**MessageTest**".
   In **Servlet** it's done by **Containers**.

Message.java

```
/**@author:VenukumarS  @date:Oct 10, 2012
 * @fileName:Message.java
 */
        package edu.core.Msg;

        public interface Message {
                public String getMessage();
        }
```



HaiMessage.java

```
/**@author:VenukumarS  @date:Oct 10, 2012
 * @fileName:HaiMessage.java
 */
package edu.core.Msg;

public class HaiMessage implements Message {
	public String getMessage() {
		return "Hai message";
	}
}
```

HelloMessage.java

```
/**@author:VenukumarS  @date:Oct 10, 2012
 * @fileName:HelloMessage.java
 */
	package edu.core.Msg;

public class HelloMessage implements Message {
	public String getMessage() {
		return "Hello message";
	}

}
```

MessageTest.java

```
/**@author:VenukumarS  @date:Oct 10, 2012
 * @fileName:MessageTest.java
 */

package edu.core.Msg;
public class MessageTest {
	private static Properties properties = new Properties();
	private static String msgClass;
	static {
		try {
			properties.load(MessageTest.class.getClassLoader()
					.getResourceAsStream("message.properties"));
			msgClass = properties.getProperty("message.class");
		} catch (IOException ie) {
			System.err.println(ie);
		}
	}

	public static void main(String args[]) throws Exception {
		Message msg = (Message) Class.forName(msgClass).newInstance();
		System.out.println(msg.getMessage());
	}
}

/*Output:
	java MessageTest
	Hai message
*/
```

message.properties

```
message.class=edu.core.Msg.HaiMessage
```

If message.class is HelloMessage then it'll execute HelloMessage.

```
message.class=edu.core.Msg.HelloMessage
```

****

→ Suppose, if we don't know what is the object we are passing & what is the return type, then the argument type & return type must be "Object".

In the API level, if the method argument type & return type is object means it never takes Object class object.., it is design to accept any object.

Ex:

```
public class ArrayList {
	public void add(Object object) {
	}

	public Object get() {
	}
}
```

Program:

```
/**@author:VenukumarS  @date:Oct 6, 2012
 * @fileName:WhyObject.java
 */
package edu.core;

public class WhyObject {
	public static Long WhyObjectMehtod1(Long sno) {
		return sno;
	}

	public static String WhyObjectMehtod2(String name) {
		return name;
	}

	public static Object WhyObjectMehtod(Object object) {
		return object;
	}

	public static void main(String args[]) {
		Long sno = WhyObjectMehtod1(new Long(1));
		String name = WhyObjectMehtod2("Guru");
		Object obj = WhyObjectMehtod("Raam");

		System.out.println("obj..=.className." + obj.getClass().getName());

		name = (String) obj;
		System.out.println("name..= "+name+" -> name Length()...=" + name.length());
	}
}

/*Output:
	java WhyObject
obj..=.className.java.lang.String
name..= Raam -> name Length()...=4
*/
```

Program:

```
/**@author:VenukumarS  @date:Oct 6, 2012
 * @fileName:WhyObject.java
 */
package edu.core;
public class WhyObject {
	public static Long WhyObjectMehtod1(Long sno) {
		return sno;
	}

	public static String WhyObjectMehtod2(String name) {
		return name;
	}
```

```
	public static Object WhyObjectMehtod(Object object) {
		return object;
	}

	public static void main(String args[]) {
		Long sno = WhyObjectMehtod1(new Long(1));
		String name = WhyObjectMehtod2("Guru");
		Object obj = WhyObjectMehtod("Raam");

		System.out.println("obj..=.className." + obj.getClass().getName());

		name = (String) obj;
		System.out.println("name..= "+name+" -> name Length()...=" + name.length());
		System.out.println("Obj Length()=" + obj.length());//CE:cannot find symbol length() in
class java.lang.Object
	}
}

/*
javac WhyObject.java
WhyObject.java:24: cannot find symbol
symbol  : method length()
location: class java.lang.Object
		System.out.println("Obj Length()=" + obj.length());
                                                       ^
*/
```

abstract class:

→ For any Java class if we don't want instantiation(creation of object) then we have to declare that class as abstract i.e. for abstract classes instantiation is not possible

```
abstract class A {

}
```

→ Abstract class may consist of abstract methods/general methods.

→ Abstract class is implemented by extends keyword.

```
class B extends A {
}
```

→ If we want to make methods/logics of one java class accessible only through subclasses then make that class as abstract class even though that class contains only concrete methods.

Ex: *HttpServlet* class is an *abstract class* containing no abstract methods

interface:

→ Any *Service Requirement Specification*(**SRS**) is considered as Interface.

→ From the client point of view an Interface defines the set of Services what is expecting.

→ From the Service provider point of view an Interface defines the set of services what is offering.

→ Hence an Interface considered as contract between client & service provider.

→ With in the Interface we can't write any implementation because it has to highlight just the set of services what we offering or what you are expecting. Hence, every method present inside interface should be abstract. Due to this, Interface is considered as pure abstract class.

Advantages:

1) We can achieve Security, because we are not highlighting out internal implementation.

2) Enhancement will become very easy, because without effecting outside person we can change our internal implementation.

3) Two different systems can communicate via Interface (A Java application can communicate with Mainframe System through Interface).

→ We can declare an Interface by using *interface* keyword, we can implement an Interface by using *implements* keyword.

→ If a class implements an Interface compulsory we should provide implementation for every method of that Interface. Otherwise we have to declare class as abstract. Violation leads to Compile-time error.

→ Whenever we are implementing an Interface method compulsory it should declared as public otherwise we will get Compile-time error.

Extends Vs Implements:

1) A class can extend only one class at a time.
2) A class can implement any number of Interfaces at a time.
3) A class can extend a class and can implement any number of Interfaces simultaneously.
4) A Interface can extend any number of Interfaces at a time.

Ex:

```
interface A {
}

interface B {
}

interface C extends A, B {
}
```

Ex:

```
interface A {
}

interface B {
}
```

```
class P {
}

class C extends P implements A, B {
}
```

Interface methods: Every interface method is by default public & abstract, whether we are specifying or not.

Ex:

```
interface A {
        void m1();
}
```

by default m1() has public & abstract.

public: To make this method (m1()) availability for every implementation class.
abstract: Because interface methods specifies requirements, but not implementation.

Interface variables:

→ n interface can contain variables. The main purpose of these variables is to specify constants at requirement level.

→ Every interface variable is always public, static & final, whether we are specifying or not.

Ex:

```
interface A {
        int x = 10;
}
```

variable x is public, static & final by default.

public: To make this variable available for every implementation class.
static: Without existing object also implementation class can access this variable.
final: Implementation class can access this variable but can't modify.

→ For the interface variable compulsory we should perform initialization at the time of declaration only, otherwise we will get compile-time error.

Marker Interface: It an interface with no methods and variables.

→ If an interface wont contain any method & by implementing that interface if our objects will get ability such type of interfaces are called Marker interface or Tag interface or ability interface.

→ **JVM is responsible to provide required ability in Marker interfaces**. because,to reduce complexity of the programing

Ex: Serializable, Clonable, RandomAccess, Single Thread model.
These interfaces are marked from some ability.

Ex: By implementing Serializable interface we can send object across the network and we can save state of object to a file. This extra ability is provided through Serializable interface.

By implementing clonable interface our object will be in a position to provide exactly dup-

licate object.

****

Adapter class:

→ **Adapter class** is a simple java class that implements an interface, an interface only with empty implementation.

Ex:

```
interface B {
        void m1();

        void m2();

        void m3();
}

abstract class AdapterB implements B {
        void m1() {
        }

        void m2() {
        }

        void m3() {
        }
}
```

→ If we implement an interface directly compulsory we should provide implementation for every method of that interface whether we are intrusted or not & whether it is required or not. This approach increases length of the code, so that readability will be reduced.

```
class Test implements B {
        // provide implementation for all method available in interface B
        void m1() {
        }

        void m2() {
        }

        void m3() {
        }
}
```

→ If we extends Adapter class instead of implementation interface directly then we have to provide implementation of only for required method, but not all. This approach reduces length of the code and improves readability.

```
        class Test extends AdpterB {
        void m1() {
        }
}
```

NOTE:

→ We don't know any thing about implementation just we have requirements specification, then we should go for **interface**. Ex: Servlet.

→ We are talking about implementation, but not completely(Just partial implementation) then we should go for **abstract class**. Ex. Generic-Servlet, HttpServlet.

→ We are talking about implementation completely & ready to provide service, then we should go for **concrete class**. Ex: our own Servlet.

→ When interface variable points to its implementation class object then it becomes object of implementation class and it is not the object of interface.

Ex:

```
interface B {
        void m1();
}

class D implements B {
        void m1() {
        //logic
        }

        void m2() {
        //logic
        }
}
class Test{
  public static void main(String[] args) {
        B obj = new D(); //obj is reference variable of interface "B" but not object.
        obj.m1(); //obj is object of implementation class "D".
  }
}
```

****

Differences between interface & abstract class

interface:

1) If we don't know any thing about implementation just we have requirement specification then we should go for interface.

2)Every method present inside interface is by default public & abstract.

3) The following modifiers are not allowed for interface methods:
protected, private, static, final, native, strictfp, synchronized

4) Every variable present inside interface is public, static, final by default.

5) for the interface variables we can't declare the following modifiers:
private, protected, transient, volatile

6) For the interface variables compulsory we should perform initialization at the time of declaration only.

7) Inside interface we can't take static block & instance blocks.

8) Inside interface we can't take constructor.

Abstract class:

1) If we are talking about implementation but not completely(partial implementation) then we should go for abstract class.

2) Every method present inside abstract class need not be public & abstract. We can take concrete methods also.

3) There is no restrictions for abstract class method modifier i.e., we can use any modifier.

4) Abstract variable need not be public, final, static.

5) There are no restriction for abstract class variable modifiers.

6) For the abstract class variables there is no restriction like performing initialization at the time of declaration.

7) Inside abstract class we can take static block & instance blocks.

9) Inside abstract class we can take constructor.

Inheritance:

```
/*
// Inheritance concept is applicable for static methods including main() also. Hence if the child class doesn't contain main() then parent class main() will be executed
class A
{
public static void main(String[] args)
{
System.out.println("PARENT");
}
}

class C extends A
{
}*/
/*
output:
PARENT
*/
```

```
	/**@author:Venu Kumar.S  @date:Oct 6, 2012
	 * @fileName:C.java
	 */
	class p {
		public static void main(String[] args) {
			System.out.println("PARENT");
		}
	}
	class C extends p // in this main() related it is not called as
```

```
                                        // "method Overriding", it is called as "method Hiding"
        {
                public static void main(String[] args) {
                        System.out.println("Child");
                }
        }

/*
Output:

java C

Child

java p
PARENT
*/


        /**@author:Venu Kumar.S  @date:Oct 6, 2012
         * @fileName:AbTest.java
         */
        class A {
                public void m1() {
                        System.out.println("A");
                }
        }

        class B extends A {
                private void m1() {
                        System.out.println("B");
                }

        }

        class AbTest {
                public static void main(String[] args) {
                        B b = new B();
                        b.m1();

                }
        }

/*

javac AbTest.java

AbTest.java:11: m1() in B cannot override m1() in A; attempting to assign weaker
 access privileges; was public
 private void m1()
              ^

*/
```

Overloading:

```
/*
In OverLoading method resolution always takes care by Compiler based on ref type and RunTime Object never
play any role in Overloading.
*/
	/**@author:Venu Kumar.S  @date:Oct 6, 2012
	 * @fileName:OverloadingTest.java
	 */
	class Animal {
	}

	class Monkey extends Animal {
	}

	class OverLoadingTest {
		public void m1(Animal a) {
			System.out.println("Animal Version");
		}

		public void m1(Monkey m) {
			System.out.println("Monkey Version");
		}

		public static void main(String args[]) {
			OverLoadingTest t = new OverLoadingTest();

			Animal a = new Animal();
			t.m1(a); // Animal Version

			Monkey m = new Monkey();
			t.m1(m); // Monkey Version

			Animal a1 = new Monkey();
			t.m1(a1);// Animal

		}
	}
/*
Output:

*/
/*
In OverLoading method resolution always takes care by Compiler based on ref type and RunTime Object never play any role in Overloading.
*/

/**@author::Venu Kumar.S  @date:Oct 6, 2012
 * @fileName:OverLoadingTest1.java
 */

	class OverLoadingTest1 {
		public void m1(int a, float b) {
```

```
                System.out.println("int-float Version");
        }

        public void m1(float a, int b) {
                System.out.println("float-int Version");
        }

        public static void main(String args[]) {
                OverLoadingTest1 t = new OverLoadingTest1();

                t.m1(10, 10.5f);// Ok
                t.m1(10.5f, 10);// Ok
                t.m1(10, 10); // CE: reference to m1 is ambiguous, both method
                                        // m1(int,float)
                t.m1(10.5f, 10.5f); // CE: cannot find symbol

        }
}

/*
Output:
javac OverLoadingTest1.java

OverLoadingTest1.java:26: reference to m1 is ambiguous, both method m1(int,float) in OverLoading-
Test1 and method m1(float,int) in OverLoadingTest1 match
t.m1(10,10); //CE:
 ^
OverLoadingTest1.java:27: cannot find symbol
symbol  : method m1(float,float)
location: class OverLoadingTest1
t.m1(10.5f,10.5f); //CE:
 ^
2 errors

*/
```

Overriding:

```
/*
Overriding:  What ever the parent has by default available to the child, if the child not satisfied with
parent class implementation then child is allowed to redefine its implementation in its own way--This
process is called "Overriding".

        The Parent class method which is overridden is called "Overridden method" &
        The child class method which is overriding is called "Overriding".

In overriding the method resolution always takes care by JVM, based on runtime obj &
In Overriding ref type never play any role.

*/
/**@author:Venu Kumar.S  @date:Oct 6, 2012
 * @fileName:OverRidingTest.java
 */
class P {
```

```
	public void m1() {
		System.out.println("Parent");
	}
}

class C extends P {
	public void m1() {
		System.out.println("Child");
	}
}

class OverRidingTest {
	public static void main(String args[]) {
		P p = new P();
		System.out.println("P p=new P()....->");
		p.m1(); // parent

		C c = new C();
		System.out.println("C c=new C()....->");
		c.m1(); // Child

		P p1 = new C();
		System.out.println("P p1=new C()....->");
		p1.m1(); // Child

		/*
		 * C c1= new P(); //CE: incompatible types
		 * System.out.println("C c1=new P()....->"); c1.m1();
		 */

		/*
		 * C c1=(C) new P(); //RE: java.lang.ClassCastException: P cannot be
		 * cast to C System.out.println("C c1=new P()....->"); c1.m1();
		 */
	}
}

/*
Output:
java OverRidingTest

P p=new P()....->
Parent
C c=new C()....->
Child
P p1=new C()....->
Child

*/
```

Singleton Class

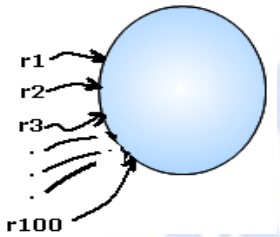
Singleton Class: For any java if we are allowed to create "only one object such type of class is "singleton class".

Ex: Runtime, ActionServlet(Struts1.x), BusinessDeligate(EJB), ServiceLocation(EJB).. etc.

→ The main advantage of singleton is, instead of creating a separate object for every requirement we can create a single object and reuse the same object for every requirement.

→ This approach improves memory utilization & performance of the system.

```
Runtime r1 = Runtime.getRuntime();
Runtime r2 = Runtime.getRuntime();

Runtime r3 = Runtime.getRuntime();
.
.
Runtime r100 = Runtime.getRuntime();
//Runtime is a class. getRuntime()is a static method
```



Creation of our own Singleton class:

→ We can create out own singleton classes also for this we have to user "private constructor & factory method".

Ex:

```
class Test{
        private static Test t;

        private Test() {
        }

        public static Test getInstance() {
                if (t == null) {
                        t = new Factory();
                }
                return t;
        }

        public Object clone() {
                return this;
        }
}

Test t1 = Test.getInstance();
Test t2 = Test.getInstance();
Test t3 = Test.getInstance();
Test t4 = Test.clone();
```

Factory method: By using class name if we call any method & return same class object, then that method is consider as Factory method.

```
Ex: Runtime r= Runtime.getRuntime(); //getRuntime() is factory method
    DateFormat df = DateFormat.getInstance();
    Test t = Test.getInstance();
```

Similarly, we can create Doubleton, Thribleton,....xxxton classes.

To Create doubleton class:

```
class Test {
        private static Test t1;
        private static Test t2;

        private Test() {
        }

        public static Test getInstance() {
                if (t1 == null) {
                        t1 = new Test();
                        return t1;
                } else if (t2 == null) {
                        t2 = new Test();
                        return t2;
                } else {
                        if (Math.random() < 0.5) {
                                return t1;
                        } else {
                                return t2;
                        }
                }
        }

}
```

Default constructor:

→ If we are not writing any constructor then compiler will always generate default constructor.

→ If we are writing at least one constructor then compiler won't generate default constructor.

Hence, a class can contain either programmer written constructor or compiler generated constructor but not both simultaneously.

Ex:

Programmer's code

```
1) class Test{
   }
```

```
2)public class Test {
  }
```

Compiler generated code

```
class Test {
        Test() {
                super();
        }
}
```

```
public class Test {
                public Test() {
                        super();
                }
}
```

```
3)  class Test {
        Test() {
        }
    }
```

```
class Test {
        Test() {
          super();
        }
}
```

```
4) class Test {
     Test() {
          this(10);
     }

     Test(int I) {
     }
}
```

```
class Test {
Test() {
          this(10);
        }

        Test(int I) {
          super();
        }
}
```

```
5) class Test {
        Test(int I) {
        }
}
```

```
class Test {
Test(int I) {
                super();
        }
}
```

```
6) class Test {
    void Test() {
    }
}
```

```
class Test {
        Test() {
                super();
        }
        void Test() {
        }
}
```

super() & this(): These are constructor calls. we should use only in constructors.
this(): Is used to call current class constructors.
super(): to call parent class constructors.

→ Compiler provides super() as default but not this().
→ The first line inside a constructor should be either super() or this().
→ If we are not writing any thing compiler will always places super().

1) We have to keep either **super()** or **this()** only as the first line of the constructor. otherwise "CE:call to super must be first statement in constructor".

```
class Test {
        Test() {
                System.out.println("Test");
                super();
        }
}
```

2) within the constructor we can use wither super() or this(), but not both simultaneously. Otherwise "CE:call to this must be first statement in constructor".

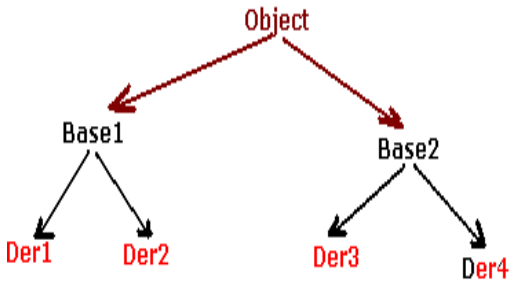
```
class Test {
        Test() {
```

```
                super();
                this();
            }
        }
```

super & this:

→ These are keywords to refers parent and current class instance members respectively.

→ We can use any where except in static area.




```
Base2 b = new Der4(); //OK

Object o = (Base2) b; //OK
Object o = (Base1) b; → CE:inconvertible types.
                                found:Base2
                                required: Base1

Base2 b1 = (Base2) o; //OK
Base1 b3 = (Der1)(new Der2()); → CE:inconvertible types.
                                found: Der2
                                required: Der1
```

→ Strictly in type-casting we are converting only type of object but not underlying object itself.

```
String s1 = new String("Guru");
Object o = (Object)s1;
System.out.println(s1==o); → true

/**@author:Venu Kumar S  @date:Oct 7, 2012
 * @fileName:TestTc.java
 */
class A {
        public void m1() {
                System.out.println("A");
        }
}

class B extends A {
        public void m1() {
                System.out.println("B");
        }
}

class C extends B {
        public void m1() {
                System.out.println("C");
        }
}

class TestTc {
        public static void main(String[] args) {
                C c = new C();
```

```
                c.m1(); // C

                ((B) c).m1();// C --> B b=new C(); b.m1();
                ((A) c).m1();// C --> A a=new C(); a.m1();
        }
}

/*
 * Output: C C C --Because, in type casting just we are converting only type of
 * object but not underlying object itself
 */
```

Ex: static methods:

```
/**@author:Venu Kumar S  @date:Oct 7, 2012
 * @fileName:TestTc1.java
 */
class A {
        public static void m1() {
                System.out.println("A");
        }
}

class B extends A {
        public static void m1() {
                System.out.println("B");
        }
}

class C extends B {
        public static void m1() {
                System.out.println("C");
        }
}

class TestTc1 {
        public static void main(String[] args) {
                C c = new C();
                c.m1(); // Output: C

                ((B) c).m1();//Output: B --> B b=new C(); b.m1();
                ((A) c).m1();//Output: A --> A a=new C(); a.m1();
        }
}

/*
Output:
C
B
A
*/
```

Ex: → Because, the overriding concept is not applicable for variables;

```
/**@author:Venu Kumar S  @date:Oct 7, 2012
 * @fileName:TestTc2.java
 */
class A {
	int x = 999;
}

class B extends A {
	int x = 888;
}

class C extends B {
	int x = 999;
}

class TestTc2 {
	public static void main(String[] args) {
		C c = new C();
		System.out.println("c.x..=" + c.x); //c.x..=999
		System.out.println("((B)c).x..=" + ((B) c).x); //((B)c).x..=888
System.out.println("((A)((B)c)).x..=" + ((A) ((B) c)).x);//((A)((B)c)).x..=999

	}
}

/*
Output:

c.x..=999
((B)c).x..=888
((A)((B)c)).x..=999

*/
```

From the above: the overriding concept is not applicable for variables.
if we declare all variables as static then there is no chance of change of the output.

Differece between == & equals():

→ "==" operator is always meant for **reference comparison.
where as "**equals()**" method meant for **content comparison**.

Ex:
```
String s1=new String ("guru");
String s2=new String ("guru");
String s3=s1;

System.out.println(s1==s2); → false, s1, s2 references are different.
System.out.println(s1.equals(s2)); → true, s1, s2 contents are same.
```

Reference equality: Two references, one object on the heap.

Two references that refer to the same object on the heap are equal. If we call the hashCode() method for both references, we'll get the same result(if we don't override the hashCode()).

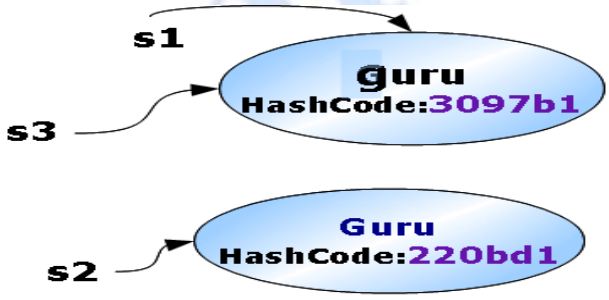
If we want to know if two references are really referring to the same object use the **==** operator, which compares the bits in the variables. If both references point to the same object, the bits will be identical.

Object equality: Two references, two objects on the heap, but the objects are considered meaningfully equivalent. For this we are using **equals().**

→ If we want to treat two different objects as equal, we must override both the hashCode() and equals() methods inherited from class Object.

→ If we don't override hashCode(), the default behaviour is to give each object a unique hash-

```
String s1 = new String ("guru");
String s3 = s1;

String s2 = new String ("Guru");
```



(s1 == s2); → false. Because s1, s2 references are different.
s1.equals(s2)) → false. s1, s2 contents are different

(s1 == s3); → true. Because s1, s3 references are same.
s1.equals(s3)) → true. s1, s3 contents are same

code value.

```
class RefObj {
        public static void main(String args[]) {
                String s1 = new String("guru");
                String s2 = new String("Guru");
                String s3 = s1;

         System.out.println("s1-->hashCode()...>"+Integer.toHexString(s1.hashCode()));
         System.out.println("s2-->hashCode()...>"+Integer.toHexString(s2.hashCode()));
         System.out.println("s3-->hashCode()...>"+Integer.toHexString(s3.hashCode()));

        System.out.println("s1==s2..>" + (s1==s2)); //false,s1,s2references are different.
        System.out.println("s1.equals(s2).." + s1.equals(s2)); //false, contents are diff
        System.out.println("s1==s3..>" + (s1==s3));//true, s1, s3 references are same.
        System.out.println("s1.equals(s3).." + s1.equals(s3)); //true, contents are same.
        }
}
/*Output:

s1-->hashCode() ...>3097b1
s2-->hashCode() ...>220bd1
s3-->hashCode() ...>3097b1
```

```
s1==s2..>false
s1.equals(s2)..false
s1==s3..>true
s1.equals(s3)..true
*/
```

== :

1)It is an operator applicable for both primitives & object references.
2)In the case of object references == operator is always meant for reference comparison i.e. if two references pointing to the same object then only == operator returns True.
3)We can't override == operator for content comparison.
4)In the case of heterogeneous type objects == operator causes CE, saying "in-compatible" types.
5)For any object reference r, r==null is always false.

equals():

1)It is a method applicable only for object references but not for primitives.
2)By default equals() method present in Object class, is also meant for reference comparison only.
3)We can override equals() method for content comparison.
4)In the case of heterogeneous objects equals() method simply return false & we won't get any CE or RE.
5)For any object reference r, r.equals(null) is always false.

→ In String classes equals() is overridden for content comparison.

→ In StringBuffer class equals() is not overridden for content comparison. Hence, Object class "equals()" got executed which is meant for reference comparison.

→ In Wrapper class "equals()" is overridden for content comparison.

Relation between == & equals()

→ If r1==r2 is true, then r1.equals(r2) is always true.

→ If r1==r2 is false, then we can't except about r1.equals(r2) exactly if may returns true or false.

→ If r1.equals(r2) returns true, we can't conclude about r1==r2, it may returns either true or false.

→ If r1.equals(r2) is false, then r1==r2 is always false.

clone(): The process of creating exactly duplicate objects is called cloning. The main objective of cloning is to maintain backup.

→ We can get cloned object by using clone() of objects class.

Syn: protected native Object clone() throws CloneNotSupportedException

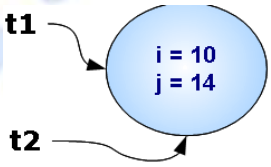
```
/**@author:Venu Kumar S  @date:Oct 7, 2012
 * @fileName:CloneTest.java
 */
class CloneTest implements Cloneable {
```

```
	int i = 10;
	int j = 14;

	public static void main(String args[]) throws CloneNotSupportedException {
		CloneTest t1 = new CloneTest();
		CloneTest t2 = (CloneTest)t1.clone();

		System.out.println(t1.i + "...." + t1.j);// 10....14
		System.out.println(t2.i + "...." + t2.j);// 10....14
		System.out.println(t1.hashCode() + "...." + t2.hashCode());// 4072869....1671711
		System.out.println("t1.hashCode()==t2.hashCode()::::..."
				+ (t1.hashCode() == t2.hashCode()));
//t1.hashCode()==t2.hashCode()::::...false
		System.out.println("t1==t2::::..." + (t1 == t2));//t1==t2::::...false
		System.out.println("t1.equals(t2)::::..." + (t1.equals(t2))); //t1.equals(t2)::::...false
		// changing t2 values
		t2.i = 90;
		t2.j = 80;
		System.out.println(t1.i + "...." + t1.j);// 10....14
		System.out.println(t2.i + "...." + t2.j);// 90....80

	}

}
```

Deep cloning & Shallow cloning:

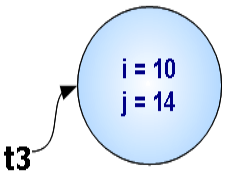
→ **Shallow cloning:** The process of creating just duplicate reference variable but not duplicate object is called "Shallow cloning".

```
CloneTest t1 = new CloneTest();
CloneTest t2 = t1; // Shallow cloning
```



→ **Deep cloning:** The process of creating exactly duplicate independent objects is considered as "Deep cloning".

```
CloneTest t3 = (CloneTest) t1.clone(); //Deep cloning
```




```
/**@author:Venu Kumar S  @date:Oct 7, 2012
 * @fileName:CloneTest1.java
 */
class CloneTest1 implements Cloneable {
	int i = 10;
	int j = 14;
```

```
        public static void main(String args[]) throws CloneNotSupportedException {
                CloneTest1 t1 = new CloneTest1();

                // Shallow cloning
                CloneTest1 t2 = t1;
                System.out.println("Shallow cloning......");
                System.out.println("t1.hashCode() " + t1.hashCode()
                                + "...t2.hashCode() " + t2.hashCode());//
t1.hashCode()4072869....t2.hashCode()4072869
                System.out.println("t1.hashCode()==t2.hashCode()::..."
                                + (t1.hashCode() == t2.hashCode())); //
t1.hashCode()==t2.hashCode()::...true
                System.out.println("t1==t2::..." + (t1 == t2));// t1==t2::...true
                System.out.println("t1.equals(t2)::..." + (t1.equals(t2))); // t1.equals(t2)::...true

                // Deep cloning
                CloneTest1 t3 = (CloneTest1) t1.clone();
                System.out.println("\n\nDeep cloning......");
                System.out.println("t1.hashCode() " + t1.hashCode()
                                + "...t3.hashCode() " + t3.hashCode());//
t1.hashCode()4072869....t3.hashCode() 1671711

                System.out.println("t1.hashCode()==t3.hashCode()::..."
                                + (t1.hashCode() == t3.hashCode())); //
t1.hashCode()==t3.hashCode()::...false

                System.out.println("t1==t3::..." + (t1 == t3));// t1==t3::...false

                System.out.println("t1.equals(t3)::..." + (t1.equals(t3)));// t1.equals(t3)::...false
        }
}

/*OUTPUT:
Shallow cloning......
t1.hashCode() 4072869...t2.hashCode() 4072869
t1.hashCode()==t2.hashCode()::...true
t1==t2::...true
t1.equals(t2)::...true


Deep cloning......
t1.hashCode() 4072869...t3.hashCode() 1671711
t1.hashCode()==t3.hashCode()::...false
t1==t3::...false
t1.equals(t3)::...false
*/
```

Do and find the output for the following programs:

```
class TestAbs {
	public static void main(String args[]) {
		Test1 t = new Test1();
		System.out.println(t.go().toString());
	}

}

abstract class Animal2 {
}

class Bear extends Animal2 {
}

class Test1 {
	public Animal2 go() {
		return new Bear();
	}
}

/*
Output:
	JAVATest.Bear@42e816
*/



public class Animal1 {
String name;

Animal1(String name) {
	this.name = name;
}

Animal1() {
	this(makeRandomName());
}

static String makeRandomName() {
	int x = (int) (Math.random() * 5);
String name = new String[] { "fluffy", "Fido", "Rover", "Spike", "Gigi" }[x];
	return name;
}

public static void main(String args[]) {
	Animal1 a = new Animal1();
	System.out.println(a.name);

	Animal1 b = new Animal1("Zeus");
	System.out.println(b.name);
}
}
```

```
interface Moveable1 {
	void moveIt();
}

interface Spherical1 {
	void doSphericalThing();
}

interface Bounceable1 extends Moveable1, Spherical1 {
	void bounce();

	void setBounceFactor(int bf);
}

class Ball1 implements Bounceable1 {
	public void bounce() {
		System.out.println("Ball.. Bounce");
	}

	public void setBounceFactor(int bf) {
		System.out.println("BounceFactor is .." + bf);
	}

	public void moveIt() {
		System.out.println("Ball... MoveIt..");
	}

	public void doSphericalThing() {
		System.out.println("Ball do sphericalThing");
	}
}

class BallTest {
	public static void main(String[] args) {
		Ball1 b = new Ball1();
		b.bounce();
		b.setBounceFactor(5);
		b.moveIt();
		b.doSphericalThing();
	}
}


//Interfaces

interface Moveable {
	void moveIt();
}

interface Spherical {
	void doSphericalThing();
```

```
	}

	interface Bounceable extends Moveable, Spherical {
		void bounce();

		void setBounceFactor(int bf);
	}

	abstract class Ball implements Bounceable {
		public Ball() {
			System.out.println("Ball.. is Abstract constructor");
		}

		public void bounce() {
			System.out.println("Ball.. Bounce");
		}

		public void setBounceFactor(int bf) {
			System.out.println("BounceFactor is .." + bf);
		}
	}

	class SoccerBall extends Ball {
		public void moveIt() {
			System.out.println("Ball... MoveIt..");
		}

		public void doSphericalThing() {
			System.out.println("Ball do sphericalThing");
		}
	}

	class BallTest1 {
		public static void main(String[] args) {
			SoccerBall sb = new SoccerBall();
			sb.bounce();
			sb.setBounceFactor(5);
			sb.moveIt();
			sb.doSphericalThing();
		}
	}

/*
OUTPUT:
	Ball.. is Abstract constructor
	Ball.. Bounce
	BounceFactor is ..5
	Ball... MoveIt..
	Ball do sphericalThing
*/
//TypeCasting
class Animal3 {
	void makeNoise() {
		System.out.println("Generic Noise");
```

```
	}
}

class Dog extends Animal3 {
	void makeNoise() {
		System.out.println("Bark..");
	}

	void playDead() {
		System.out.println("Roll Over..");
	}
}

class CastTest2 {
	public static void main(String[] args) {
		Animal3[] a = { new Animal3(), new Dog(), new Animal3() };
		for (Animal3 animal : a) {
			animal.makeNoise();
			if (animal instanceof Dog) {
				// animal.playDead(); //Shows cannot find symbol method
				// playDead() to overcome this typecast it to dog
				Dog d = (Dog) animal; // casting the ref. var.--Downcast
				d.playDead();

			}
		}
	}
}

/*
 * OUTPUT:

  Generic Noise
	Bark..
	Roll Over..
	Generic Noise

 */


public class EqualsTest {
	public static void main(String[] args) {
		Moof one = new Moof(8);
		Moof two = new Moof(8);
		if (one.equals(two)) {
			System.out.println("one and two are equal");
		}
	}
}

class Moof {
	private int moofValue;

	Moof(int val) {
```

```
		moofValue = val;
	}

	public int getMoofValue() {
		return moofValue;
	}

	public boolean equals(Object o) {
		if ((o instanceof Moof)
				&& (((Moof) o).getMoofValue() == this.moofValue)) {
			return true;
		} else {
			return false;
		}
	}
}

/*OutPut:
one and two are equal
*/


import java.util.*;

public class ScheduleDemo extends TimerTask {
	public static void main(String args[]) {
		System.out.println("Initial Time:::" + new Date().toString());
		Timer t1 = new Timer();
		t1.schedule(new Task1(), 2000);
		// t1.schedule(new Task1(),5000,3000);

		Calendar c1 = Calendar.getInstance();
		c1.set(2012, Calendar.DECEMBER, 21, 9, 18, 45);
		t1.schedule(new Task1(), c1.getTime());
	}

	/*
	 * (non-Javadoc)
	 *
	 * @see java.util.TimerTask#run()
	 */
	@Override
	public void run() {
		// TODO Auto-generated method stub

	}
}


/*
Output:
	Welcome to Timer
	--Fri Feb 15 12:41:57 IST 2013
```

```
*/


public class TestAnimals {
	public static void main(String[] args) {
		Animal4 a = new Animal4();
		Animal4 b = new Horse1(); // Animal ref, but a Horse object
		a.eat(); // Runs the Animal version of eat()
		b.eat(); // Runs the Horse version of eat()
		// b.buck();//Can't invoke buck(), because Animal class does't have that
		// method
	}
}

class Animal4 {
	public void eat() {
		System.out.println("Generic Animal Eating Generically..");
	}
}

class Horse1 extends Animal4 {
	private void eat() {
		System.out.println("Horse eating hay, oats, " + "and horse treats");
	}

	public void buck() {
		System.out.println("Horse Buck");
	}
}


public class TestAnimals1 {
	public static void main(String[] args) {
		Animal5 a = new Animal5();
		Horse2 h = new Horse2(); // Animal ref, but a Horse object
		// a.eat(); //Runs the Animal version of eat()
		h.eat(); // Runs the Horse version of eat()

	}
}
class Animal5 {
	private void eat() {
		System.out.println("Generic Animal Eating Generically..");
	}
}
class Horse2 extends Animal5 {
	public void eat() {
		System.out.println("Horse eating hay, oats, " + "and horse treats");
	}

	public void buck() {
		System.out.println("Horse Buck");
	}
```

```
}


class TestAnimals4 {
	public static void main(String[] args) {
		Horse4 h = new Horse4();
		Animal7 a1 = h; // Up-cast Ok with no explicit cast
		Animal7 a2 = (Animal7)h; // Up-cast Ok with explicit cast
		a1.eat();
		//a2.buck(); //error because buck is in child class
		((Horse4) a2).buck();
	}
}

class Animal7 {
	public void eat() {
		System.out.println("Generic Animal Eating Generically..");
	}
}

class Horse4 extends Animal7 {
	public void eat() {
		System.out.println("Horse eating hay, oats, " + "and horse treats");
	}

	public void buck() {
		System.out.println("Horse Buck");
	}
}


class TestInterface {
	public static void main(String args[]) {
		Test t = new Test();
		System.out.println(t.getChewable());
	}

}

interface Chewable {
}

class Gum implements Chewable {
}

class Test {
	public Chewable getChewable() {
		return new Gum();
	}
}

/*
Output:
```

```
	JAVATest.Gum@42e816
*/

//Overridden & Overloading

class Animal {
}

class Horse extends Animal {
}

class UseAnimals {
	public void doStuff(Animal a) {
		System.out.println("In the Animal Version");
	}

	public void doStuff(Horse h) {
		System.out.println("In the Horse Version");
	}

	public static void main(String[] args) {

		UseAnimals ua = new UseAnimals();

		Animal animalObj = new Animal();
		Horse horseObj = new Horse();

		ua.doStuff(animalObj);
		ua.doStuff(horseObj);

/*Hint : which overridden version of the method to call is decided at runtime based on object type, but
which overloaded version of the method to call is based on the reference type of the arg passed at
compile time */

		Animal animalRefToHorse = new Horse();
		ua.doStuff(animalRefToHorse);

/* Reg above two lines if you invoke a method passing it an Animal ref to a Horse object, the compiler
  knows only about the Animal, so it chooses the overloaded version of the method that takes an Ani-
mal
  It does not matter that at run-time there's actually a Horse being passes.*/
	}
}
/*
Output:
	In the Animal Version
	In the Horse Version
	In the Animal Version
*/


class DeadlockRisk {
```

```
	private static class Resource {
		public int value;
	}

	private Resource rA = new Resource();
	private Resource rB = new Resource();

	public int read() {
		synchronized (rA) {// may deadlock here
			synchronized (rB) {
				return rB.value + rA.value;
			}
		}
	}

	/*
	 * public void write(int a, int b) { synchronized(rA){// may deadlock here
	 * synchronized(rB){ rA.value = a; rB.value = b; } } }
	 */

	public void write(int a, int b) {
		synchronized (rB) {// may deadlock here
			synchronized (rA) {
				rA.value = a;
				rB.value = b;
			}
		}
	}

}

class DeadlockRiskTest {
	public static void main(String args[]) {
		DeadlockRisk d = new DeadlockRisk();
		d.write(5, 4);
		System.out.println(d.read());
	}
}
```

String:

→ String is immutable

→ Once we created a String object we can't perform any changes in the existing object. If we are trying to perform any changes, with those changes a new object will be created. This behaviour is nothing but "***immutability of String object***".

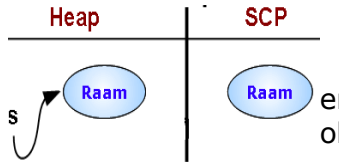
```
String s=new String("Raam");
s.concat("Kumar");
System.out.println(s); → Raam
```

→ In String class "equals()" is overridden for content comparison. Hence "equals()" returns

"true" if content is same even though objects are different.

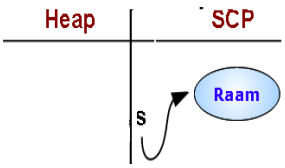
```
String s1=new String("Raam");
String s2=new String("Raam");
System.out.println(s1==s2); //false
System.out.println(s1.equals(s2)); //true
```

String s=new String("Raam");



→ In this case two objects will be created one is in **Heap**, & other is in **SCP**(**String Constant Pool**) and "**s**" is always pointing to heap object. Garbage collection not allowed in SCP area.

String s = "Raam"



→ In this case only one object will be created in SCP and s is always pointing to that object.

SCP:

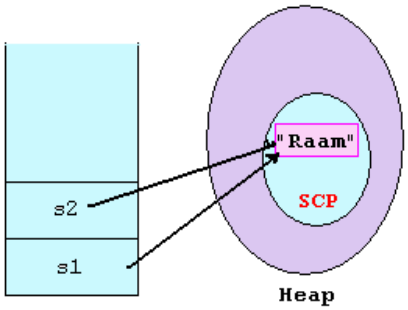
→ GC is not allowed to access in SCP area, hence even though object doesn't have any reference variable still it is not eligible for GC, if it is present in SCP area.

→ All objects present on SCP will be destroyed automatically at the time of JVM shutdown.

→ Object creation in SCP is always optional. First JVM will check is any object already present in SCP with required content or not, if it is already available then it will reuse existing object instead of creating new object. if it is not already available then only a new object will be created. Hence, there is no chance of two objects with the same content in SCP i.e., duplicate objects are not allowed in SCP.

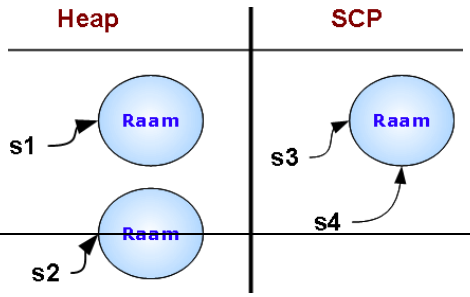
Ex1:

```
String s1 = new String("Raam");
String s2 = new String("Raam"); //no new object will be
                                  created
```



Ex2:

```
String s1 = new String("Raam"); // creates two objects
and one reference variable
String s2 = new String("Raam");
String s3 = "Raam";
String s4= "Raam";
```

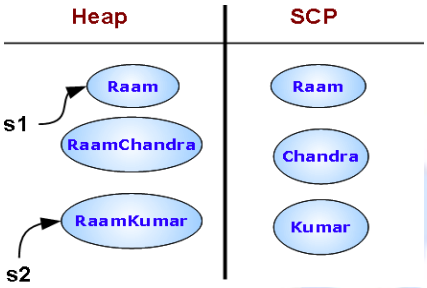


SCP contains only one object related to s1,s2,s3,s4 i.e. "Raam". s3, s4 points to this only.

s1, s2 points to heap.

Ex3:

```
String s1=new String("Raam");

s1.concat("Chandra");

String s2=s1.concat("Kumar");
```

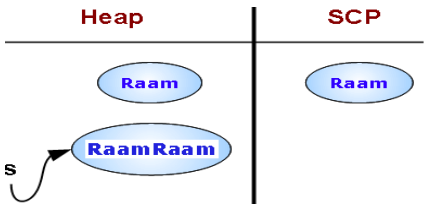



```
class StrTest {
	public static void main(String args[]) {
		String s1=new String("Raam");
		System.out.println("s1.." + s1);

		s1.concat("Chandra");
		System.out.println("s1..After s1.concat(\"Chandra\")..." + s1);

		String s2=s1.concat("Kumar");
		System.out.println("s1..After s1.concat(\"Kumar\")..." + s1);
		System.out.println("s2.." + s2);

	}
}
/*Output:
java StrTest
s1..Raam
s1..After s1.concat("Chandra")...Raam
s1..After s1.concat("Kumar")...Raam
s2..RaamKumar
*/
```

→ For every String constant compulsory ob object will be created in SCP area.

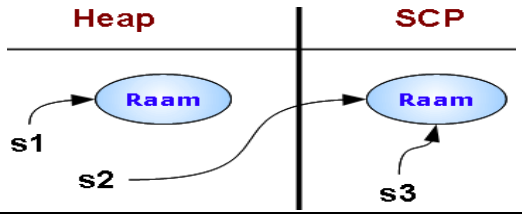
→ Because of some run-time operation if an object is required to created, that object should be created only on heap but not in SCP.

```
String s = "Raam"+new String("Raam");
```



Interning of String: By using heap object reference if we want to get corresponding SCP object reference then we should go for "**intern()**" method.

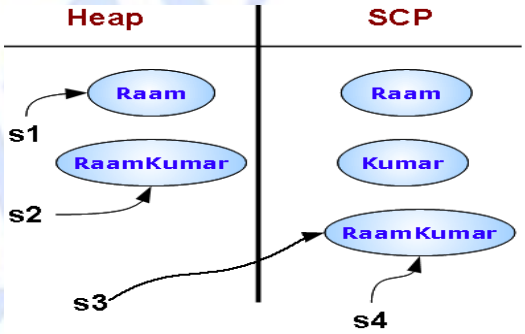
```
String s1=new String("Raam");
String s2=s1.intern();
System.out.println(s1==s2);//false
```



```
String s3="Raam";
System.out.println(s1==s3);//true

class StrTest {
	public static void main(String args[]) {
		String s1 = new String("Raam");
		String s2 = s1.intern();
		String s3 = "Raam";
		System.out.println("s1.." + s1 + "\ns2.." + s2 + "\ns3.." + s3);
		System.out.println("s1==s2..." + (s1 == s2));// false
		System.out.println("s2==s3..." + (s2 == s3)); // true
	}
}
/*Output:
java StrTest
s1..Raam
s2..Raam
s3..Raam
s1==s2...false
s2==s3...true
*/
```

→ If the corresponding object not available in SCP, then intern() creates that object and returns it.

```
String s1=new String("Raam");

String s2=s1.concat("Kumar");


String s3=s2.intern();

String s4="RaamKumar";

System.out.println(s3==s4); //true
```




```
class StrTest {
	public static void main(String args[]) {
		String s1=new String("Raam");
		String s2=s1.concat("Kumar");
		String s3=s2.intern();
		String s4="RaamKumar";
		System.out.println("s1.."+s1+"\ns2.."+s2+"\ns3.."+s3+"\ns4.."+s4);
		System.out.println("s3==s4..."+(s3==s4)); //true
	}
}
/*Output:
java StrTest
s1..Raam
s2..RaamKumar
s3..RaamKumar
s4..RaamKumar
s3==s4...true
```

```
*/
```

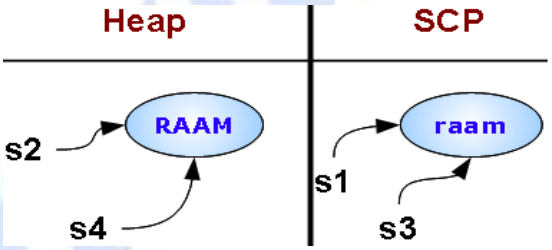
→ The main **advantage** of **SCP** is instead of creating a separate object for every requirement we can create only one object in SCP & we can reuse the same object for every requirement. So that performance and memory utilization will be increased.

→ The main **disadvantage** of **SCP** is we should compulsory maintain String objects as "**immutable**".

*** Once we created a String object we can't perform any changes in the existing object. If we are trying to perform any change with those changes a new object will be created on the Heap.

→ Because of our runtime method call if there is a change in content then only new object will be created.

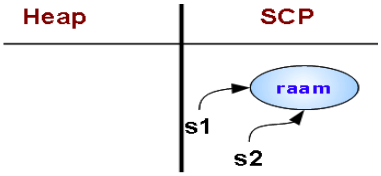
```
String s1 = "raam";
String s2 = s1.toUpperCase();
String s3 = s1.toLowerCase();
String s4 = s2.toUpperCase();
System.out.println(s1 == s2); //→ false
System.out.println(s1 == s3);//→ true
System.out.println(s2 == s4); //→ true
```




```
class StrTest {
	public static void main(String args[]) {
		String s1 = "raam";
		String s2 = s1.toUpperCase();
		String s3 = s1.toLowerCase();
		String s4 = s2.toUpperCase();
		System.out.println("s1 == s2.." + (s1 == s2)); // --> false
		System.out.println("s1 == s3.." + (s1 == s3));// --> true
		System.out.println("s2 == s4.." + (s2 == s4)); // --> true
	}
}
/*Output:
java StrTest
s1 == s2..false
s1 == s3..true
s2 == s4..true
*/
```

→ If there is no change in content existing object only will be reused.

```
String s1 = "Raam";
String s2 = s1.toString();
System.out.println("s1 == s2.." + (s1 == s2)); //→ true
```



```
class StrTest {
	public static void main(String args[]) {
		String s1 = "Raam";
		String s2 = s1.toString();
		System.out.println("s1 == s2.." + (s1 == s2)); // --> true
	}
}

/*Output:
java StrTest
s1 == s2..true
*/
```

StringBuffer:

→ If the content will change frequently then it is never recommended to go for String. Because for every change compulsory a new object will be created.
→ To handle this requirement compulsory we should go for StringBuffer where all changes will be performed in existing object only instead of creating new object.

→ StringBuffer sb=**new** StringBuffer();

→ creates an empty StringBuffer object with default initial **capacity 16**.
→ once StringBuffer reaches its max. capacity a new StringBuffer object will be created with **new capacity = (current Capacity+1)*2**

→ **StringBuffer** is **mutable**.
Once we created a StringBuffer object we can perform any changes in the existing object. This behaviour is nothing but "**mutability of StringBuffer object**".

```
StringBuffer sb = new StringBuffer("Raam");
s.append("Kumar");
System.out.println(sb); → RaamKumar
```

→ In StringBuffer class "equals()" method is not overridden for content comparison. Hence Object class "equals()" method will be executed which is meant for reference comparison due to this "equals()" method returns "false" even though content is same if objects are different.

```
StringBuffer sb1=new StringBuffer("Raam");
StringBuffer sb2=new StringBuffer("Raam");
System.out.println(sb1==sb2); //false
System.out.println(sb1.equals(sb2)); //false
```

→ In StringBuffer for every requirement a separate object is created. There is no chance of reusing the same StringBuffer object.

StringBuilder:

→ Every method present in StringBuffer is Synchronized, hence at a time only one thread is allowed to access StringBuffer object. It increases waiting time of the Threads & effects performance of the System.

→ To resolve this problem , StringBuilder is introduced in **1.5v**.

→ StringBuilder is exactly same as StringBuffer including methods & constructors

Difference between StringBuffer & StringBuilder:

StringBuffer:	StringBuilder:
1)Every method is synchronized.	1)No method is Synchronized.
2)StringBuffer object is Thread safe. Because SB object can be accessed by only one Thread at a time.	2)StringBuilder is not Thread safe. Because it can be accessed by multiple Threads simultaneously.
3)Relatively performance is low.	3)Relatively performance is high.
4)Introduced in **1.0v**.	4)Introduced in **1.5v**.

String Vz StringBuffer Vs StringBuilder:

→ If content will not change frequently then we should go for String.

→ If content will change frequently & Thread safety is required then we should go for StringBuffer.

→ If content will change frequently & Thread safety is not required then we should go for StringBuffer.

Immutable class Creation:

→ We can create our own immutable classes also.
→ once we created an object we can't perform any change in the existing object. If we are trying to perform any changes, with those changes a new object will be created.
→ Because of our run-time method call if there is no change in the content then existing object only will be returned.

Ex:

```
final class ImmuteCreation {
        private final int i;

        ImmuteCreation(int i) {
                this.i = i;
        }
```

```
        public ImmuteCreation modify(int i) {
                if (this.i == i) {
                        return this;
                }
                return (new Test(i));
        }
}

class ImmuteCreationTest {
        public static void main(Sting args[]) {
                ImmuteCreation t1 = new ImmuteCreation(10);
                ImmuteCreation t2 = new ImmuteCreation(100);
                ImmuteCreation t3 = new ImmuteCreation(10);

                System.out.println("t1==t2.." + (t1 == t2)); // false
                System.out.println("t1==t3.." + (t1 == t3)); // true

        }
}
```

Arrays

Array Representation :

→ In Java, arrays are *full-fledged objects*.

→ Like objects, arrays are always stored on the heap. Also like objects, implementation designers can decide how they want to represent arrays on the heap.

→ Arrays have a Class instance associated with their class, just like any other object. All arrays of the same dimension and type have the same class.

→ The length of an array (or the lengths of each dimension of a multidimensional array) does not play any role in establishing the array's class. For example, an array of three ints has the same class as an array of three hundred ints. The length of an array is considered part of its instance data.

→ The name of an array's class has one open square bracket for each dimension plus a letter or string representing the array's type.

Ex:

int []a; → The class name for an array of ints is "[I".

Object[][] o = new Object[3][]; → The class name for a two-dimensional array of Objects is "[[Ljava.lang.Object".

Byte[][][] a=new Byte[2][][]; → The class name for a three-dimensional array of bytes is "[[[B".

→ **Multi-dimensional arrays are represented as arrays of arrays.**

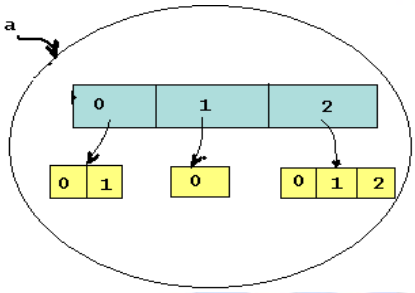
Array Declarations:

1D: `int[] a;`
`int a[];`
`int []a;`

2D: `int[][] a;`
`int [][]a;`
`int a[][];`

```
int[] a[];
int[] []a;
int []a[];
```

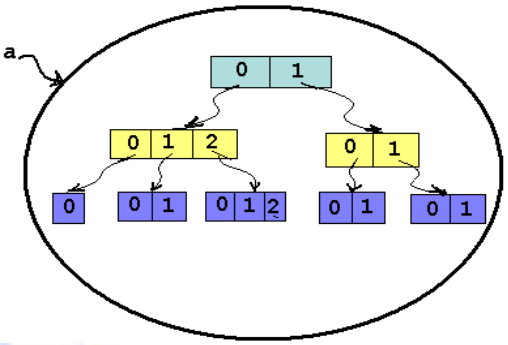
3D:
```
int[][][] a;
int a[][][];
int [][][]a;
int[] [][]a;
int[] a[][];
int[] []a[];
int[][] []a;
int[][] a[];
int [][]a[];
int []a[][];
```



Ex:2D Array:

```
int[][] a = new int[3][];
a[0] = new int[2];
a[1] = new int[1];
a[2] = new int[3];
```

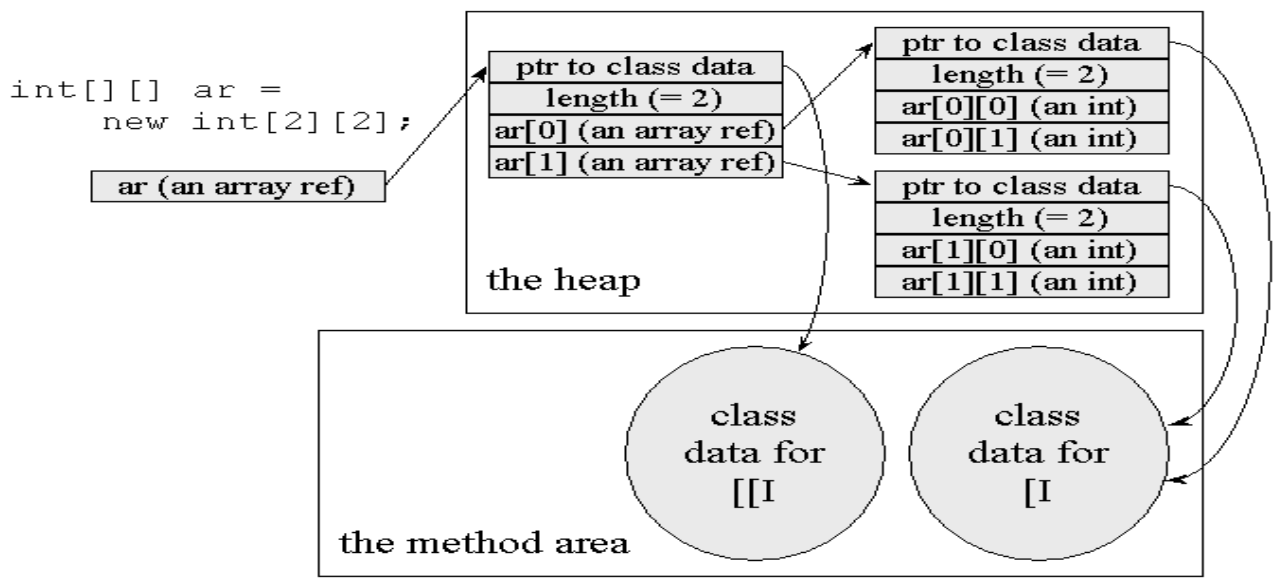
Ex: 3D Array:




```
int[][][] a = new int[2][][]
a[0] = new int[3][];
a[0][0] = new int[1];
a[0][1] = new int[2];
a[0][2] = new int[3];
a[1] = new int[2][2];
```

```
int[] a=new int[3];
System.out.println(a); →   [I@hashcode

int[][] a=new int[3][2];
System.out.println(a); →   [[I@hashcode
System.out.println(a[0]); →   [I@hashcode

int[][][] a=new int[2][][]
System.out.println(a); →   [[[I@hashcode
```

Ex: A two dimensional array of ints, would be represented by a one dimensional array of references to several one dimensional arrays of ints.




```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
 * @fileName:TestArray.java
  */

public class TestArray {
	public static void main(String[] args) {
		sum(new int[] { 10, 20, 30, 40 });
	}

	public static void sum(int[] x) {
		int total = 0;
		for (int x1 : x) {
			total += x1;
			System.out.println("sum...=" + total);
		}
	}
}
```

****

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
 * @fileName:TestArray1.java
 */

public class TestArray1 {
	public static void main(String[] args) {

		// in case of object type arrays as args, we can provide either declared
		// type or its child class objects

		Object[] o = new Object[5];
		o[0] = new Object();
```

```
		o[1] = new Integer(10);
		o[2] = new Float(11.556f);
		o[3] = new Double(3.1456987);
		o[4] = new String("vaagmichandra");

		// In case of abstract class type arrays as array elements we can
		// provide its child class Objects
		Number[] n = new Number[3];
		n[0] = new Integer(10);
		n[1] = new Float(11.556f);
		n[2] = new Double(3.1456987);

		System.out.println("Object Array Related values...");
		for (Object o1 : o) {
			System.out.println(o1);
		}

		System.out.println("Number Array Related values...");
		for (Number n1 : n) {
			System.out.println(n1);
		}
		// In case of Interface type array, as array ele we can provide it's
		// impl class Objects
		Runnable[] r = new Runnable[1];
		r[0] = new Thread();
		System.out.println("Runnable Array Related values..." + r[0]);
	}

}

/*
OutPut:

Object Array Related values...
java.lang.Object@19821f
10
11.556
3.1456987
vaagmichandra
Number Array Related values...
10
11.556
3.1456987
Runnable Array Related values...Thread[Thread-0,5,main]
*/


import java.util.*;

class SearchObjArray {
	public static void main(String args[]) {
		String[] sa = { "one", "two", "three", "four" };
```

```
		Arrays.sort(sa);
		for (String s : sa)
			System.out.print(s + " ");
		System.out.println("\n one=" + Arrays.binarySearch(sa, "one"));
		System.out.println("Now reverse sort");

		ReSortComparator rs = new ReSortComparator();
		Arrays.sort(sa, rs);
		for (String s : sa)
			System.out.print(s + " ");
		System.out.println("\nOne = " + Arrays.binarySearch(sa, "one"));
		System.out.println("\nOne = " + Arrays.binarySearch(sa, "one", rs));
	}

	static class ReSortComparator implements Comparator<String> {
		public int compare(String a, String b) {
			return b.compareTo(a);
		}
	}
}

/*
Output:

	four one three two
	 one=1
	Now reverse sort
	two three one four
	One = -1

	One = 2

*/
```

Anonymous Array: Sometimes we can create an array without name also such type of nameless arrays are called "**Anonymous Arrays**". The main objective of anonymous array is just for instant(only onetime) use(not future). We can create Anonymous Array as follows:

```
	new int[]{10,20,30,40};
```

```
class AnonymousTest {
	public static void main(String args[]) {
		sum(new int[] { 10, 20, 30, 40 });
	}

	public static void sum(int[] x) {
		int total = 0;
		for (int x1 : x) {
			total += x1;
		}
		System.out.println("The Sum:..." + total);// 100
	}
}
```

```
/*OUTPUT:
The Sum:...100
*/
```

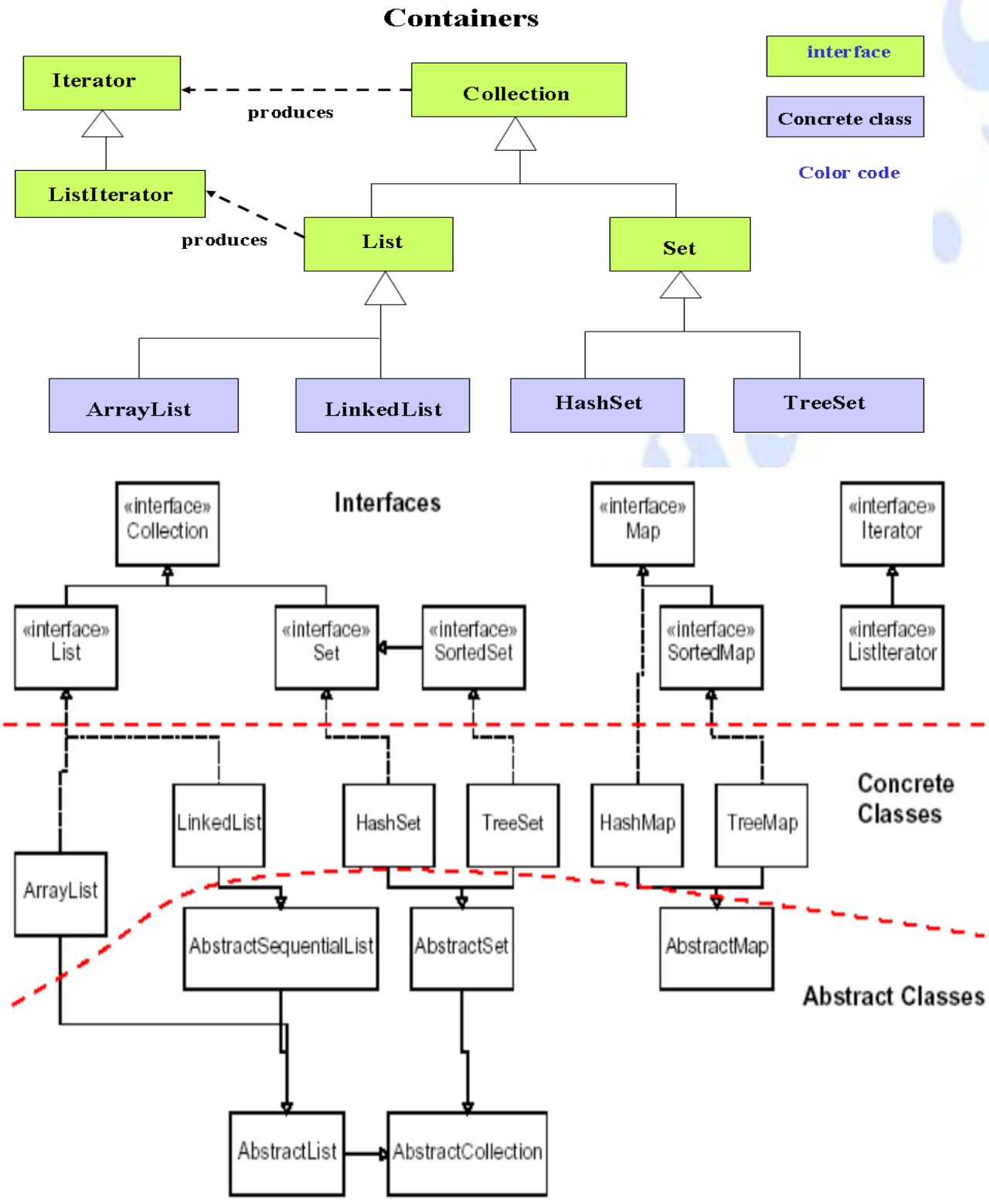
Vector

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
 * @fileName:VectorDemo.java
 */

import java.util.*;

class VectorDemo {
	public static void main(String[] args) {
		Vector v = new Vector();
		for (int i = 0; i <= 10; i++) {
			v.addElement(i);
		}
		v.addElement("GURU");
		System.out.println("Vector values....:" + v + "\nCapacity = "
				+ v.capacity() + "\tSize..=" + v.size());

		Enumeration e = v.elements();
		System.out.println("Vector values using Enumeration....:");
		while (e.hasMoreElements()) {
			System.out.println(e.nextElement());
		}

	}
}

/*
Output:
Vector values....:[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, GURU]
Capacity = 20   Size..=12
Vector values using Enumeration....:
0
1
2
3
4
5
6
7
8
9
10
GURU

Note:1)Enumeration concept is applicable only for legacy classes and hence is not a universal cursor
2) using this we can get only ReadAccess, we can't perform remove operation.

to overcome these limitations we are using the "Iterator".
```

```
*/
```

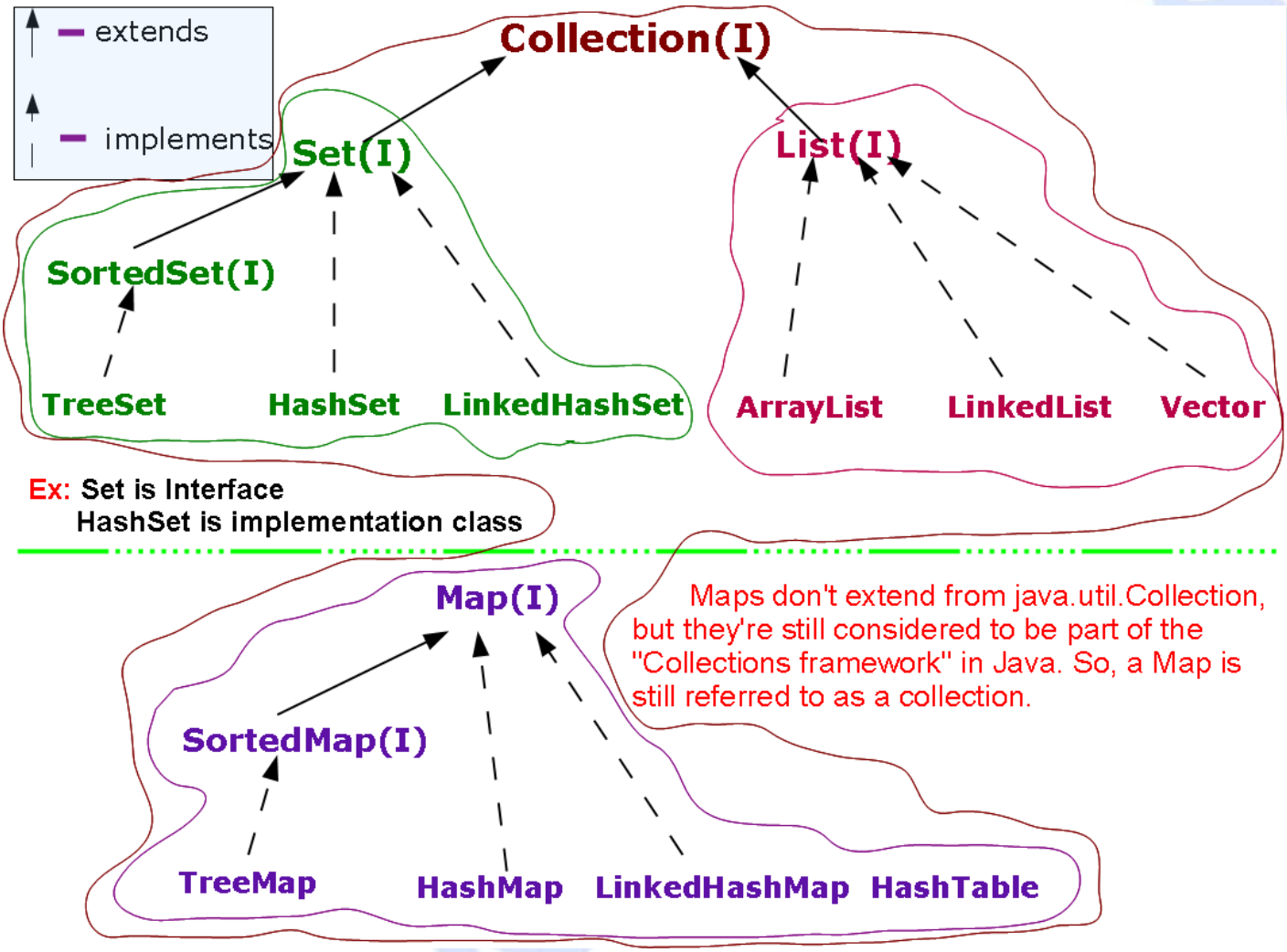
Stack: LIFO

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
 * @fileName:StackDemo.java
 */

import java.util.*;

class StackDemo {
	public static void main(String args[]) {
		Stack s = new Stack();

		System.out.println("stack empty()= "+s.empty());

		s.push("A");
		s.push("B");
		s.push("C");

		System.out.println("Stack Elements: " + s);
		System.out.println("stack empty()= "+s.empty());
		System.out.println("s.search(\"A\")= "+s.search("A"));
		System.out.println("s.search(\"Z\")= "+s.search("Z"));
		s.pop();
		System.out.println("Stack Elements ::: After pop:" + s);
		System.out.println("s.peek() = "+ s.peek());

	}
}

/*
 OUTPUT:

 javac StackDemo

stack empty()= true
Stack Elements: [A, B, C]
stack empty()= false
s.search("A")= 3
s.search("Z")= -1
Stack Elements ::: After pop:[A, B]
s.peek() = B
*/
```



Collection(I) v1.2 → **List(I) v1.2**
→ **Set(I) v1.2**
→ **Queue(I) v1.5**

Map(I) v1.2



List(I)v1.2 → ArrayList(c) v1.2
→ LinkedList(c) v1.2
→ VectorList(c) --> Stack(c) v1.0 -- Legacy classes

Set(I)v1.2 → HashSet(c) v1.2 → LinkedHashset(c) v1.4
→ SortedSet(I) v1.2 → NavigableSet(I) V1.6 → TreeSet(c) v1.2

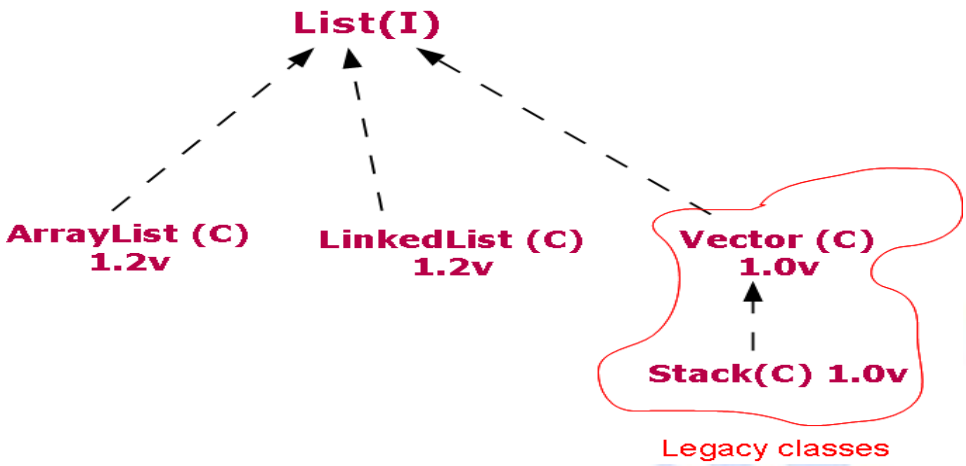
Queue(I) → PriorityQueue(c)
→ BlockingQueue(c)→ PriorityBlockingQueue
→ LinkedBlockingQueue

Map(I)v1.2 → HashMap(c) v1.2→ LinkedHashMap(c) v1.4
→ IdentityHashMap(c) v1.4
→ WeakHashMap(c) v1.2
→ SortedMap(I) v1.2 → NavigableSortedMap(I) v1.6 → TreeMap(c) v1.2
→ Dictionary(Abstract Class)v1.0 → HashTable(c)→ Properties(c) -- Legacy Classes

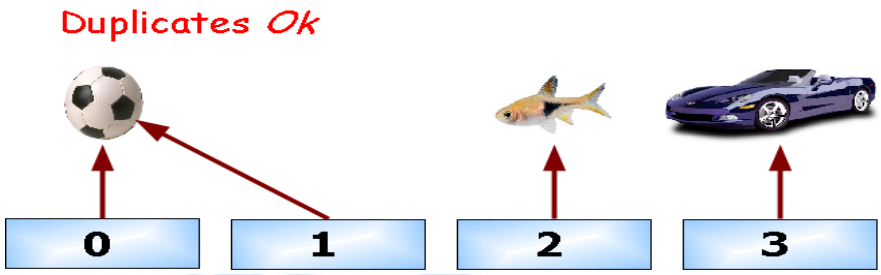
List(I)v1.2

List(I)v1.2 → ArrayList(c) v1.2

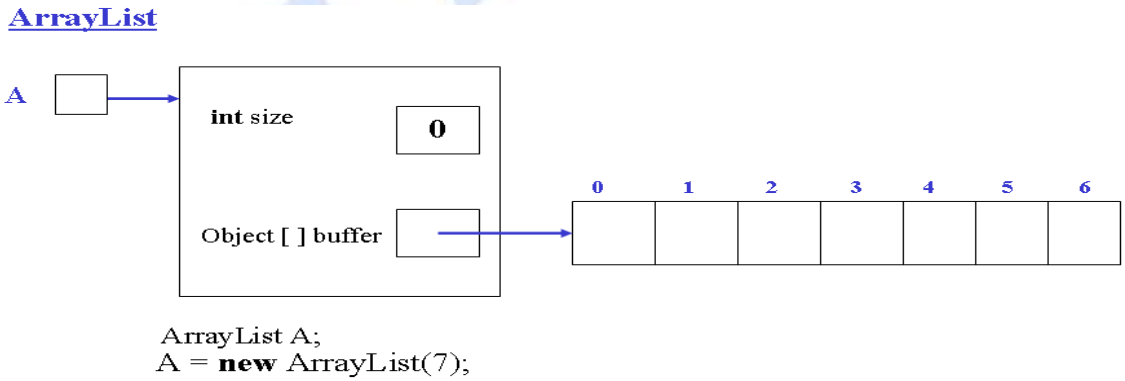
→ **LinkedList(c) v1.2**
→ **VectorList(c) --> Stack(c) v1.0 -- Legacy classes**



LIST → When sequence matters
→ **Collections that know about index position.**
→ **Lists know where something is in the list we can have more than one element referencing the same object.**



ArrayList




```
ArrayList A;
A = new ArrayList(7);
```

ArrayList:
→ It Resizable and Growable Array.
→ Insertion order is preserved.

→ Duplicate objects are allowed.

→ Heterogeneous objects are allowed.

→ ***null*** insertion is allowed.

→ *ArrayList* and *Vector* classes implements *RandomAccess Interface*, so that any random element we can access.

→ The collection efficiently (O(1)) inserts and deletes elements at the rear of the list. Operations at Intermediate positions have O(n) efficiency.

→ It automatically expands when an insertion is attempted in a full ArrayList.

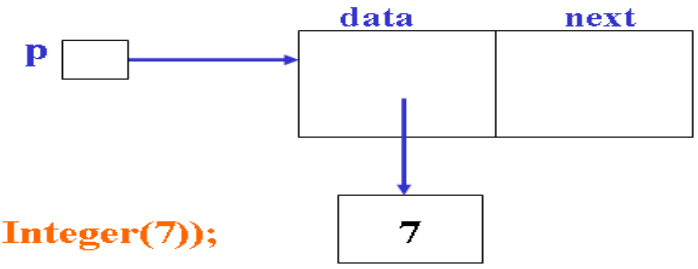
→ But! It holds only objects. Primitive types must be wrapped in their Wrapper class before being placed in an ArrayList, and objects removed or retrieved from an ArrayList must be down cast to retain the full functionality of members of their class.

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
 * @fileName:ArrayListDemo.java
 */
import java.util.*;

class ArrayListDemo {
	public static void main(String[] args) {
		ArrayList a = new ArrayList();
		a.add("AB");
		a.add(10);
		a.add('c');
		a.add(null);
		System.out
				.println("ArrayList values....:" + a + "\tSize = " + a.size());
		a.remove(2);
		System.out.println("After 2nd index removal::::ArrayList values....:"
				+ a + "\tSize = " + a.size());
		a.add(2, "V");
		a.add("G");
		System.out.println("Final ::::ArrayList values....:" + a + "\tSize = "
				+ a.size());
	}
}

/*
Output:
java ArrayListDemo

ArrayList values....:[AB, 10, c, null]  Size = 4
After 2nd index removal::::ArrayList values....:[AB, 10, null]  Size = 3
Final ::::ArrayList values....:[AB, 10, V, null, G]     Size = 5
*/
```

LinkedList

→ **LinkedList** underlying DS is Double Linked List.
→ Insertion order is preserved.
→ Heterogeneous and duplicate objects are allowed.
→ ***null*** insertion is possible.
→ Implements *Serializable* and *clonable* interfaces, but *not RandomAccess*.

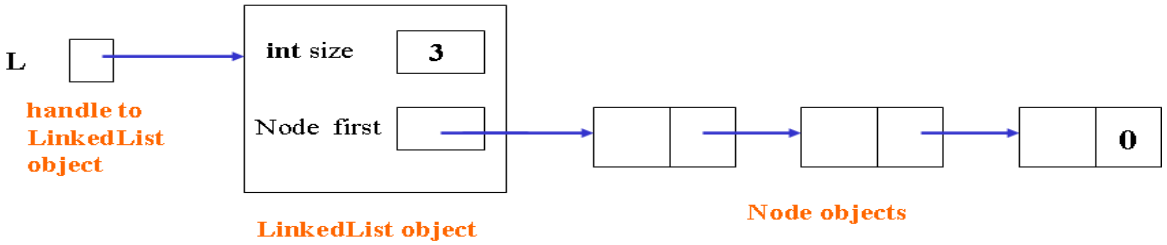
```
class Node {
    public Object data;
    public Node next;
}
```



Node p = new Node(new Integer(7));

LinkedList

These Node objects are linked together to form a structure for holding a sequence of objects.



- Positional access — manipulates elements based on their numerical position in the list
- Search — searches for a specified object in the list and returns its numerical position
- Iteration — extends Iterator semantics to take advantage of the list's sequential nature
- Range-view — performs arbitrary *range operations* on the list.

The List interface follows.

```
public interface List extends Collection { //E element
    // Positional access
    E get(int index); //Return the i-th element
    // optional
    E set(int index, E element); //Replace the i-th element with o
    // optional
    boolean add(E element); //Append E to the end
    // optional
    void add(int index, E element); //Insert Element at position i
    // optional
    E remove(int index); //Remove the i-th element
  // optional
boolean addAll(int index,  c);
```

```
        // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);

        // Iteration
    ListIterator listIterator();
    ListIterator listIterator(int index);

        // Range-view
    List subList(int from, int to);
}

/**@author:Venu Kumar.S  @date:Oct 6, 2012
 * @fileName:LinkedListDemo.java
 */
import java.util.*;

class LinkedListDemo {
    public static void main(String[] args) {
        LinkedList l = new LinkedList();
        l.add("Vagmee");
        l.add(9);
        l.add(null);
        System.out.println("LinkedList values....:" + l + "\tSize = "
                + l.size());
        l.set(0, "GURU");
        System.out.println("After l.set(0,\"GURU\"):::LinkedList values....:"
                + l + "\tSize = " + l.size());
        l.add(0, "CHANDRA");
        System.out.println("After l.add(0,\"CHANDRA\");::::LinkedList values....:"
                                + l + "\tSize = " + l.size());

        l.removeLast();

        System.out.println("After l.removeLast();::::LinkedList values....:"
                + l + "\tSize = " + l.size());

        l.addFirst("Sun");

        System.out.println("After l.addFirst(\"Sun\");::::LinkedList values....:"
                                + l + "\tSize = " + l.size());
    }
}

/*
Output:

java LinkedListDemo
LinkedList values....:[Vagmee, 9, null] Size = 3
After l.set(0,"GURU"):::LinkedList values....:[GURU, 9, null]  Size = 3
After l.add(0,"CHANDRA");::::LinkedList values....:[CHANDRA, GURU, 9, null] Size = 4
After l.removeLast();::::LinkedList values....:[CHANDRA, GURU, 9] Size = 3
After l.addFirst("Sun");::::LinkedList values....:[Sun, CHANDRA, GURU, 9] Size = 4
*/
```

Cursors

Cursors:

→ If we want to get objects one by one from the Collection we should go for cursor.
→ There are 3 types of cursors available in java.
→ 1) Enumeration(I) v1.0
→ 2) Iterator(I) v1.2
→ 3) ListIterator v1.2

1) Enumeration(I) v1.0:

→ It is a cursor to retrieve objects one by one from the Collection.
→ It is applicable for legacy classes.
→ We can create Enumeration obj by using elements()
public Enumeration elements();

ex: Enumeration e=v.elements();

methods → boolean **hasMoreElements()**;
Object **nextElement()**;

Limitations:

→ It is applicable only for legacy classes & hence it not a universal cursor.
→ By using Enumeration we can get only ReadAccess & we can't perform any remove operations.

****To overcome these limitations *Iterator* in introduced in *v1.2***

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
 * @fileName:EnumerationDemo.java
 */

import java.util.*;

public class EnumerationDemo {
	public static void main(String args[]) {
		Vector v = new Vector();
		for (int i = 0; i <= 10; i++) {
			v.addElement(i);
		}
		System.out.println(v);
		Enumeration e = v.elements();
		while (e.hasMoreElements()) {
			Integer i = (Integer) e.nextElement();
```

```
                if (i % 2 == 0) {
                        System.out.println(i);
                }
            }
            System.out.println(v);
        }
}
/*
OUTPUT:
java EnumerationDemo

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
0
2
4
6
8
10
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
 */
```

2)Iterator(I):

- → We can apply Iterator concept for any Collection object .
- → It is a universal cursor.
- → While iterating we can perform remove operation also, in addition to read operation.
- → We can get Iterator object by iterator() of collection interface.

```
Iterator itr = c.iterator();
```

- → Methods in Iterator interface :

```
boolean hasNext();
Object next();
void remove();
```

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
 * @fileName:IteratorDemo.java
 */
import java.util.*;

public class IteratorDemo {
        public static void main(String[] args) {

            HashMap hm = new HashMap();
            hm.put("chinmai", 700);
            hm.put("bal", 800);
            hm.put("venkat", 1000);
            hm.put("nagar", 500);

            System.out.println(hm);
```

```
		System.out.println("HashMap Keys...:");
		Set s = hm.keySet();
		System.out.println(s);

		System.out.println("Hashmap Values...:");
		Collection c = hm.values();
		System.out.println(c);

		Set s1 = hm.entrySet();

		Iterator its = s1.iterator();
		while (its.hasNext()) {
			Map.Entry m1 = (Map.Entry) its.next();
			System.out.println(m1.getKey() + "...." + m1.getValue());

			if (m1.getKey().equals("nagar")) {
				m1.setValue(10000);
			}

		}
		System.out.println(hm);
	}
}

	/*
	Output:

	java IteratorDemo
	{chinmai=700, bal=800, nagar=500, venkat=1000}
	HashMap Keys...:
	[chinmai, bal, nagar, venkat]
	Hashmap Values...:
	[700, 800, 500, 1000]

	chinmai....700
	bal....800
	nagar....500
	venkat....1000

	Final values
	{chinmai=700, bal=800, nagar=10000, venkat=1000}
	*/
```

3)ListIterator:

→ It is the Child Interface of Iterator.

→ While iterating objects we can move either to "**forward*(hasNext(), next(), nextIndex())***" or to the "**backward*(hasPrevious(), previous(), previousIndex())***".

→ It is bidirectional.

→ While iterating we can perform replacement and addition of new objects also in addition to read and remove operations.

other methods → **remove(), set(Object new), add(Object new)**

***Among the cursors(Enumeration, Iterator, ListIterator), the **ListIterator** is *powerful cursor*, *but* it is *applicable only* for *List* Objects.

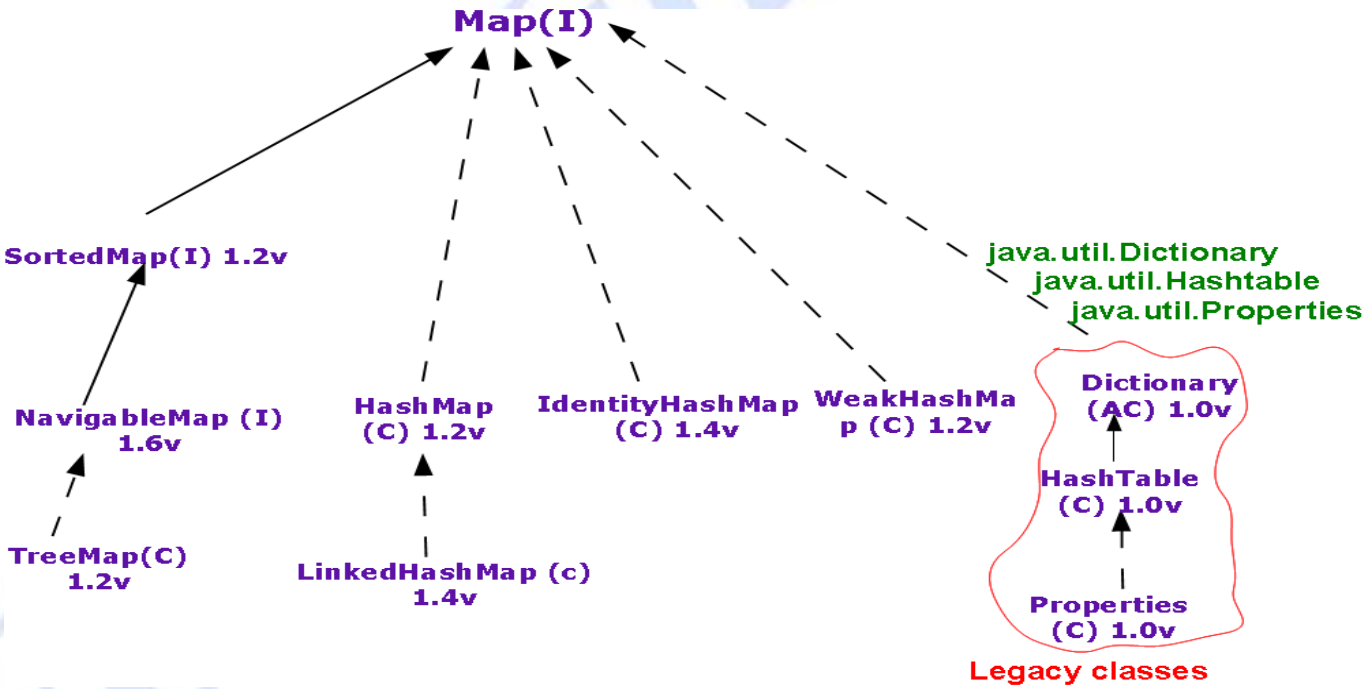
```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
 * @fileName:ListIteratorDemo.java
 */
import java.util.*;
class ListIteratorDemo {
	public static void main(String[] args) {
		LinkedList ll = new LinkedList();
		ll.add("VENKAT");
		ll.add("VENU");
		ll.add("PRAVEEN");
		ll.add("ARUN");

		System.out.println("LinkedList elements...:" + ll);

		ListIterator li = ll.listIterator();
		while (li.hasNext()) {
			String s = (String) li.next();
			if (s.equals("VENU")) {
				li.remove();
			}
			if (s.equals("PRAVEEN")) {
				li.set("ROHIN");
			}
			if (s.equals("ARUN")) {
				li.set("CHINMAI");
				li.set("GEETHIKA");
			}

		}
		System.out.println("LinkedList elements after Set in ListIterator...:"
				+ ll);
	}
}

/*
Output:

java ListIteratorDemo

LinkedList elements...:[VENKAT, VENU, PRAVEEN, ARUN]
LinkedList elements after Set in ListIterator...:[VENKAT, ROHIN, GEETHIKA]

*/
```

Cursors Comparison:

Property	Enumeration(1.0v)	Iterator(1.2v)	ListIterator(1.2v)
1) Is it legacy	Yes	No	No
2) It is applicable	Only for Legacy classes	For any Collection objects	For List objects
3) Movement	Single direction(only forward)	Single direction (only forward)	Bi-directional (backward & forward)
4) How to get it?	By using elements()	By using iterator()	By using listIterator()
5) Accessibility	only read	read & remove	read/remove/replace/add
6) Methods	hasMoreElements(), nextElement()	hasNext(), next(),remove()	9 methods

Map(I)

Map(I)v1.2 → HashMap(c) v1.2→ LinkedHashMap(c) v1.4
→ IdentityHashMap(c) v1.4
→ WeakHashMap(c) v1.2
→ SortedMap(I) v1.2 → NavigableSortedMap(I) v1.6 → TreeMap(c) v1.2
→ Dictionary(Abstract Class)v1.0 → HashTable(c)→ Properties(c) -- Legacy Classes

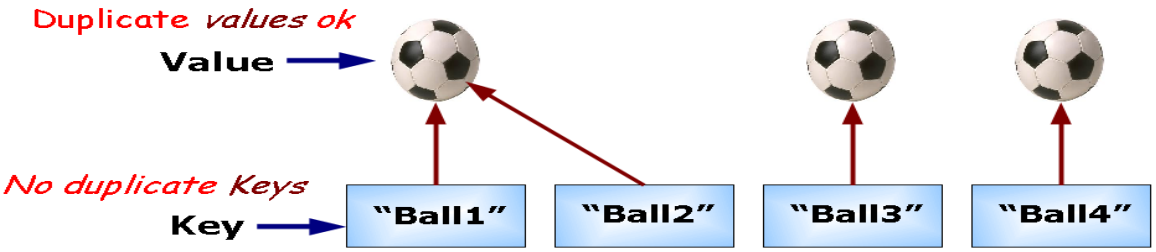


```
public interface Map {
  Object put(Object key, Object value)
  Object get(Object key);
  Object remove(Object key);
  boolean containsKey(Object key);
  boolean containsValue(Object value);
  int size();
  boolean isEmpty();

  void putAll(Map map);
  void clear();

  Set keySet();
  Collection values();
}
```

Basic ops

Bulk ops

Collection views



Each element In a Map Is actually TWO objects → a key and a value. We can have duplicate *values*, but NOT duplicate keys.

Map → When *finding something by* key matters.

→ Collections that use *key-value* pairs.

→ Maps know the value associated with a given key. We *can have two keys that reference the same value*, but we *cannot have duplicate keys*. Although keys are typically String names (so that we can make name/ value property lists, for example), a key can be any object.

→ An unordered collection that associates a collection of element values with a set of keys so that elements they can be found very quickly (O(1)!)

```
public interface Collection  {
 // Basic properties
  int size(); // The number of elements
  boolean isEmpty(); //Whether it is empty
  boolean contains(Object element);  //Membership checking.
  boolean containsAll(c); //Inclusion checking
  boolean equal(Object); // use equals() for comparison
  int hashCode();    // new equals() requires new hashCode()
```

```
// basic operations
  boolean add(Object);       // Optional; return true if this changed
  boolean addAll(c);                  //Add a collection
  boolean remove(Object);  // Optional; use equals() (not ==)
 boolean removeAll(c);  // remove a collection
 boolean retainAll(c);   //Keep the elements
  boolean clear();      //Remove all elements


public interface Iterator {
          boolean hasNext();     // cf:  hasMoreElements()
          Object next();          // cf:  nextElement()
          void remove();         // Optional
       }

        public interface Map {  // Map does not extend Collection
        // Basic Operations
          // put or replace, return replace object
                // NOTE: different from Weiss's insert(Hashable x)
                Object put(Object key, Object value);   // associates value with key optional
                Object get(Object key);  //The value associated with key
                Object remove(Object key); //Remove the mapping for key
                boolean containsKey(Object key); // Whether contains a mapping for key
                boolean containsValue(Object value); //Whether contains a mapping to value
                int size(); //The number of pairs
                boolean isEmpty(); //Whether it is empty


        // Bulk Operations
            void putAll(Map t);   //optional
             void clear();  // optional

        // Collection Views;
              //   backed by the Map, change on either will be reflected on the other.
                public Set keySet();    // cannot duplicate by definition!!
                public Collection values();   // can duplicate
                public Set entrySet();   // no equivalent in Dictionary

         // nested Interface for entrySet elements
                public interface Entry {
                   Object getKey();
                   Object getValue();
                   Object setValue(Object value);
        }
}
```

Map.EntryInterface for entrySet elements :

→ Each **key-value** pair is called on "**Entry**".

→ Without existing *Map object* there is no chance of *Entry object*. Hence, Interface *Entry* is define inside *Map Interfaces*.

```
interface Map {
        interface Entry {
                Object getKey();

                Object getValue();

                Object setValue();
        }
}
```

Ex:

```
Set s1 = hm.entrySet();
Iterator its = s1.iterator();
Map.Entry m1 = (Map.Entry) its.next();

        System.out.println(m1.getKey() + "...." + m1.getValue());

        if (m1.getKey().equals("nagar")) {
                m1.setValue(10000);
        }
```

HashMap

→ **N**o method is Synchronized.

→ Multiple Threads can operates simultaneously, hence HashMap object is not Thread Safe, because of this Threads are not required to wait and hence relatively performance is High

→ ***null*** is *allowed* for both *key & value.*

→ Introduced in 1.2v and it is non-legacy.

**

→ **HashMap is synchronized using "Map m=Collections.synchronizedMap(HashMap hm);"**

```
HashMap hm = new HashMap();
```

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
 * @fileName:HashMapDemo.java
 */

import java.util.*;

class HashMapDemo {
        public static void main(String[] args) {

                HashMap hm = new HashMap();

                hm.put("chinmai", 700);
                hm.put("bal", 800);
                hm.put("venkat", 1000);
                hm.put("nagar", 500);

                System.out.println(hm);
```

```
                System.out.println("HashMap Keys...:");

                Set s = hm.keySet();

                System.out.println(s);

                System.out.println("Hashmap Values...:");
                Collection c = hm.values();
                System.out.println(c);

                Set s1 = hm.entrySet();

                Iterator its = s1.iterator();
                while (its.hasNext()) {
                        Map.Entry m1 = (Map.Entry) its.next();
                        System.out.println(m1.getKey() + "...." + m1.getValue());

                        if (m1.getKey().equals("nagar")) {
                                m1.setValue(10000);
                        }

                }
                System.out.println(hm);
        }
}


/*
Output:

java HashMapDemo
{chinmai=700, bal=800, nagar=500, venkat=1000}
HashMap Keys...:
[chinmai, bal, nagar, venkat]
Hashmap Values...:
[700, 800, 500, 1000]

chinmai....700
bal....800
nagar....500
venkat....1000

Final values
{chinmai=700, bal=800, nagar=10000, venkat=1000}
*/
```

HashTable

→ Every method is Synchronized.

→ At a time only one Thread is allowed to operate on HASHTABLE Object, Hence it is Thread Safe. Because of this it increases waiting time of then Thread & hence relatively *performance is low*

→ ***null*** is *not allowed* for *both key & values*, otherwise we *will get NPE*.

→ Introduced in 1.0v and it is legacy.

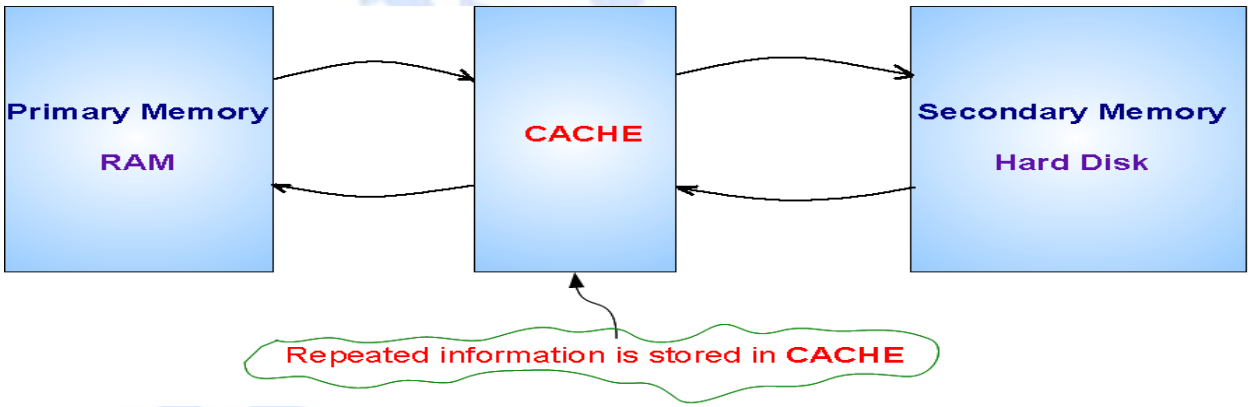
Differences between HashMap & HashTable

HashMap	HashTable
1) No Method is Synchronized.	1) Every method is Synchronized.
2) Multiple Threads can operates Simultaneously. Hence, HashMap object is not Thread safe.	2) At a time only one Thread is allowed to operate on HashTable. Hence, It is Thread safe.
3) Threads are not required to wait. Hence, relatively performance is high.	3) It increases waiting time of the Thread. Hence, relatively performance is low.
4) null is allowed for both key & value.	4) null is not allowed for both key & values. Otherwise we will get NPE.
5) Introduced in 1.2v. It is non-legacy class.	5) Introduced in 1.0v. It is legacy class

LinkedHashMap

→ The underlying DS is **HashTable + LL**.
→ Insertion Order is preserved.
→ Introduced in 1.4V

***NOTE:** The main application area of LinkedHashSet and LinkedHashMaps are **CACHE** applications implementation, where duplication is not allowed and insertion order is preserved.




```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
 * @fileName:LinkedHashMapDemo.java
 */

import java.util.*;

class LinkedHashMapDemo {
        public static void main(String[] args) {

                LinkedHashMap lhm = new LinkedHashMap();
```

```
		lhm.put("chinmai", 700);
		lhm.put("bal", 800);
		lhm.put("venkat", 1000);
		lhm.put("nagar", 500);

		System.out.println(lhm);

		System.out.println("LinkedHashMap Keys...:");
		Set s = lhm.keySet();
		System.out.println(s);

		System.out.println("LinkedHashMap Values...:");
		Collection c = lhm.values();
		System.out.println(c);

		Set s1 = lhm.entrySet();

		Iterator its = s1.iterator();
		while (its.hasNext()) {
			Map.Entry m1 = (Map.Entry) its.next();
			System.out.println(m1.getKey() + "...." + m1.getValue());

			if (m1.getKey().equals("nagar")) {
				m1.setValue(10000);
			}

		}
		System.out.println(lhm);
	}
}

/*
Output:
java LinkedHashMapDemo

{chinmai=700, bal=800, venkat=1000, nagar=500}
LinkedHashMap Keys...:
[chinmai, bal, venkat, nagar]
LinkedHashMap Values...:
[700, 800, 1000, 500]
chinmai....700
bal....800
venkat....1000
nagar....500
{chinmai=700, bal=800, venkat=1000, nagar=10000}
*/
```

Differences between HashMap & LinkedHashMap

HashMap	LinkedHashMap
1) The underlying DS is HashTable.	1) The underlying DS is HashTable + LinkedList.
2) Insertion order is not preserved	2) Insertion order is preserved.
3) Introduced in 1.2v.	3) Introduced in 1.4v.

IdentityHashMap

→ It is exactly same as HashMap, except the following:

1) In HashMap to identify duplicate keys JVM always uses ".equals()", which is mostly meant for content comparison.

2) If we want to use "==" operator instead of "equals()" to identify keys we have to use IdentityHashMap. "==" always meant for reference comparison.

General HashMap Related Program:

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
 * @fileName:HashMapDemo1.java
 */
import java.util.*;

class HashMapDemo1 {
        public static void main(String[] args) {

                HashMap hm = new HashMap();

                Integer i1 = new Integer(10);
                Integer i2 = new Integer(10);

                hm.put(i1, "Vaagmee");
                System.out.println(hm);

                hm.put(i2, "Rohin");

                System.out.println(hm);

        }
}

/*
Output:

java HashMapDemo1
{10=Rohin}
*/
```

The above program using IdentityHashMap:

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
 * @fileName:IdentityHashMapDemo.java
 */

import java.util.*;

class IdentityHashMapDemo {
	public static void main(String[] args) {

		IdentityHashMap ihm = new IdentityHashMap();

		Integer i1 = new Integer(10);
		Integer i2 = new Integer(10);

		ihm.put(i1, "Vaagmee");
		ihm.put(i2, "Rohin");

		System.out.println(ihm);

	}
}

/*
Output:

java IdentityHashMapDemo
{10=Vaagmee, 10=Rohin}
*/
```

WeakHashMap

→ **HashMap dominates GC**
→ **WeakHashMap is dominated by GC**

Program using HashMap:

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
 * @fileName:WeakHashMapDemo.java
 */

import java.util.*;

class HashMapDemoWeak {
	public static void main(String[] args) throws InterruptedException {
		HashMap m = new HashMap();
		Temp t = new Temp();
		m.put(t, "Vaagmee");

		System.out.println("\nBefore Garbagge Collection:\n" + m);

		t = null;
```

```
                System.gc();
                Thread.sleep(5000);

                System.out.println("\nAfter Garbagge Collection:\n" + m);

        }
}

class Temp {
        public String toString() {
                return "temp";
        }

        public void finalize() {
                System.out.println("Finalize Method Called");
        }

}

/*
OUTPUT:

java HashMapDemoWeak

Before Garbagge Collection:
{temp=Vaagmee}

//finalize() in Temp class is not called

After Garbagge Collection:
{temp=Vaagmee}
*/
```

REASON: HashMap dominates GC. HashMap object is not eligible for gc even-though it doesn't have any external references if it is associated with HashMap.

WeakHashMap is exactly same as HashMap; except that, in the case of WeakHashMap even-though object associated with WeakHashMap, it is eligible for GC, if it does not have any external references.

WeakHashMap is dominated by GC

WeakHashMap Program :

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
 * @fileName:WeakHashMapDemo1.java
 */
import java.util.*;

class WeakHashMapDemo1 {
        public static void main(String[] args) throws InterruptedException {
                WeakHashMap whm = new WeakHashMap();
                Temp1 t = new Temp1();
                whm.put(t, "Vaagmee");
```

```
                System.out.println("\nBefore Garbagge Collection:\n" + whm);

                t = null;
                System.gc();
                Thread.sleep(5000);

                System.out.println("\nAfter Garbagge Collection:\n" + whm);

        }
}

class Temp1 {
        public String toString() {
                return "temp";
        }

        public void finalize() {
                System.out.println("Finalize Method Called");
        }

}

/*
OUTPUT:

java WeakHashMapDemo1

Before Garbagge Collection:
{temp=Vaagmee}
Finalize Method Called  //finalize() in Temp1 is called

After Garbagge Collection:
{}
*/
```

REASON: WeakHashMap is dominated by GC

SortedMap(I)

→ If we want to represent a group of entries according to some sorting order then we should go for SortedMap.

→ The sorting should be done based on the keys but not based on the values.

→ SortedMap interface is the child interface of Map.

→ SortedMap interface defines the following 6 methods:

Object **lastKey()**, SortedMap **headMap(Object key)**, **tailMap(Object key)**, **subMap(Object key1, Object key2)**, Comparator **compatator()**

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
 * @fileName:SortedMapDemo.java
 */
import java.util.*;

class SortedMapDemo {
	public static void main(String[] args) {
		SortedMap sm = new TreeMap();

		sm.put("XXX", 10);
		sm.put("AAA", 20);
		sm.put("ZZZ", 30);
		sm.put("LLL", 40);

		System.out.println("\nSortedMap:\n" + sm);
	}
}

/*
OUTPUT:

java SortedMapDemo

SortedMap:
{AAA=20, LLL=40, XXX=10, ZZZ=30}

*/
```

SortedMapWithComparator:

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
 * @fileName:SortedMapWithComparatorDemo.java
 */

import java.util.*;

class SortedMapWithComparatorDemo {
	public static void main(String[] args) {
		SortedMap sm = new TreeMap(new MyComparator2());

		sm.put("XXX", 10);
		sm.put("AAA", 20);
		sm.put("ZZZ", 30);
		sm.put("LLL", 40);

		System.out.println("\nSortedMap With comparator:\n" + sm);
	}
}

class MyComparator2 implements Comparator {
	public int compare(Object obj1, Object obj2) {
		String s1 = obj1.toString();
```

```
                String s2 = obj2.toString();
                return s2.compareTo(s1);
        }
}

/*
OUTPUT:

java SortedMapWithComparatorDemo

SortedMap With comparator:
{ZZZ=30, XXX=10, LLL=40, AAA=20}

*/
```

NavigableMap(I)

→ It is the child interface of SortedMap to define several method for Navigation for the TreeMap Obj.

→ NavigableMap consist of the following methods:

1) **ceilingKey(e)**: returns the lowest element which is >= e
2) **higherKey(e)**:returns the highest element which is > e
3) **floorKey(e):** returns the highest element which is <= e
4) **lowerKey(e)**: returns the highest element which is < e
5) **pollFirstEntry()**: remove & returns first element.
6) **pollLastEntry()**: remove & returns last element.
7) **descendingMap()**: returns the NavigationSet in reverse order

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
 * @fileName:NavigableMapDemo.java
 */

import java.util.*;

class NavigableMapDemo {
        public static void main(String args[]) {
                TreeMap<String, String> t = new TreeMap<String, String>();
                t.put("b", "banana");
                t.put("c", "cat");
                t.put("a", "apple");
                t.put("d", "dog");
                t.put("g", "gun");
                System.out.println("Tree Map" + t);

                System.out.println("t.ceilingKey(\"c\")= " + t.ceilingKey("c"));
                System.out.println("t.higherKey(\"e\")= " + t.higherKey("e"));
                System.out.println("t.floorKey(\"e\")= " + t.floorKey("e"));
                System.out.println("t.lowerKey(\"e\")= " + t.lowerKey("e"));
                System.out.println("t.pollFirstEntry()= " + t.pollFirstEntry());
                System.out.println("t.pollLastEntry()= " + t.pollLastEntry());
                System.out.println("t.descendingMap()= " + t.descendingMap());
                System.out.println("Tree Map" + t);
        }
}
```

```
/*
 OUTPUT: javac NavigableMapDemo

 Tree Map{a=apple, b=banana, c=cat, d=dog, g=gun}
        t.ceilingKey("c")= c
        t.higherKey("e")= g
        t.floorKey("e")= d
        t.lowerKey("e")= d
        t.pollFirstEntry()= a=apple
        t.pollLastEntry()= g=gun
        t.descendingMap()= {d=dog, c=cat, b=banana}
 Tree Map{b=banana, c=cat, d=dog}
 */
```

TreeMap(c)

→ The underlay DS is RED-BLACK Tree.

→ Insertion order is not preserved and all entries are inserted according to some sorting order of keys.

→ If we are depending on default natural sorting order then the keys should be Homogeneous and Comparable, otherwise we will get ClassCastException(CCE).

→ If we defining our own sorting order by Comparator then the keys need not be Homogeneous & Comparable.

→ There is no restrictions on values, they can be Heterogeneous & non-comparable.

→ **null** acceptance:

for the "*empty TreeMap*" as the first entry with null key is allowed, after this if we are trying to insert any null we will get NPE

for the "*non-empty TreeMap*" if we are trying to insert any null we will get NPE.

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
 * @fileName:TreeMapDemo.java
 */
import java.util.*;

class TreeMapDemo {
        public static void main(String[] args) {
                TreeMap tm = new TreeMap();

                tm.put(100, "ZZZ");
                tm.put(103, "YYY");
                tm.put(101, "XXX");
                tm.put(104, 106);
                tm.put(107, null);

                // tm.put("FF","XXX"); //CCE
                // tm.put(null,"XXX"); //NPE

                System.out.println("\nTreeMap:\n" + tm);

        }
```

```
}
/*
OUTPUT:
java TreeMapDemo

TreeMap:
{100=ZZZ, 101=XXX, 103=YYY, 104=106, 107=null}

*/
```

TreeMapWithComparator

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
 * @fileName:TreeMapWithComparatorDemo.java
 */

import java.util.*;

class TreeMapWithComparatorDemo {
	public static void main(String[] args) {
		TreeMap tm = new TreeMap(new MyComparator1());

		tm.put("XXX", 10);
		tm.put("AAA", 20);
		tm.put("ZZZ", 30);
		tm.put("LLL", 40);

		System.out.println("\nTreeMap With comparator:\n" + tm);
	}
}

class MyComparator1 implements Comparator {
	public int compare(Object obj1, Object obj2) {
		String s1 = obj1.toString();
		String s2 = obj2.toString();
		return s2.compareTo(s1);
	}
}


/*
OUTPUT:

java TreeMapWithComparatorDemo

TreeMap With comparator:
{ZZZ=30, XXX=10, LLL=40, AAA=20}

*/
```

HashTable(c)

→ The underlay DS is HashTable.
→ Heterogeneous objects are allowed for both keys and values.
→ Insertion order is not preserved and it is based on HashCode of keys

→ **null** is *not allowed for both key and values* otherwise we will get *NPE*.

→ *Duplicate keys* are *not allowed*, but *values can be duplicated*.

→ All methods are *Synchronized* and hence HashTable object is **ThreadSafe**.

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
 * @fileName:HashTableDemo.java
 */
import java.util.*;

class HashTableDemo {
	public static void main(String[] args) {
		Hashtable h = new Hashtable();

		h.put(new Temp1(5), "A");
		h.put(new Temp1(2), "B");
		h.put(new Temp1(6), "C");
		h.put(new Temp1(15), "D");
		h.put(new Temp1(23), "E");
		h.put(new Temp1(16), "F");

		System.out.println("\nHashTable:\n" + h);
	}
}

class Temp1 {
	int i;

	Temp1(int i) {
		this.i = i;
	}

	public int hashCode() {
		return i;
	}

	public String toString() {
		return i + " ";
	}
}

/*
OUTPUT:

java HashTableDemo

HashTable:
{6 =C, 16 =F, 5 =A, 15 =D, 2 =B, 23 =E}

*/
```

Properties(c)

→ It is the child class of HashTable. In our program if any thing which frequently(like db, user-name, pwd, url) never recommended to hard code the value in the java program. Because for every change, we have recompile, rebuild, redeploy the application and sometime even server restart also required. which creates a big business impact to the client.

→ We have to configure those variables inside properties files and we have to read those values from java code.

→ The main advantage of this approach is, if any change in the properties file just redeployment is enough which not a business impact to the client.

abc.properties file

```
#updated by venu
#Sat Oct 06 15:32:21 IST 2012
user=scott
venu=7777
pwd=tiger
```

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
 * @fileName:PropertiesDemo.java
 */
import java.util.*;
import java.io.*;

class PropertiesDemo {
	public static void main(String[] args) throws IOException {
		Properties p = new Properties();

		FileInputStream fis = new FileInputStream("abc.properties");
		p.load(fis);
		System.out.println("Properties-->abc.property -> consist:\n" + p);

		String s = p.getProperty("venu"); // to get property value
		System.out.println(s);

		p.setProperty("venu", "9999"); // to modify existing property value
		p.setProperty("Vaagmee", "Rohin"); // to set new property

		FileOutputStream fos = new FileOutputStream("abc.properties");
		p.store(fos, "updated by venu");
	System.out.println("After modifications==>Properties-->abc.property -> consist:\n"+ p);
	}
}
/*
OUTPUT:
java PropertiesDemo

Properties-->abc.property -> consist:
{user=scott, venu=7777, pwd=tiger}
7777

After modifications==>Properties-->abc.property -> consist:
{user=scott, venu=9999, Vaagmee=Rohin, pwd=tiger}
*/
```

abc.properties file

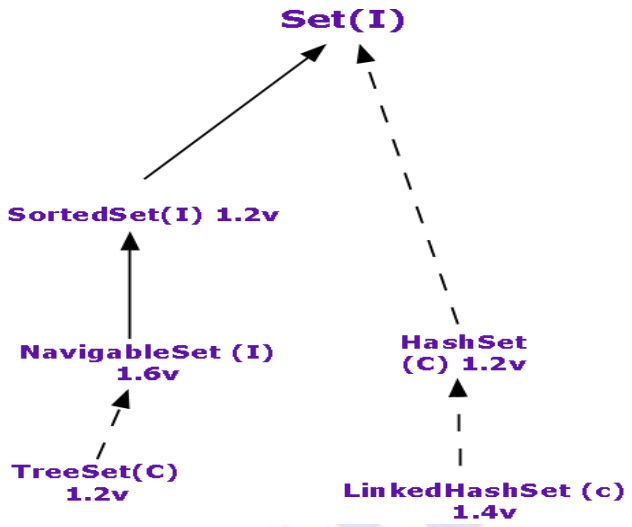
```
#updated by venu
#Sat Oct 06 15:32:21 IST 2012
user=scott
venu=9999
```

Vaagmee=Rohin
pwd=tiger

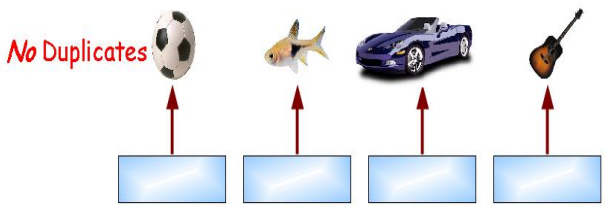
Set(I)v1.2

Collection(I) v1.2 → Set(I)v1.2

Set(I)v1.2 → HashSet(c) v1.2 → LinkedHashset(c) v1.4
→ SortedSet(I) v1.2 → NavigableSet(I) V1.6 → TreeSet(c) v1.2



SET → When uniqueness matters.



→ Collections that *do not allow duplicates*. Sets know whether something is already in the collection.

→ We can never have more than one element referencing the same object (or more than one element referencing two objects that are considered equal).

interface **java.util.Set** has the following methods:

```
public interface Set extends Collection  {
   // Basic operations
   int size();
   boolean isEmpty();
   boolean contains(Object element);
   // optional
   boolean add(E element);
   // optional
   boolean remove(Object element);
   Iterator iterator();

   // Bulk operations
   boolean containsAll(Collection c);
   // optional
   boolean addAll(Collection c);
   // optional
```

```
    boolean removeAll(Collection c);
    // optional
    boolean retainAll(Collection c);
    // optional
    void clear();

    // Array Operations
   Object[] toArray();
  Object[] toArray(Object[] a);
 }
```

Set is an interface;
There are four implementations:
HashSet is best for most purposes
TreeSet guarantees that an iterator will return elements in sorted order
LinkedHashSet guarantees that guarantees that an iterator will return elements in the order they were inserted
AbstractSet is a "helper" abstract class for new implementations
It's poor style to expose the implementation, so:
Good: Set s = new HashSet( ); Fair: HashSet s = new HashSet( );

HashSet

→ **HashSet u**nderlying DS is HashTable. duplicate objects are not allowed.
→ If we are trying to add duplicate objs we won't to get any CE or RE, hs.add() simply returns false.
→ Insertion order is not preserved, and all objects are inserted according to hashCode of the Objs.
→ Heterogeneous Objects are allowed.
→ **null** insertion is possible only once.
→ It impls "Serializable" and "clonable" Interfaces.

→ Every object has a reasonably-unique associated number called a *hash code*
public int hashCode() in class Object

→ HashSet stores its elements in an array a such that a given element o is stored at index o.hashCode() % array.length

→ Any element in the set must be placed in one exact index of the array
searching for this element later, we just have to check that one place to see if it's there (O(1))

HashSet *finds a matching hashcode for two objects*-- one we're inserting and *one already in the set – the HashSet will then call one of the object's equals() methods* to see if these hash code-matched objects really are equal.

And if they're equal, the HashSet knows that the object we're attempting to add is a duplicate of something in the Set, so the add doesn't happen.

We don't get an exception, but the HashSet's add() method returns a boolean to tell you whether the new object was added. So if the add() method returns false, we know the new object was a duplicate of something already in the set.

When testing whether a HashSet contains a given object:
Java computes the hashCode for the given object looks in that index of the HashSet's internal array.
Java compares the given object with the object in the HashSet's array using equals; if they are equal, returns true.

s1.containsAll(s2) → returns true if s2 is a subset of s1. (s2 is a subset of s1 if set s1 contains

all of the elements in s2)

s1.addAll(s2) → transforms s1 into the union of s1 and s2. (The union of two sets is the set containing all of the elements contained in either set)

s1.retainAll(s2) → transforms s1 into the intersection of s1 and s2. (The intersection of two sets is the set containing only the elements common to both sets)

s1.removeAll(s2) → transforms s1 into the (asymmetric) set difference of s1 and s2. (For example, the set difference of s1 minus s2 is the set containing all of the elements found in s1 but not in s2.)

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
 * @fileName:HashSetDemo.java
 */
import java.util.*;

class HashSetDemo {
	public static void main(String[] args) {
		HashSet hs = new HashSet();

		hs.add("B");
		hs.add("C");
		hs.add("D");
		hs.add("Z");
		hs.add(null);
		hs.add(10);

		System.out.println("HashSet elements...:" + hs);
		System.out.println("Adding duplicate value...::hs.add(\"Z\") = "
				+ hs.add("Z"));

	}
}

/*
Output:

HashSet elements...:[null, D, B, C, 10, Z]
Adding duplicate value...::hs.add("Z") = false

*/
```

LinkedHashSet

LinkedHashSet = HashSet + LinkedList.

→ Insertion order is preserved. It is in 1.4v, HashSet is in 1.2V

→ Application area of LinkedHashSet and LinkedHashMap is implementing **CACHE** applications, where duplicated are not allowed and insertion order must be preserved.

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
```

```
 * @fileName:LinkedHashSetDemo.java
 */

import java.util.*;
class LinkedHashSetDemo
{
public static void main(String[] args){
        LinkedHashSet lhs=new LinkedHashSet();
        lhs.add("B");
        lhs.add("C");
        lhs.add("D");
        lhs.add("Z");
        lhs.add(null);
        lhs.add(10);

        System.out.println("LinkedHashSet elements...:"+lhs);
        System.out.println("Adding duplicate value...::lhs.add(\"Z\") = "+lhs.add("Z"));

        }
}

/*
Output:

LinkedHashSet elements...:[B, C, D, Z, null, 10]
Adding duplicate value...::lhs.add("Z") = false

*/
```

SortedSet

SortedSet: → It is the child Interface of Set.

first(): returns first element of SortedSet.
last(): returns last element of SortedSet.
headSet(Obj): returns the SortedSet whose elements are < obj
tailSet(obj): returns the SortedSet whose elements are >= obj
subset(obj1, obj2): returns the SortedSet whose elements are >= obj1, but < obj2

comparator(): returns comparator obj described underlying sorting technique.
If we used default natural sorting order then we will get "null".

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
 * @fileName:SortedSetDemo.java
 */
import java.util.*;

class SortedSetDemo {
        public static void main(String[] args) {
                SortedSet ss = new TreeSet();
                ss.add(107);
                ss.add(104);
                ss.add(101);
                ss.add(102);
                ss.add(105);
```

```
		ss.add(103);
		ss.add(100);

		System.out.println("SortedSet elements...:" + ss);
		System.out.println("First element...:" + ss.first());
		System.out.println("Last element...:" + ss.last());
		System.out.println("headSet(104)...:" + ss.headSet(104));
		System.out.println("tailSet(104)...:" + ss.tailSet(104));
		System.out.println("subSet(101,107)...:" + ss.subSet(101, 107));
		System.out.println("comparator...:" + ss.comparator());

	}
}
/*
Output:

SortedSet elements...:[100, 101, 102, 103, 104, 105, 107]
First element...:100
Last element...:107
headSet(104)...:[100, 101, 102, 103]
tailSet(104)...:[104, 105, 107]
subSet(101,107)...:[101, 102, 103, 104, 105]
comparator...:null
*/
```

SortedSetWithIterator:

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
 * @fileName:SortedSetWithIteratorDemo.java
 */
import java.util.*;
class SortedSetWithIteratorDemo {
	public static void main(String[] args) {
		SortedSet ss = new TreeSet();
		ss.add(107);
		ss.add(104);
		ss.add(101);
		ss.add(102);
		ss.add(105);
		ss.add(103);
		ss.add(100);

		System.out.println("SortedSet elements...:");

		Iterator i = ss.iterator();

		/*
		 * 1) while(i.hasNext()) { System.out.println(i.next()); }
		 */

		/*
		 * 2) while(i.hasNext()) { Object element=i.next();
		 * System.out.println(element.toString()); }
		 */
```

```
		while (i.hasNext()) {
			// Object element=i.next();
			System.out.println(((Object) i.next()).toString());
		}

		System.out.println("First element...:" + ss.first());
		System.out.println("Last element...:" + ss.last());
		System.out.println("headSet(104)...:" + ss.headSet(104));
		System.out.println("tailSet(104)...:" + ss.tailSet(104));
		System.out.println("subSet(101,107)...:" + ss.subSet(101, 107));
		System.out.println("comparator...:" + ss.comparator());

	}
}
/*
Output:

SortedSet elements...:
100
101
102
103
104
105
107
First element...:100
Last element...:107
headSet(104)...:[100, 101, 102, 103]
tailSet(104)...:[104, 105, 107]
subSet(101,107)...:[101, 102, 103, 104, 105]
comparator...:null
*/
```

TreeSet

- → Underlying DS is Balanced Tree. duplicate objects are not allowed.
- → Insertion order is not preserved, because objects will be inserted according to some sorting order.
- → "*Heterogeneous objects are not allowed*. otherwise we will get "**ClassCastException**".
- → "**null**" insertion is *not possible*. otherwise we well get **RE→ NullPointerException**.

Comparable Interface: This Interface present in **java.lang** package and contains only one method "**compareTo()**".
When we are depending on default natural sorting order internally JVM calls "**compareTo()**"

comparator(): returns comparator obj described underlying sorting technique.
If we used default natural sorting order then we will get "null".

- → **Comparable** is meant for **default natural sorting order**.
- → **Comparator** is meant for **customized sorting order.**

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
 * @fileName:TreeSetDemo.java
```

```
 */

import java.util.*;

class TreeSetDemo {
	public static void main(String[] args) {
		TreeSet ts = new TreeSet();

		ts.add(107);
		ts.add(104);
		ts.add(101);
		ts.add(102);
		ts.add(105);
		ts.add(103);
		ts.add(100);

		System.out.println("TreeSet elements...:" + ts);
		System.out.println("\n\nTreeSet elements using Iterator...:");

		for (Iterator i = ts.iterator(); i.hasNext();) {
			System.out.println(i.next());
		}

	}
}

/*
Output:

TreeSet elements...:[100, 101, 102, 103, 104, 105, 107]

TreeSet elements using Iterator...:
100
101
102
103
104
105
107
*/
/**@author:Venu Kumar.S  @date:Oct 6, 2012
  * @fileName:TreeSetDemo1.java
 */
import java.util.*;

class TreeSetDemo1 {
	public static void main(String[] args) {
		TreeSet ts = new TreeSet();

		ts.add("Z");
		ts.add("K"); // "K".compareTo("Z");
		ts.add("A"); // "A".compareTo("K");
		ts.add("P"); // "P".compareTo("A");
		ts.add("G"); // "G".compareTo("P");
```

```
		ts.add("B"); // "B".compareTo("G");

		// ts.add(new Integer(10)); error: CCE-->ClassCastException
		// ts.add(null); RE: NPE

		System.out.println("TreeSet elements...:" + ts);
		System.out.println("\n\nTreeSet elements using Iterator...:");

		for (Iterator i = ts.iterator(); i.hasNext();) {
			System.out.println(i.next());
		}

	}
}

/*
Output:

TreeSet elements...:[A, B, G, K, P, Z]


TreeSet elements using Iterator...:
A
B
G
K
P
Z
*/

/**@author:Venu Kumar.S  @date:Oct 6, 2012
  * @fileName:TreeSetDemo2.java
 */
import java.util.*;
class TreeSetDemo2 {
	public static void main(String[] args) {
		TreeSet ts = new TreeSet();

		ts.add(107);
		ts.add(104);
		ts.add(101);
		ts.add(102);
		ts.add(105);
		ts.add(103);
		ts.add(100);

		System.out.println("TreeSet elements...:" + ts);
		System.out.println("\n\nTreeSet elements using Iterator...:");

		for (Iterator i = ts.iterator(); i.hasNext();) {
			System.out.println(i.next());
		}

	}
```

```
}
/*
Output:
TreeSet elements...:[100, 101, 102, 103, 104, 105, 107]
TreeSet elements using Iterator...:
100
101
102
103
104
105
107
*/
```

TreeSetUsingComparator:

Comparator interface: It is used to define our own sorting order instead of default sorting(**compareTo()**)

If we are *not passing* "**Comparator**" at Line 1, then JVM internally calls "**compareTo()**, which is meant for **default natural sorting order** → [0, 5, 10, 15, 20]

If we are *passing* "**Comparator**" object at Line 1 then our *own* "**compare()**" method will be executed which is meant for **customized sorting order** → [20, 15, 10, 5, 0]

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
  * @fileName:TreeSetUsingComparatorDemo.java
 */

import java.util.*;

class TreeSetUsingComparatorDemo {
        public static void main(String[] args) {
                TreeSet ts = new TreeSet(new MyComparator());

                ts.add(20);
                ts.add(0); // compare(0,10) → +ve

                ts.add(15); // compare(15,20) → +ve
                // compare(15,0) → -ve

                ts.add(5); // compare(5,20)→ +ve
                // compare(5,0) → +ve
                // compare(5,15) → -ve

                ts.add(10); // compare(10,20) → +ve
                // compare(10,0) → -ve
                // compare(10,15) → +ve
                // compare(10,5) → -ve
```

```
                System.out.println("TreeSet elements...:" + ts);
                System.out.println("\n\nTreeSet elements using Iterator...:");

                for (Iterator i = ts.iterator(); i.hasNext();) {
                        Integer x = (Integer) i.next();
                        System.out.println(x);
                }

        }
}

class MyComparator implements Comparator {
        public int compare(Object obj1, Object obj2) {
                Integer i1 = (Integer) obj1;
                Integer i2 = (Integer) obj2;

                if (i1 < i2)
                        return +1000;
                else if (i1 > i2)
                        return -1000;
                else
                        return 0;
        }
}


/*
Output:
TreeSet elements...:[20, 15, 10, 5, 0]


TreeSet elements using Iterator...:
20
15
10
5
0
*/
```

Comparable Vs Comparator :

→ 1) For predefined Comparable classes default natural sorting order is already available. If we are not satisfied with that we can define our own customized sorting by using Comparator. **Ex:String**

→ 2) For predefined non-Comparable classes default natural sorting order is not available compulsory we should define sorting by using comparator object only. **Ex: StringBuffer**

→ 3) For our own customized classes to define default natural sorting order we can go for comparable & to define customized sorting we should go for Comparator.

Ex: Employee, Student, Customer.

Comparison between Comparable & Comparator:

Comparable	Comparator
1) We can use Comparable to define default **natural** sorting order.	1) We can use Comparator to define **customized** sorting order
2) This interface present in **java.lang** package.	2) This interface present in **java.util** package.
3) Defines only only method i.e. **compareTo()**.	3) Defines two methods: i) **compare()** & ii) **equals()**
4) **All Wrapper classes & String class implements Comparable** interface.	4) **No predefined class implements** Comparator Interface.

NavigableSet(I)

→ It is the child interface of SortedSet to define several method for Navigation for the TreeSet Obj.

→ NavigableSet consist of the following methods:

1) **ceiling(e):** returns the lowest element which is >= e
2) **higher(e):**returns the highest element which is > e
3) **floor(e):** returns the highest element which is <= e
4) **lower(e):** returns the highest element which is < e
5) **pollFirst():** remove & returns first element.
6) **pollLast():** remove & returns last element.
7) **descendingSet():** returns the **NavigationSet** in reverse order

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
  * @fileName:NavigableSetDemo.java
 */

import java.util.*;

class NavigableSetDemo {
        public static void main(String args[]) {
                TreeSet<Integer> t=new TreeSet<Integer>();
                t.add(1000);
                t.add(2000);
                t.add(3000);
                t.add(4000);
                t.add(5000);

                System.out.println("Tree Map" + t);

                System.out.println("t.ceiling(2000)= " + t.ceiling(2000));
                System.out.println("t.higher(2000)= " + t.higher(2000));
                System.out.println("t.floor(3000)= " + t.floor(3000));
                System.out.println("t.lower(3000)= " + t.lower(3000));
                System.out.println("t.pollFirst()= " + t.pollFirst());
                System.out.println("t.pollLast()= " + t.pollLast());
                System.out.println("t.descendingSet()= " + t.descendingSet());
                System.out.println("Tree Map" + t);
        }
}
```

```
/*
 OUTPUT: javac NavigableSetDemo

  Tree Map[1000, 2000, 3000, 4000, 5000]
t.ceiling(2000)= 2000
t.higher(2000)= 3000
t.floor(3000)= 3000
t.lower(3000)= 2000
t.pollFirst()= 1000
t.pollLast()= 5000
t.descendingSet()= [4000, 3000, 2000]
Tree Map[2000, 3000, 4000]
*/
```

Set implemented classes comparison:

Property	HashSet	LinkedHashSet	TreeSet
1) Underlying DS	HashTable	HashTable+LinkedList	Balanced Tree
2) Insertion order	Not preserved	Preserved	Not Preserved
3) Sorting order	Not preserved	Not preserved	Not preserved
4) Heterogeneous Objects	Allowed	Allowed	Not Allowed
5) Duplicate objects	Not Allowed	Not Allowed	Not Allowed
6) null acceptance	Allowed one	Allowed one	for the empty TreeSet add the first element null insertion is possible. In all other cases we will get NPE

		Hash Set	**Tree Set**
Storage method:	→	hash table	red black tree
Space used:	→	O(n)	O(n)
Put speed:	→	O(1)	O(log n)
Iteration order:	→	Arbitrary	Sorted

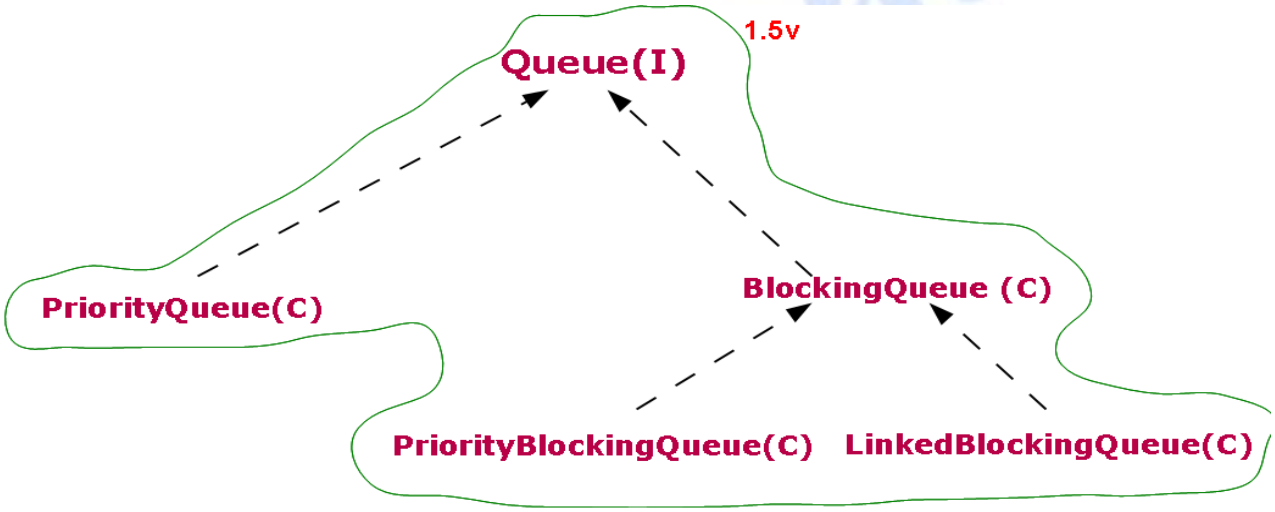
		Hash map	**Tree map**
Storage method:	→	hash table	red black tree
Space used:	→	O(n)	O(n)
Put speed:	→	O(1)	O(log n)
Iteration order:	→	**Arbitrary**	**Sorted**

concrete implements description

collection		
HashSet	Set	hash table
TreeSet	SortedSet	balanced binary tree
ArrayList	List	resizable-array
LinkedList	List	linked list
Vector	List	resizable-array
HashMap	Map	hash table
TreeMap	SortedMap	balanced binary tree
Hashtable	Map	hash table

Queue (v1.5)

Queue(I) → PriorityQueue(c)
→ BlockingQueue(c) → PriorityBlockingQueue



→ LinkedBlockingQueue

Queue(I): → It is the child Interface of collection. if we want to represent a group of individual objects prior to processing then we should go for Queue.

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
  * @fileName:PriorityQueueDemo.java
 */
import java.util.*;
import java.io.*;

class PriorityQueueDemo {
        public static void main(String[] args) throws IOException {
                PriorityQueue q = new PriorityQueue();

                System.out.println("q.peek() :" + q.peek()); // null
                // System.out.println(q.element()); //RE: NoSuchElementException

                for (int i = 0; i <= 10; i++) {
                        q.offer(i);
```

```
		}

		System.out.println("Priority Q: " + q);
	System.out.println("q.poll() :" + q.poll()); // del and returns the head element
		System.out.println("After Poll Priority Q: " + q);

	}
}
/*
OUTPUT:
java PriorityQueueDemo

q.peek() :null
Priority Q: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
q.poll() :0
After Poll Priority Q: [1, 3, 2, 7, 4, 5, 6, 10, 8, 9]

*/
```

PriorityQueue(C)

→ **PriorityQueue** is the DS to hold a group of individual objs prior to processing according to some priority.

→ The priority can be either default natural sorting order or customized sorting order.

→ If we are depending on default natural sorting compulsory objects should be *homogeneous & Comparable* otherwise we will get *classCastException*.

→ If we are defining our own customized sorting by *Comparator* then the objects need not be *Homogeneous and Comparable*.

→ Duplicate objs are not allowed.

→ Insertion order is not preserved.

→ **null** insertion is not possible even as first element also.

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
  * @fileName:PriorityQueueDemo1.java
 */

import java.util.*;
import java.io.*;

class PriorityQueueDemo1 {
	public static void main(String[] args) throws IOException {
		PriorityQueue q = new PriorityQueue(15, new MyComparator3());

		System.out.println("q.peek() :" + q.peek()); // null
		// System.out.println(q.element()); //RE NoSuchElementException

		q.offer("A");
		q.offer("Z");
		q.offer("L");
		q.offer("B");
```

```
                System.out.println("Priority Q: " + q);

        }
}

class MyComparator3 implements Comparator {
        public int compare(Object obj1, Object obj2) {
                String s1 = (String) obj1;
                String s2 = (String) obj2;
                return s2.compareTo(s1);

        }
}
/*
OUTPUT:
java PriorityQueueDemo1

q.peek() :null
Priority Q: [Z, B, L, A]
*/
```

Package

Package:

→ It is an Encapsulation mechanism to group related classes and interfaces into a single module. The main purposes of packages are:

1. To resolve naming conflicts.
2. To provide security to the classes & interfaces, so that outside person can't access directly.
3. It improves modularity of the application.

java package → **lang**, **io**, **net**, **util**, **awt**, **applet**, **sql**, **rmi**, **math**, **text**

javax package → **swing**, **sql**, **xml**, **naming**, **transaction**

→ There is one universally accepted convention to name packages i.e., to use internet domain name in reverse.

domain name in reverse.module name.submodule name.class name

Ex: **com.hdfcbank.loan.housingloan.Account**

```
package com.venu.kumar;

public class TestPackage {
        public static void main(String[] args) {

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

        }
}
```

1. javac TestPackage.java → Generated class file will be placed in current working directory.
2. javac -d . TestPackage.java

-d → destination to place generated class files

. → current working directory

→ Generated class file will be placed into corresponding package structure.

→ If specified package structure is not already available then this command itself will create that package structure.

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
  * @fileName:TestPackage.java
 */
package com.venu.kumar;

public class TestPackage {
	public static void main(String[] args) {

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

	}
}

/*
To compile:

javac -d . TestPackage.java → The generated class file is placed in corresponding package structure

-d → destination to place generated class file
. → current working directory

TestPackage.class is placed in com→ venu→ kumar→ TestPackage.class

To Execute:

java com.venu.kumar.TestPackage

output:
Package demo
*/
```

static imports:

This feature is introduced in java 5 to import static members of class

By using this feature we can access all →

→ non-private static members without using class name from other classes with in the package and

→ protected and public members from outside package class members without using class name.

syntax: import static packagename.classname.*;

→ It allows to call all static members of the class

or

import static packagename.classname.staticmembername;

→ allows only to call the imported static member.

Ex: **import p1.*;** → We can access all classes from p1 package

import p1.A; → We can access only class A from p1 package.

import static p1.A.*; → We can access all static members of class A from p1 package.

→ Using this import statement we cannot access non-static members, we can't create object, we can't develop subclass from A class. for this purpose we must also write import statement separately for accessing class A as import p1.A;

import static p1.A.a; → We can access only the static variable "**a**".If "**a**" is a non-static variable it leads to **CE:**cannot find symbol "**static a**"

import static p1.A.m1;→ We can access only static method "**m1**".

Ex1:Program

```
package p2;

import static java.lang.System.*;

public class Test {
	public static void main(String args[]) {
		out.println("Hi");
	}
}
/*Output:
 java Test
	Hi
*/
```

Ex2:Progam

```
package p1;

public class Example {
	public static int a = 10;
	public int x = 20;

	public static void m1() {
		System.out.println(".p1.Example.m1().");
	}

	public void m2() {
		System.out.println(".p1.Example.m2().");
	}

}

package p2;
```

```
import static p1.Example.*;

public class StaticSample {
	public static void main(String args[]) {
		// static int a=10;

		// accessing static members
		System.out.println("p1.Example.a...."+p1.Example.a);
		m1();

		// accessing static members with classname
		// System.out.println(Example.a);//CE:
		// Example.m1();//CE:

		// accessing non-static members
		// Example e=new Example();CE:
		// System.out.printlm(e.x);
		// e.m2();
	}
}

/*
Output:
	java StaticSample
	p1.Example.a....10
	.p1.Example.m1().
*/


package p2;

import p1.Example;

public class NonStaticSample {
	public static void main(String args[]) {
	// accessing static members with classname
		System.out.println("...Accessing static members....:");
		System.out.println("Example.a...."+Example.a);
		Example.m1();

		// accessing non-static members
		System.out.println("\n...Accessing non-static members....:");
		Example e=new Example();
		System.out.println("e.x...."+e.x);
		e.m2();
	}
}

/*
Output:
	java NonStaticSample
...Accessing static members....:
Example.a....10
```

```
        .p1.Example.m1().

        ...Accessing non-static members....:
        e.x....20
        .p1.Example.m2().
    */
```

VarArgs 1.5v

VarArgs 1.5v: Until 1.4v we can't declare a method with variable number of args, if there is any change in number of arguments compulsory we should declare a new method. This approach increases length of the code & reduces readability.

→ To resolve these problem, **var-arg** method in 1.5v. Hence, from **1.5v** onwards we can declare a method with variable number of args such type of methods are called **var-arg** methods.

→ We can declare var-arg method as : **methodname(int... variable);**

Ex: m1(int... x);

→ We can invoke this method by passing any number of int values include zero number also.

```
m1();
m1(10,20);
m1(10);
m1(10,20,30,40);
```

→ Internally var-arg method is implemented by using single dimensional arrays concept. Hence with in the var-arg method we can differentiate args by using index.

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
  * @fileName:TestVarArgs.java
 */
public class TestVarArgs {
        public static void sum(int... x) {
                System.out.println("Summation.....:");
                int total = 0;
                for (int y : x) {
                        total += y;
                        System.out.println(total);
                }
                System.out.println("total..=" + total);
        }

        public static void main(String[] args) {

                System.out.println("Var Args Demo...");
                sum();
                sum(10);
                sum(20, 30);
                sum(40, 50, 60, 70);
        }

}
```

```
/*
OutPut:
Var Args Demo...
Summation.....:
total..=0
Summation.....:
10
total..=10
Summation.....:
20
50
total..=50
Summation.....:
40
90
150
220
total..=220

*/
```

Inner classes

→ We can declare a class inside another class, such type of classes are called "Inner Classes".

→ Inner classes concept introduced in 1.1v to fix GUI bugs as the part Event handling.

→ Because of powerful features & benefits of Inner classes slowly programmers started using even in regular coding also.

→ without existing one type of object if there is no chance of existing another type object, then we should go for Inner class concept.

Ex:

1) → Without existing Car object, if there is no chance of exiting Wheel object then we should go for Inner classes.

We have to declare Wheel class with in the Car class.

```
class Car {
        class Wheel{
        }
}
```

2) → Without existing Bank object there is no chance of existing Account object. Hence we have to define Account class inside Bank class.

```
class Bank {
        class Account {
        }
}
```

3) → A Map is a collection of Key-value pairs and each key-value pair is called Entry. Without existing Map Object there is no chance of existing Entry object. Hence interface entry is defined inside Map interface.

```
interface Map {
        interface Entry {
        }
}
```

→ The relationship between inner & outer classes "is not parent to child" relationship, it is "**Has-A**" relationship.

→ Based on the purpose & position of declaration all inner classes are divided into the following 4 types:

1) Normal or Regular Inner Classes
2) Method Local Inner Classes
3) Annonymous Inner Classes (without class name)
4) Static Nested classes

From static Nested class we can access only static members of outer class directly. But in Normal Inner classes we can access both static & non-static members of outer class directly.

1)Normal or Regular Inner class:

→ If we declare any named class directly inside a class without static modifier, such type of class is called "Normal or Regular Inner class".

Ex1:

```
class Outer {
        class Inner {
        }
}
```

javac Outer.java → creates Outer.class & Outer$Inner.class

java Outer → RE:NoSuchMethodError:main
java Outer$Inner → RE:NoSuchMethodError:main

Ex2:

```
class Outer {
        class Inner {
        }

        public static void main(String args[]) {
                System.out.println("Outer class main()");
        }
}
```

javac Outer.java → creates Outer.class & Outer$Inner.class

java Outer → output: Outer class main()
java Outer$Inner → RE:NoSuchMethodError:main

Ex3:

Inside inner classes we **can't declare static members** hence, it is not possible to declare main() & hence we can't invoke inner class directly from command prompt.

```
class Outer {
        class Inner {
                public static void main(String args[]) {
                        System.out.println("Outer class main()");
                }
        }
}
```

javac Outer.java → CE:Inner classes can't have static declarations

Accessing Inner class code from static area of Outer class:

```
class Outer {
        class Inner {
                public void m1() {
                        System.out.println("Inner class m1()");
                }
        }

        public static void main(String args[]) {
                System.out.println("Outer class main()");
                Outer o = new Outer(); // or Outer.Inner i = new Outer().new Inner();
                Outer.Inner i = o.new Inner();
                i.m1(); // or new Outer().new Inner.m1();
        }
}

/*Output:
 Outer class main()
 Inner class m1()
*/
```

Accessing Inner class code from Instance area of Outer class:

```
class Outer {
        class Inner {
                public void m1() {
                        System.out.println("Inner class m1()");
                }
        }

        public void m2() {
                Inner i = new Inner();
                i.m1();
        }
```

```
        public static void main(String args[]) {
                Outer o = new Outer();
                o.m2();
        }
}

/*OUTPUT:
 Inner class m1()
*/
```

Accessing inner class code from outside of outer class:

```
class Outer {
        class Inner {
                public void m1() {
                        System.out.println("Inner class method m1()");
                }
        }
}

class Test {
        public static void main(String args[]) {
                Outer o = new Outer();
                Outer.Inner i = o.new Inner();
                i.m1();
        }
}

/*OUTPUT:
Inner class method m1()
*/
```

According to inner class code:

→ From static area of Outer class or from outside of Outer class

```
Outer o=new Outer();
Outer.Inner i=o.new Inner();
i.m1();
```

→ From Instance area of Outer class

```
Inner i=new Inner();
i.m1();
Outer o=new Outer();
```

→ From the Inner class we can access all members of Outer class (both static & non-static) directly.

```
class Outer {
        static int x = 10;
        int y = 20;

        class Inner {
                public void m1() {
                        System.out.println("x = "+x);
                        System.out.println("y = "+y);
                }
```

```
		}

		public static void main(String args[]) {
			new Outer().new Inner().m1();
		}
	}

	/*OUTPUT:
	x = 10
	y = 20
	*/
```

→ With in the Inner class "**this**" always pointing to current inner class object.

→ To refer current outer class object we have to use "Outerclassname.this".

Outerclassname.this

```
class Outer {
	int x = 10;

	class Inner {
		int x = 100;

		public void m1() {
			int x = 1000;
			System.out.println("x = " + x);
			System.out.println("Inner x ... this.x = " + this.x);
			System.out.println("Outer... Outer.this.x = " + Outer.this.x);
		}
	}

	public static void main(String args[]) {
		new Outer().new Inner().m1();
	}
}

/*OUTPUT:
x = 1000
Inner x ... this.x = 100
Outer... Outer.this.x = 10
*/
```

→ For the Outer classes(Top-level classes) the applicable modifiers are **public, <default>, final, abstract, strictfp**.

But, for the Inner classes in addition to above the following modifiers are also applicable.

private,protected, static

only for Outer classes → public, default, final, abstract, strictfp + private,protected, static = Inner class.

2) Method local Inner classes:

→ Some times we can declare a class inside a method such type of classes are called "**Method Local Inner classes**".

→ The main purpose of method local inner class is to define method specific functionality.

→ The scope of method local inner class is the method in which we declared it that is from outside of the method we can't access method local inner classes.

→ As the scope is very less, this type of inner classes are most rarely used inner classes.

```
class Test {
	public void m1() {
		class Inner {
			public void sum(int x, int y) {
				System.out.println("sum :" + (x + y));
			}
		}
		Inner i = new Inner();
		i.sum(10, 20);
		i.sum(100, 200);
	}

	public static void main(String args[]) {
		new Test().m1();
	}
}
/*OUTPUT:
 sum :30
 sum :300
*/
```

→ We can declare Inner class either in instance method or in static method.

→ If we declare inner class inside Instance method then we can access both static & non-static variables of outer class directly from that inner class..

→ If we declare inner class inside static method then we can access only static members of outer class directly from that inner class.

```
class Test {
	int x = 10;
	static int y = 20;

	public void m1() {
		class Inner {
			public void m2() {
				System.out.println("x = " + x);
				System.out.println("y = " + y);
			}
		}
		Inner i = new Inner();
		i.m2();
	}

	public static void main(String args[]) {
		new Test().m1();
	}
```

```
}

/*OUTPUT:
x = 10
y = 20
*/
                                        **********
class Test {
        int x = 10;
        static int y = 20;

        public static void m1() {
                class Inner {
                        public void m2() {
                                // System.out.println(x);
                                System.out.println("y = " + y);
                        }
                }
                Inner i = new Inner();
                i.m2();
        }

        public static void main(String args[]) {
                new Test().m1();
        }
}

/*OUTPUT:
y = 20
*/
```

→ From method local inner class we can't access local variables of the method in which we declared it, but if that local variable declared as the final then we can access.

```
class Test {
        int x = 10;
        static int y = 20;

        public void m1() {
                int y = 20;
                class Inner {
                        public void m2() {
                                System.out.println(x);
                                System.out.println(y);
                        }
                }
                Inner i = new Inner();
                i.m2();
        }

        public static void main(String args[]) {
                new Test().m1();
        }
}
```

javac Test.java

CE: local variable y is accessed from within inner class; needs to be declared final
System.out.println(y);

→ Only the applicable modifiers for method Local inner class are: **final,abstract,strictfp**

```
class Test {
	int x = 10;
	static int y = 20;

	public void m1() {
		final int y = 20;
		class Inner {
			public void m2() {
				System.out.println("x = " + x);
				System.out.println("y = " + y);
			}
		}
		Inner i = new Inner();
		i.m2();
	}

	public static void main(String args[]) {
		new Test().m1();
	}
}

/*OUTPUT:
x = 10
y = 20
*/
```

3) Anonymous Inner class:

→ Sometimes we can declare a class without name also, such type of nameless inner classes are called Anonymous inner classes.

→ This type of inner classes are most commonly used type of inner classes.

→ There are 3 types of anonymous inner classes.

i) Anonymous Inner Class that extends a class.
ii) Anonymous Inner Class that implements an Interface.
iii)Anonymous Inner Class that defined inside method arguments.

i) Anonymous Inner Class that extends a class:

```
	class Popcorn {
		public void taste() {
			System.out.println("Salty");
		}
	}

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

```
                Popcorn p = new Popcorn() {
                        public void taste() {
                                System.out.println("Anonymous Inner class taste()...");
                                System.out.println("....Sweety");
                        }
                };
                p.taste(); // Sweety
                Popcorn p1 = new Popcorn();
                p1.taste(); // salty
        }
}

/*OUTPUT:
Anonymous Inner class taste()...
....Sweety
Salty
*/
```

→ 1) The internal class name generated for Anonymous inner class is "**Test$1.class**"

→ 2) Parent class reference can be used to hold child class object but by using that reference we can call only methods available in the parent class & we can't call child specific methods.

In Anonymous inner classes also we can define new methods but we can't call these methods from outside of the class because we are depending on parent reference, these methods just for internal purpose only.

Explanation:

Popcarn p=new Popcarn(); → Just we are creating an object of Popcorn class

Popcarn p=new Popcarn(){ }; → We are creating child class for the Popcorn & for that child class we are creating an object with parent reference.

Ex:

```
class Test {
        public static void main(String args[]) {
                Thread t = new Thread() {
                        public void run() {
                                for (int i = 0; i < 5; i++) {
                                        System.out.println("...Child Thread");
                                }
                        }
                };
                t.start();
                for (int i = 0; i < 5; i++) {
                        System.out.println("Main Thread");
                }
        }
}
/*OUTPUT:
Main Thread
Main Thread
```

```
Main Thread
Main Thread
Main Thread
...Child Thread
...Child Thread
...Child Thread
...Child Thread
...Child Thread
*/
```

→ from the above program both main & child Threads will be executed simultaneously & Hence we can't get exact output.

ii) Anonymous Inner Class that implements an Interface:

```
class Test {
	public static void main(String args[]) {
		Runnable r = new Runnable() {
			public void run() {
				for (int i = 0; i < 5; i++) {
					System.out.println("...Child Thread");
				}
			}
		};
		Thread t = new Thread(r);
		t.start();
		for (int i = 0; i < 5; i++) {
			System.out.println("Main Thread");
		}
	}
}
/*OUTPUT:
Main Thread
Main Thread
Main Thread
Main Thread
Main Thread
...Child Thread
...Child Thread
...Child Thread
...Child Thread
...Child Thread
*/
```

iii) Anonymous Inner Class that defined inside method arguments:

```
class Test {
	public static void main(String args[]) {
		new Thread(new Runnable() {
			public void run() {
				for (int i = 0; i < 5; i++) {
					System.out.println("...Child Thread");
				}
			}
		}).start();
```

```
                for (int i = 0; i < 5; i++) {
                        System.out.println("Main Thread");
                }
        }
}

/*OUTPUT:
Main Thread
Main Thread
Main Thread
Main Thread
Main Thread
...Child Thread
...Child Thread
...Child Thread
...Child Thread
...Child Thread
*/
```

4) Static nested classes:

→ Some times we can declare inner class with static modifier, such type of inner classes are called "static nested classes".

→ In the normal inner class, inner class object always associated with outer class object. i.e., without existing outer class object there is no chance of existing inner class object.

But, static nested class object is not associated with outer class object i.e., without existing outer class object there may be a chance of existing static nested nested class object.

Ex:

```
class Outer {
        static class Nested {
                public void m1() {
                        System.out.println("Static Nested class Method");
                }
        }

        public static void main(String args[]) {
                Outer.Nested n = new Outer.Nested();
                n.m1();
        }
}

/*OUTPUT:
Static Nested class Method
*/
```

→ With in the static nested class we can declare static members including main() also. Hence,it is possible to invoke nested class directly from command prompt.

```
class Outer {
        static class Nested {
                public static void main(String args[]) {
```

```
                        System.out.println("Static Nested class main() Method");
                    }
                }

                public static void main(String args[]) {
                    System.out.println("Outer class main() Method");
                }
            }
```

javac Outer.java

java Outer
Output:
Outer class main() Method

java Outer$Nested
Output:
Static Nested class main() Method

Normal Inner Class Vs Static Nested Class :

Inner class:
1)Inner class object is always associated with Outer class object. i.e., without existing Outer class object there is no chance of existing Inner class object.
2) Inside normal Inner class we can't declare static members.
3) Inside normal Inner class we can't declare main() and hence it is not possible to invoke inner class directly from command prompt.

Static Nested Class:
1)Static Nested class object is not associated with Outer class object. i.e.,with out existing Outer class object there is a chance of existing Static Nested class object.

2) Inside normal Static Nested class we can declare static members.

3) Inside normal Static Nested class we can declare main() and hence it is possible to invoke inner class directly from command prompt.

General class Vs Anonymous Inner Class:

→ A General class can extend only one class at a time.
Where as Anonymous Inner class also can extend only one class at time.

→ A General class can implement any number of Interfaces.
Where as Anonymous Inner class can implement only one interface at time.

→ A General class can extend another class & can implement an interface simultaneously.
Where as Anonymous Inner class can extend another class or can implement an interface but not both simultaneously.

enum

enum: Every enum in java is direct child class of "java.lang.Enum" always every enum is "final", because of this we can conclude inheritance concept is not applicable for enums explicitly. But enum can

impl any number of interfaces at a time.

→ **invalid:** calss x{} enum y extends x{}, enum x{}class y extends x{},enum x{}, enum y extends x, enum x extends java.lang.Enum {}
CTE:Cannot inherit from final x
enum types not extensible

→ **valid:** interface x{} enum y implements x{}

java.lang.enum: Is an abstract class and direct child class of Object class.
This class implements Comparable & Serializable interfaces. Hence every enum in java is by default **Serializable** and **Comparable**.

→ **values():** This method is used to list out all values of enum.
Beer[] b=Beer.values();

→ **ordinal():** Using this method we can find ordinal value of enum constant. ordinal value is the order value of enum constants

→ We can declare enum either with in the class or outside of the class but not inside a method.
→ If we are trying to declare enum with in a method we will get CTE
→ If we declare enum outside the class the applicable modifiers are public, default, strictfp.
→ If we declare enum with in a class the applicable modifiers are pub-lic,default,strictfp,private,protected, static.

From 1.5v onwards we can use enum as arg to switch statement.
switch(arg) args 1.4v → byte, short, char, int
1.5v → Byte, Short, Character, Integer, enum
1.6v → String

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
  * @fileName:EnumTest.java */

enum Beer {
        KF, RC, BW, KO;
}

class EnumTest {
        public static void main(String args[]) {
                Beer b1 = Beer.RC;
                switch (b1) {
                case KF:
                        System.out.println("King Fisher");
                        break;
                case RC:
                        System.out.println("Royal Challenge");
                        break;
                case KO:
                        System.out.println("KnockOut");
                        break;
```

```
                case BW:
                        System.out.println("BudWiser");
                        break;
                default:
                        System.out.println("Other Brands not recommanded");
                }
        }
}

/*
Output:
java EnumTest

Royal Challenge
*/
```

java.lang.enum: Is an abstract class and direct child class of Object class.

→ This class implements Comparable & Serializable interfaces. Hence every enum in java is by default Serializable and Comparable.

values(): This method is used to list out all values of enum.

```
Beer[] b = Beer.values();
```

ordinal(): using this method we can find ordinal value of enum constant. ordinal value is the order value of enum constants

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
  * @fileName:EnumTest1.java
 */
enum Beer {
        KF, RC, BW, KO;
}
class EnumTest1 {
        public static void main(String args[]) {
                Beer[] b = Beer.values();
                for (Beer b1 : b) {
                        System.out.println(b1 + "---" + b1.ordinal());
                }

        }
}

/*
Output:
java EnumTest1
KF---0
RC---1
BW---2
KO---3
*/
```

→ Inside enum we can declare main() & hence, we can invoke enum class directly from command prompt.

→ In addition to constant if we want to take any extra members compulsory list of constants should be in the 1st line & should ends with ";".

ex: enum Colour {
RED, GREEN, BLUE;
public void m1() {
}
}

→ Inside enum without having constant we can't to take any extra members, but empty enum is always valid.

invalid: enum Colour {
public void m1() {
}
}

valid : enum Colour {
}

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
  * @fileName:EnumTest2.java
 */

enum EnumTest2 {
	KF, RC, BW, KO;
	public static void main(String args[]) {
		System.out.println("ENUM main method...");
		EnumTest2[] b = EnumTest2.values();
		for (EnumTest2 b1 : b) {
			System.out.println(b1 + "---" + b1.ordinal());
		}

	}
}

/*
Output:
java EnumTest2

ENUM main method...
KF---0
RC---1
BW---2
KO---3

*/
```

enum class constructors:

→ with-in enum we can take constructors also.

→ enum class constructors will be executed automatically at the time of enum class loading, because enum constants will be created at the time of class loading only.

→ we can't invoke enum constructors explicitly. we can't create Objects of enum explicitly & hence we can't call constructors directly. If we try it shows CTE

Beer b = new Beer(); CTE: enum types may not be instantiated.

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
  * @fileName:EnumTest3.java
 */
enum Beer {
	KF, RC, BW, KO;
	Beer() {
		System.out.println("enum constuctor...");
	}
}

class EnumTest3 {
	public static void main(String args[]) {
		System.out.println("main method...");
		Beer b = Beer.KF;
		System.out.println(b + "---" + b.ordinal());
	}
}
/*
Output:
java EnumTest3

main method...
enum constuctor...
enum constuctor...
enum constuctor...
enum constuctor...
KF---0

*/
```

→ Within the enum we can take instance & static methods, but we can't to take abstract methods.

→ Every enum constant represents an Object hence whatever the methods we can apply on normal java Object we can apply those on enum constants also.

→ **enum:** It is a keyword which can be used to define a group of named constants.

→ **Enum:** It is a class present in "java.lang" pkg which acts as a base class for all java enums.
Enumeration: It is an Interface present in "java.util" pkg, which can be used for retrieving objects from Collection one by one.

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
  * @fileName:EnumTest4.java
 */

enum Colour {
	BLUE, RED {
		public void info() {
			System.out.println("Dangerous Colour...");
		}
	},
	GREEN;
```

```
        public void info() {
                System.out.println("Universal Colour...");
        }
}

class EnumTest4 {
        public static void main(String args[]) {
                Colour[] c = Colour.values();
                for (Colour c1 : c) {
                        c1.info();
                }
        }
}

/*
Output:
java EnumTest4

Universal Colour...
Dangerous Colour...
Universal Colour...
*/
```

Thread

Multitasking:

→ Executing several tasks simultaneously is called "Multitasking".

→ There are two types of multitasking.

1) Process-based multitasking.
2) Thread-based multitasking.

1) Process-based multitasking:

→ Executing several tasks simultaneously, where each task is a separate independent process, is called *process based multitasking*.

Ex: While typing a java program in editor we can able to listen audio songs by Mp3 player in the system. At the same time we can download a file from the net. All these tasks are executing simultaneously & independent of each other. Hence, it is process-based multitasking.

→ It is best suitable at *Operating System* level.

2) Thread-based multitasking:

→ Executing several tasks simultaneously where each task is a separate independent part of the same program is called Thread based multitasking & each independent part is called Thread.

→ It is best suitable for "Programmatic Level"

→ The main important application areas of Multi-Threading are developing video games, multimedia Graphics, implementing animations...

→ Whether it is process-based or Thread-based the main objective of multitasking is to improve performance of the system by reducing Response-time.

→ Java provides inbuilt support for multithreading by introducing a rich API(Thread, Runnable, ThreadGroup, ThreadLocal..).

→ The ways to define a new Thread:
1) By extending Thread class
2) By implementing Runnable Interface.

1) By extending Thread class :

```
class MyThread extends Thread {
	public void run() {
		for (int i = 1; i <= 5; i++) {
			System.out.println("Child Thread.." + i);
		}
	}
}

class ThreadTest {
	public static void main(String args[]) { // main thread
		MyThread t = new MyThread(); // instantiation of Thread
		t.start(); // starting of a Thread

		for (int i = 1; i <= 5; i++) {
			System.out.println("Main Thread.." + i);
		}
	}
}
```

Thread Schedular:→ Thread Schedular is the part of JVM. Whenever multiple Threads are waiting to get chance for execution which Thread will get chance first is decided by Thread Schedular. Whose behaviour is JVM vender dependent. Hence we can't expect exact execution order and hence exact output.

t.start() & t.run(): → In the case of **t.start()** a **new Thread** will be created & that Thread is responsible to execute **run()**.
→ But in case of t.**run()** no new Thread will be created & *run() will be executed just like a normal method call*.

Importance of Thread class start():

→ To start a Thread, the required mandatory activities(like registering Thread with Thread Scheduler) will be performed automatically by Thread class start() method. without executing start() there is no chance of starting a new Thread.

run(): If we are not overriding run() method, then thread class run() will be executed which has empty implementation. Hence, we won't get any output. It is used to define our job.

Overloading run(): Overloading of the run() is possible, but Thread class start() will always call no argument run() only. We have to call the other run() explicitly, just like a normal method call.

```
class MyThread extends Thread {
```

```
        public void run() {
                System.out.println("run() method..");
        }

        public void run(int x) {
                {
                        System.out.println("run(int x)... " + x);
                }
        }
}
class ThreadTest {
        public static void main(String args[]) { // main thread
                MyThread t = new MyThread();
                t.start(); // run() is executed automatically
                t.run(5); // we should call parametrized run(int x)
        }
}
/*Output:
java ThreadTest

run(int x)... 5
run() method..
*/
```

Overloading start(): If we overload start(), then start() will be executed just like a normal method call and no new Thread will be created.

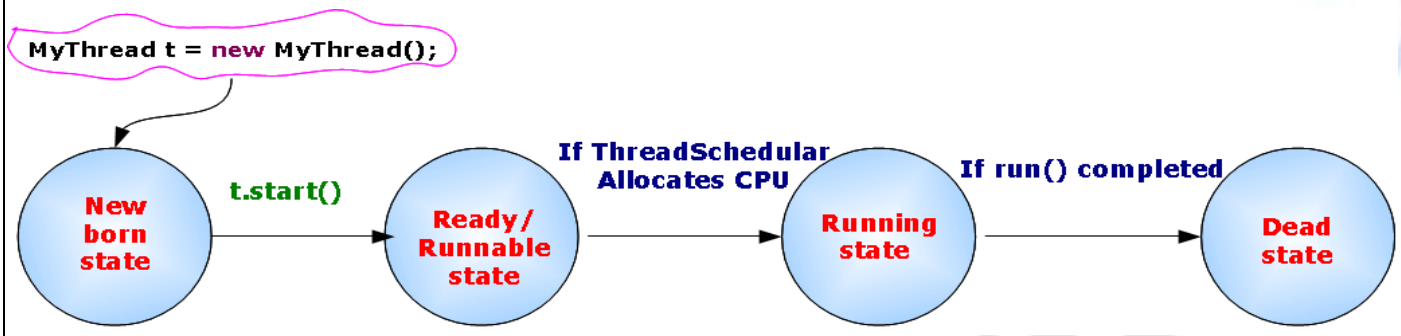
```
class MyThread extends Thread {
        public void start() {
                System.out.println("start() method..");
        }

        public void run() {
                {
                        System.out.println("run() method..");
                }
        }
}

class ThreadTest {
        public static void main(String args[]) { // main thread
                MyThread t = new MyThread();
                t.start(); // executes MyThread start() only. not Thread start()
        }
}

/*Output:
java ThreadTest

start() method..
*/
```

→ Once we **created** a *Thread object* then it is said to be in "**new state** or **born state**".

→ If we call **start()**, then the Thread will be entered into "**ready** or **runnable state**".

→ If **ThreadSchedular** allocates CPU, then the Thread will entered into "**running state**".

→ If **run()** completes then the Thread will entered into "**Dead State**".

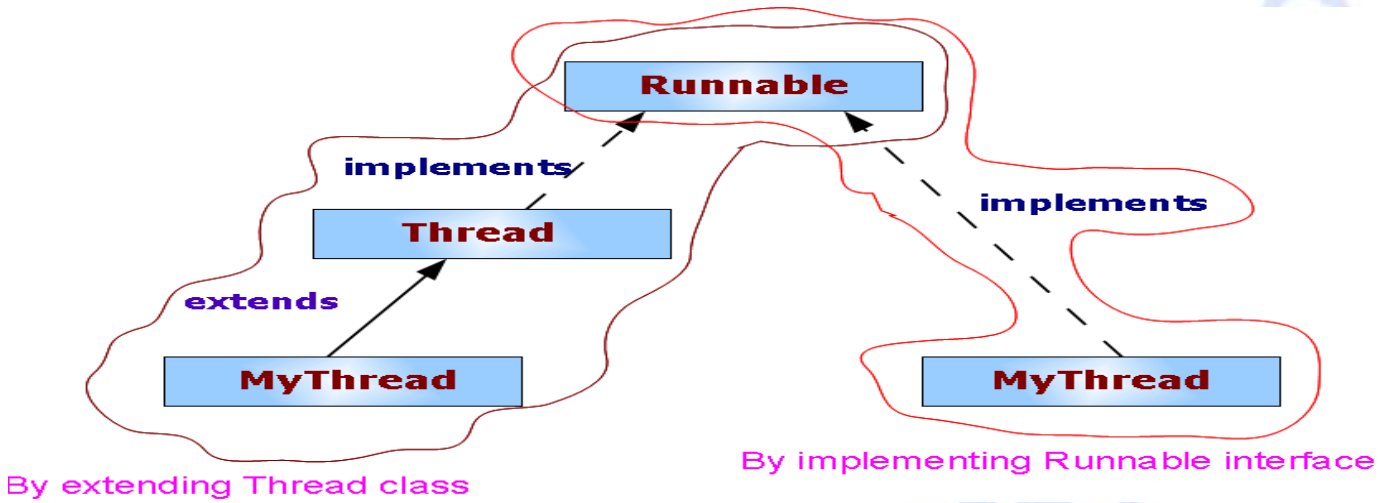


→ After *starting a Thread* we are *not allowed to restart the same Thread* once again. *Otherwise* we will get **RE:** "**IllegalThreadStateException**".

```
Thread t = new Thread();
t.start();
t.start();
```

→ With in the run(), if we call **super.start()**, we'll get the same **RE:** "**IllegalThreadStateException**".

```
class MyThread extends Thread {
        public void run() {
                {
                        super.start(); // calling super.start()
                        System.out.println("run() method..");
                }
        }
}
class ThreadTest {
        public static void main(String args[]) { // main thread
                MyThread t = new MyThread();
                t.start();
        }
}
```



2) By implementing Runnable Interface:

Runnable Thread: We can define a Thread even by implementing "Runnable" Interface also.
"Runnable" Interface present in java.lang package and contains only one method "**join()**" method

Ex: MyThread "implements" Runnable
MyThread "extends" Thread "implements" Runnable

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
  * @fileName:RunnableThreadDemo.java
 */
class MyRunnable implements Runnable {
	public void run() {
		for (int i = 1; i <= 5; i++) {
			System.out.println("child Thread:" + i);
			}
		}

	}
}

class RunnableThreadDemo {
	public static void main(String[] args) throws InterruptedException {
		MyRunnable r = new MyRunnable();
		Thread t = new Thread(r); // target runnable "r".
		t.start();

		for (int i = 1; i < 5; i++) {
			System.out.println("Main Thread:" + i);
		}
	}
}
```

Best approach to define a Thread:

→ Among the two ways of defining a Thread , the 2nd approach i.e., Thread implements Runnable mechanism is *recommended* to use.

→ In the *1st approach our class* always extending Thread class and hence, there is *no chance of extending any other class*. But, in the 2nd approach we can extend some other class also while implementing Runnable interface. Hence, 2nd approach is recommended to use.

```
class MyThread extends Thread {
        public void run() {
                {
                        System.out.println("Child..");
                }
        }
}

class ThreadTest {
        public static void main(String args[]) { // main thread
                MyThread t = new MyThread();// child thread
                Thread t1 = new Thread(t);
                t1.start();
                System.out.println("Main..");
        }
}
```

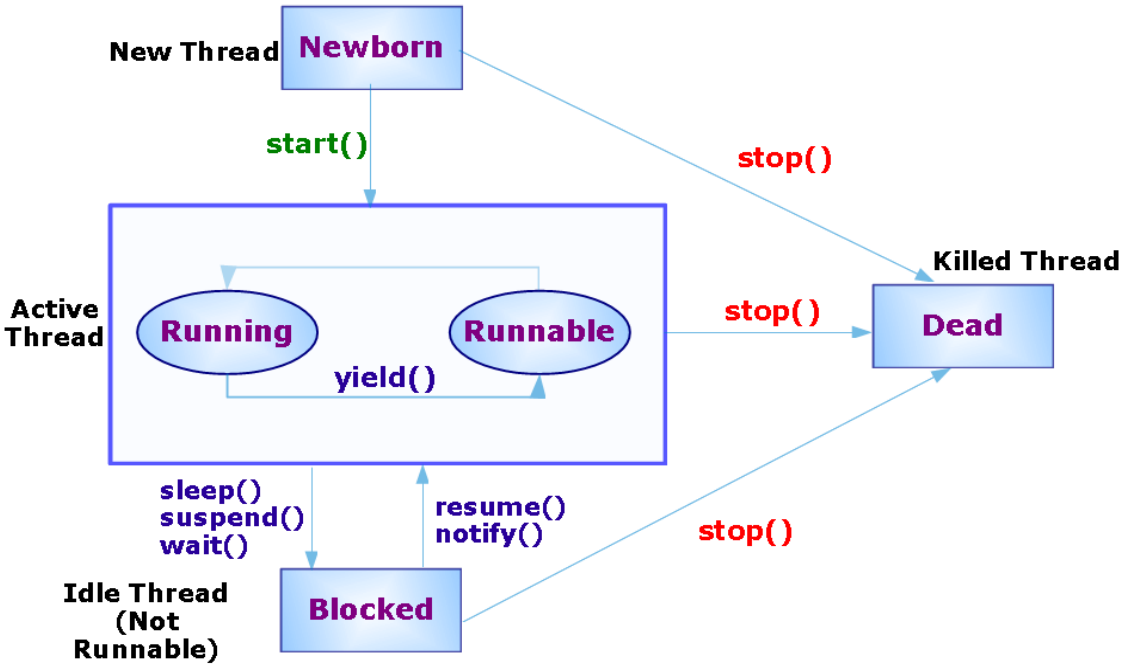
Thread Life Cycle:

Thread has many different state through out its life.

1. Newborn State
2. Runnable State
3. Running State
4. Blocked State
5. Dead State

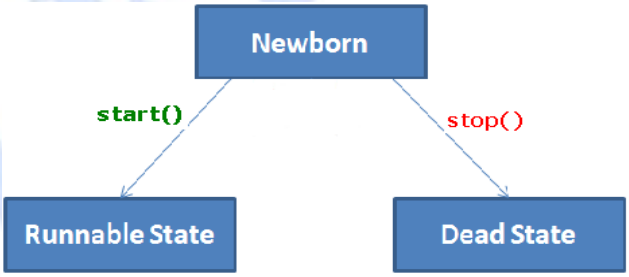
Thread should be in any one state of above and it can be move from one state to another by different methods and ways.

State Transition diagram of a Thread



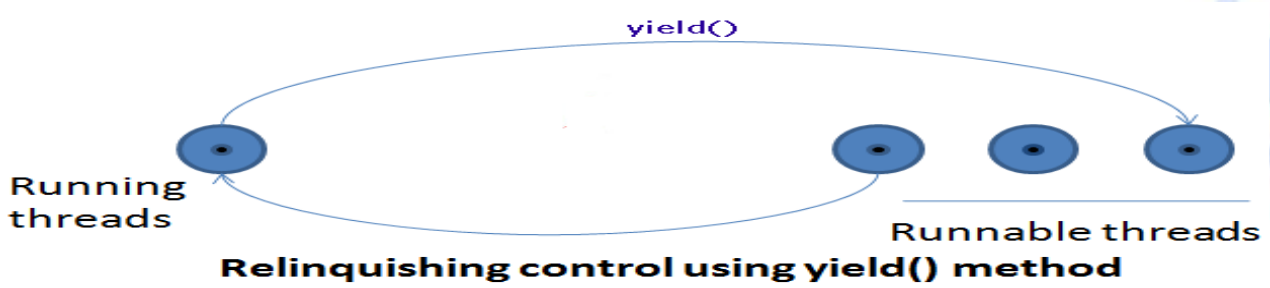
1. Newborn State: When we create a thread it will be in Newborn State.
The thread is just created still its not running.

We can move it to running mode by invoking the **start()** method and it can be killed by using **stop()** method.



Scheduling a newborn thread

2. Runnable State: It means that thread is now ready for running and its waiting to give control.



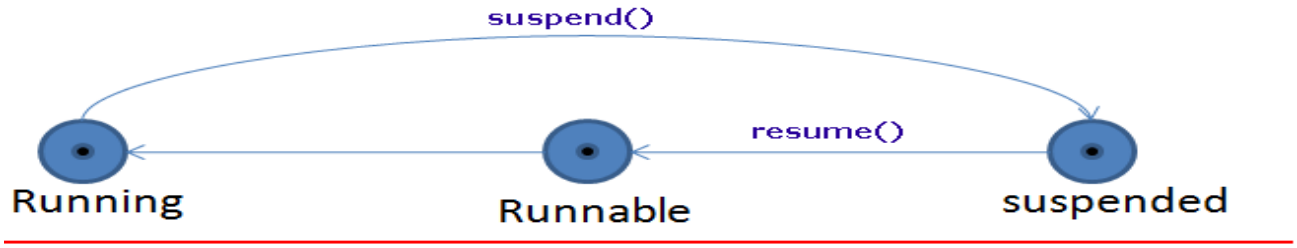
Relinquishing control using yield() method

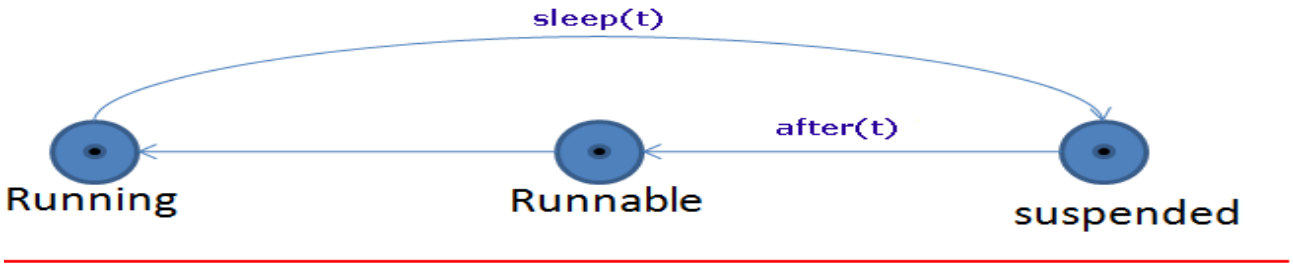
We can move control to another thread by **yield()** method.

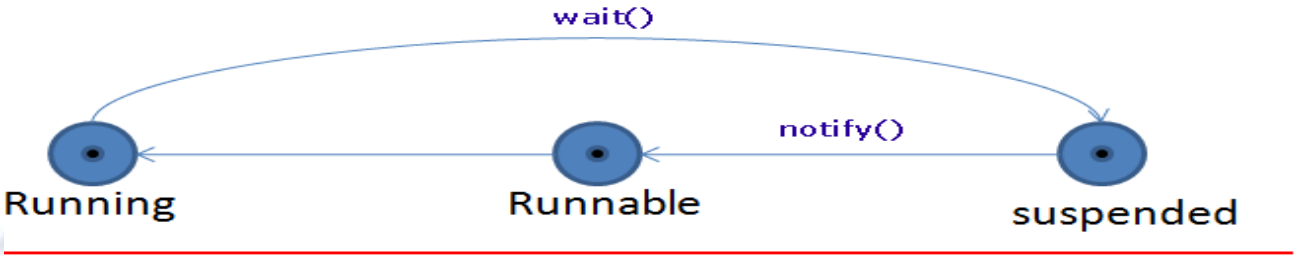
3. Running State: It means thread is in its execution mode because the control of CPU is given to that particular thread.

It can be move in three different situation from running mode.

These all are different methods which can be apply on running thread and how the state is changing and how we can come in our original previous state using different methods are shown in the figure.







4 .Blocked State : A thread is called in Blocked State when it is not allowed to entering in Runnable State or Running State.

It happens when thread is in waiting mode, suspended or in sleeping mode.

5. Dead State : When a thread is completed executing its **run()** method the life cycle of that particular thread is end.

We can kill thread by invoking **stop()** method for that particular thread and send it to be in Dead State.

****

1)**Create Thread –** Thread can be create through **Thread Class** or **Runnable interface**.

2)**New State –** when any object of thread is being made by **new Thread()**

3)**Runnable –** When start method of thread is called means thread is now in runnable state.

4)**Running –** After Runnable state, based on the priorities of thread, is being to running state.

5)**Dead –** After running state, thread will go to Dead. Or it can be block states.

6)**Block States –** Thread can be in block state by calling **wait()**, **sleep()**, **suspend()** methods.

7)**Sleep State –** When **sleep()** method is called by thread, thread will go to the sleep state. Sleep State means, This thread is still alive and it can be go on runnable state. Thread has sleep method for passing the time in millisecond. And throw **InterruptedException**.

static void sleep(long millisecond) throws InterruptedException

8)**Wait State–** when **wait()** method is called by any thread, thread will goes to waiting state, and wait for notification by another thread. Through calling the **notify()** and **notifyAll()** method, thread will goes to runnable state.

9)**Join State–** When **join()** method is called by thread, it will go to the join state, when joined thread is completed then this thread again go to the running state.

10)**yield()**- By invoking this method the current thread pause its execution temporarily and allow other threads to execute.

To get & set Name of a Thread:

public final String getName();
public final void setName(String name);

→ **We can get current executing Thread reference using "currentThread()"**

public static Thread currentThread();

```
class ThreadTest {
        public static void main(String args[]) {
                System.out.println(Thread.currentThread().getName());
                Thread.currentThread().setName("Guru");
                System.out.println(Thread.currentThread().getName());
        }
```

```
}

/*Output:
main
Guru
*/
```

Thread priority: Every Thread in java has some priority, but the range of Thread priorities is "**1** to **10**".

→ Thread class defines the following constants to define some standard priorities:

Thread.MIN_PRIORITY → 1
Thread.NORM_PRIORITY → 5
Thread.MAX_PRIORITY → 10

→ Thread Scheduler will use these priorities while allocating CPU.

→ The Thread which is having highest priority will get chance first,

→ If two Threads having same priority then we can't expect exact execution order, it depends on Thread Scheduler.

→ **Default priority:** The default priority only for the "main" Thread is 5.But, for all the remaining Threads it will be Inheriting from the parent i.e., whatever the priority parent has the same priority will be inheriting to the child.

→ **To get & set Thread Priority:**

public final int getPriority();
public final void setPriority(int p); → allowed values are 1 to 10, otherwise RE:IllegalArgumentException.

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
  * @fileName:RunnableThreadDemo.java
 */
class MyRunnable implements Runnable {
	public void run() {
		for (int i = 1; i < 5; i++) {
			System.out.println("child Thread:" + i);
			try {
				Thread.sleep(3000);
			} catch (Exception e) {
			}
		}

	}
}

class RunnableThreadDemo {
	public static void main(String[] args) throws InterruptedException {
		MyRunnable r = new MyRunnable();
		Thread t = new Thread(r);
		t.start();
		t.join(); // Main Thread execution is waiting, after "t" Thread execution only main Thread
```

```
                        executed.
                for (int i = 1; i < 5; i++) {
                        System.out.println("Main Thread:" + i);
                }
        }
}

/*
OUTPUT:
java RunnableThreadDemo

child Thread:1
child Thread:2
child Thread:3
child Thread:4
Main Thread:1
Main Thread:2
Main Thread:3
Main Thread:4
*/
```

The methods to prevent Thread execution:

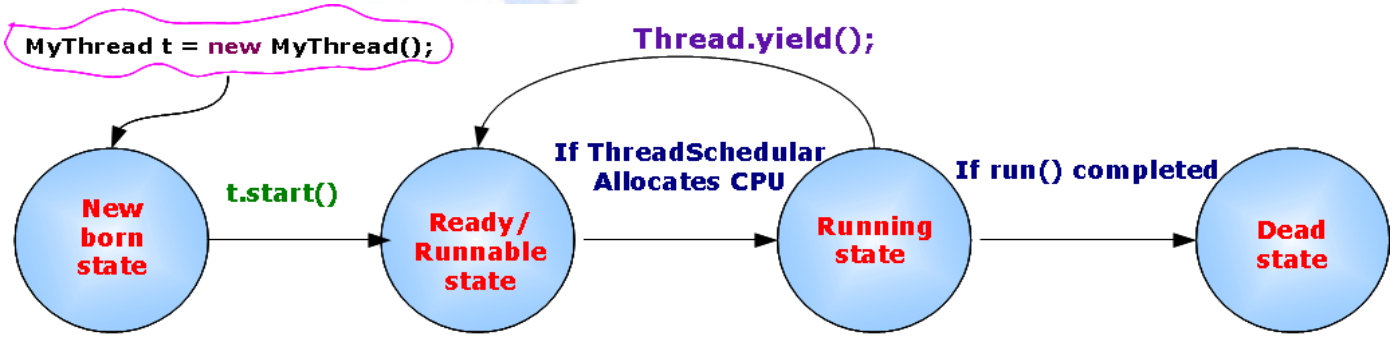
→ We can prevent thread from execution by using the following methods: i)**yield()** iii)**sleep()** iii)**join()**

i)yield() : This method causes, to pause current executing Thread for giving the chance to remaining waiting Threads of same priority.

→ If there are no waiting Threads or all waiting threads have low priority then the same Thread will continue it's execution once again.

→ The Thread which is yielded, when it will get chance once again for execution is decided by Thread Scheduler, we can't expect exactly.

public static native void yield()



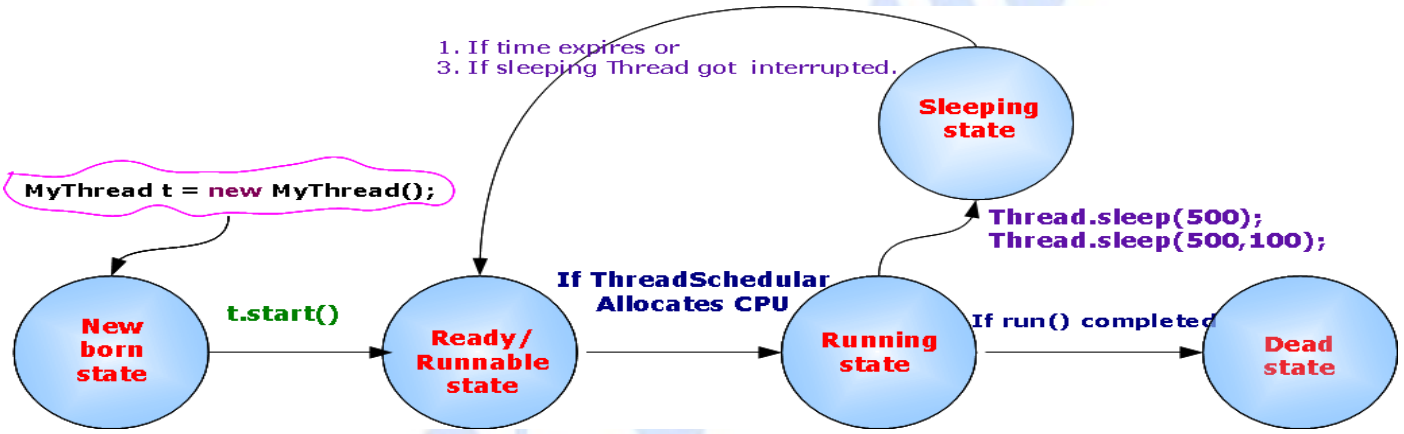

```
class MyThread extends Thread {
        public void run() {
                for (int i = 1; i <= 5; i++) {
                        Thread.yield(); //calling yield()
                        System.out.println("Child Thread.." + i);
                }
        }
}
```

```
class ThreadYieldTest {
	public static void main(String args[]) {
		MyThread t = new MyThread();
		t.start();

		for (int i = 1; i <= 5; i++) {
			System.out.println("Main Thread.." + i);
		}
	}
}
```

ii)sleep(): If a Thread don't won't to perform any operation for a particular amount of time(just pausing) then we should go for sleep().

public static void sleep(long ms) throws InterruptedException
public static void sleep(long ms, int ns) throws InterruptedException

Whenever we use "**sleep()**", we should handle "InterruptedException" using "**throws**", otherwise we will get CE.

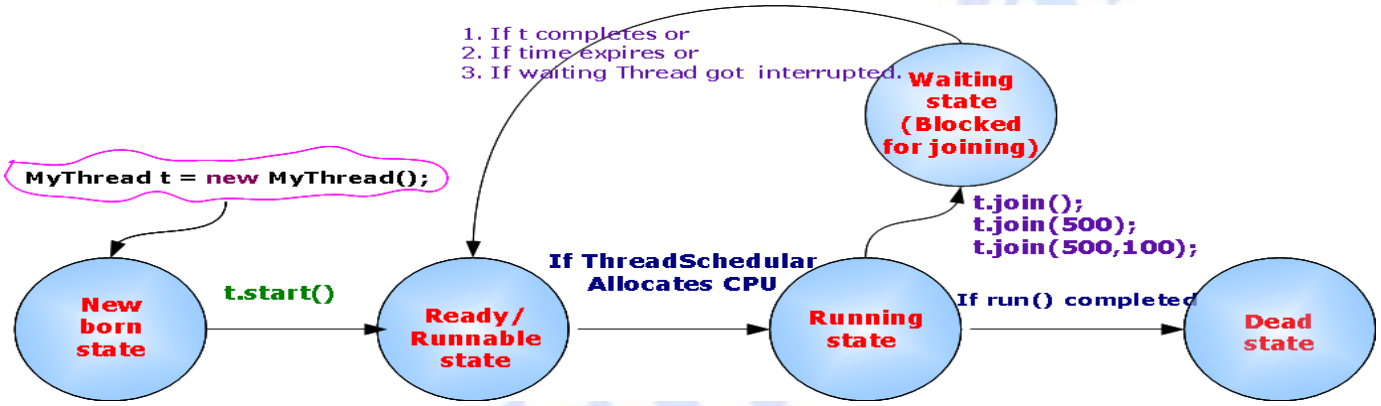



```
/**@author:VenukumarS  @date:Oct 6, 2013
 * @fileName:ThreadSleepTest.java
*/
class ThreadSleepTest {
	public static void main(String args[]) throws InterruptedException {
		System.out.println("Main Thread start..");
		Thread.sleep(2000);
		System.out.println("After Sleep .. Main Thread..");

	}
}
/*
Output:
java ThreadSleepTest
Main Thread start..
After Sleep .. Main Thread..
*/
```

iii) join(): If a Thread wants to wait until completing some other Thread then we should go for "**join()**" method.

```
class ThreadJoinTest {

        public static void main(String[] args) throws InterruptedException {
                MyThread t = new Thread();

                t.start();
                t.join();

                System.out.println("Main Thread:" + i);
        }
}
```

If "**main**" Thread executes, "**t.join()**" then, "**main**" Thread will entered into waiting state until "**t**" will continue its execution. Once **t** completes then "**main**" will continue its execution.



whenever we use "join()", we should handle "InterruptedException" using "throws", otherwise we will get CE.

```
public final void join() throws InterruptedException
public final void join(long ms) throws InterruptedException
public final void join(long ms, int ns) throws InterruptedException
```

→ join() is overloaded and every *join() throws InterruptedException*. Hence, whenever we are using join() compulsory we *should handle* InterruptedException, either by **try-catch** or by **throws**, otherwise we will get CE.

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
  * @fileName:RunnableThreadDemo1.java
 */

class MyRunnable1 implements Runnable {
        public void run() {
                for (int i = 1; i < 5; i++) {
                        System.out.println("child Thread:" + i);
                        try {
                                Thread.sleep(1000);
                        } catch (Exception e) {
```

```
                        }
                }

        }
}

class RunnableThreadJoinTest {
        public static void main(String[] args) throws InterruptedException {
                MyRunnable1 r = new MyRunnable1();
                Thread t = new Thread(r);
                t.start();
                t.join(); //Line-19..calling join() which blocks the "main Thread execution"
                for (int i = 1; i < 5; i++) {
                        System.out.println("Main Thread:" + i);
                }
        }
}

/*Output:
 java RunnableThreadJoinTest
child Thread:1
child Thread:2
child Thread:3
child Thread:4
Main Thread:1
Main Thread:2
Main Thread:3
Main Thread:4
*/
```

→ The **main** Thread will wait until completing **child(t)** Thread. Hence, in this case ht output is expected i.e. on the completion of **child(t)** Thread execution, it executes the **main** Thread.

→ If we commenting ***Line-19***, then both Threads will be executed simultaneously and we can't expect exact execution order, hence, we can't expect exact output. It is as follows:

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
  * @fileName:RunnableThreadJoinTest.java
 */

class MyRunnable1 implements Runnable {
        public void run() {
                for (int i = 1; i < 5; i++) {
                        System.out.println("child Thread:" + i);
                        try {
                                Thread.sleep(1000);
                        } catch (Exception e) {
                        }
                }

        }
}

class RunnableThreadJoinTest {
        public static void main(String[] args) throws InterruptedException {
```

```
                MyRunnable1 r = new MyRunnable1();
                Thread t = new Thread(r);
                t.start();
            //t.join();
                for (int i = 1; i < 5; i++) {
                        System.out.println("Main Thread:" + i);
                }
        }
}

/*
java RunnableThreadJoinTest
Main Thread:1
Main Thread:2
Main Thread:3
Main Thread:4
child Thread:1
child Thread:2
child Thread:3
child Thread:4
*/


/**@author:VenukumarS  @date:Oct 6, 2013
 * @fileName:ThreadJoinTest.java
 */

class MyThread extends Thread {
        public void run() {
                for (int i = 1; i <= 5; i++) {
                        System.out.println("Child Thread.." + i);
                        try {
                                Thread.sleep(500);
                        } catch (InterruptedException ie) {
                        }
                }
        }
}

class ThreadJoinTest {
        public static void main(String args[]) throws InterruptedException {
                MyThread t = new MyThread();
                t.start();
                t.join();

                for (int i = 1; i <= 5; i++) {
                        System.out.println("Main Thread.." + i);
                }
        }
}

/*Output:
java ThreadJoinTest
Child Thread..1
```

```
Child Thread..2
Child Thread..3
Child Thread..4
Child Thread..5
Main Thread..1
Main Thread..2
Main Thread..3
Main Thread..4
Main Thread..5
*/
```

yield(), join(), sleep() Comparison :

Property	yield()	sleep()	join()
1.Purpose	To pause current executing Thread to give the chance for the remaining Threads of same priority	If a Thread don't want to perform any operation for a particular amount of time(pausing) go for sleep()	If a thread want to wait until completing some other Thread then we should go for join()
2.static	Yes	Yes	No
3.Is it overloaded	No	Yes	Yes
4.Is it final	No	No	Yes
5.Is it throws InterruptedException	No	Yes	Yes
6.Is it native method	Yes	sleep(long ms) → native sleep(long ms, int ns) → non-native	No

Interrupt:

Interruption of a Thread:

→ A thread can Interrupt another sleeping or waiting Thread.

→ For this Thread class defines "interrupt()" method.

Syntax: public void interrupt()

→ We mayn't see the impact if interrupt call immediately.

→ Whenever we are calling interrupt() method, if the target Thread is not in sleeping or waiting state then there is no impact immediately. Interrupt call will wait until target Thread entered into sleeping or waiting state. Once target Thread entered into sleeping or waiting state the interrupt call will impact the target Thread.

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
  * @fileName:InterruptDemo.java
 */

class MyThread extends Thread {
        public void run() {
                try {
```

```
                for (int i = 1; i < 5; i++) {
                        System.out.println(i + "Lazy Thread");
                        Thread.sleep(2000);
                }
            } catch (InterruptedException e) {
                System.out.println("I got Interrupted");
            }
        }
}

class InterruptTest {
        public static void main(String args[]) {
            MyThread t = new MyThread();
            t.start();
            t.interrupt();
            System.out.println("End of Main");
        }
}
/*
OUTPUT:
java InterruptTest

End of Main
1Lazy Thread
I got Interrupted
*/
```

Synchronization:

→ **synchronized** is the modifier applicable only for methods & blocks and we can't apply for classes and variables.

→ If a **method or block** declared as **synchronized** then at a time **only one Thread** is allowed to execute that **method or block** on the given object.

→ The main advantage of **synchronized** keyword is we can resolve data inconsistency problem.

→ The main limitation of **synchronized** keyword is it increases waiting time of the Threads & effects performance of the system. Hence, if there is no specific requirement it is never recommanded to use.

→ Every object in java has a unique lock synchronization concept internally implemented by using this Lock concept. Whenever we are using synchronization then only **Lock** concept will active.

→ If a **Thread** wants to execute any **synchronized** method on the given object, first it has **to get the lock of that object**. Once a Thread get a lock, then it allowed to execute any synchronized method on that object.

→ Once synchronized method completes, then automatically the lock will be released.

→ While a Thread executing any synchronized method on the given object the remaining Threads are not allowed to execute any synchronized method on the given object simultaneously. But, remaining Threads are allowed to execute any non-synchronized methods simultaneously (**Lock concept is implemented based on object, but not based on method**).

Ex:

```
class X {
```

```
                public synchronized void m1() {
                }

                public void m3() {
                }

        }
```

Program:

```
        class X {
                public synchronized void m1() {
                        for (int i = 0; i < 3; i++) {
                System.out.println("."+Thread.currentThread().getName()+"....m1()..");
                                try {
                                        Thread.sleep(2000);
                                } catch (InterruptedException ie) {
                                }
                        }
                }

                public void m3() {
                        for (int i = 0; i < 3; i++) {
                System.out.println("."+Thread.currentThread().getName()+"....m3()..");
                                try {
                                        Thread.sleep(2000);
                                } catch (InterruptedException ie) {
                                }
                        }
                }

        }

        class MyThread3 extends Thread {
                X x;

                MyThread3(X x) {
                        this.x = x;
                }

                public void run() {
                        x.m1();
                        x.m3();
                }
        }

        public class SynchronizedTest2 {
                public static void main(String args[]) {
                        X x1 = new X();
                        MyThread3 t1 = new MyThread3(x1);
                        MyThread3 t2 = new MyThread3(x1);

                        t1.start();
                        t2.start();
                }
        }
```

```
/*Output:
java SynchronizedTest
.Thread-0....m1()..
.Thread-0....m1()..
.Thread-0....m1()..
.Thread-0....m3()..
.Thread-1....m1()..
.Thread-1....m1()..
.Thread-0....m3()..
.Thread-0....m3()..
.Thread-1....m1()..
.Thread-1....m3()..
.Thread-1....m3()..
.Thread-1....m3()..
*/
```

Lock: It is an **Object** associated with for every Object.

```
class Display {
	public synchronized void wish(String name) {
		for (int i = 0; i < 3; i++) {
			System.out.print("Good Morning..");
			try {
				Thread.sleep(2000);
			} catch (InterruptedException ie) {
			}
			System.out.println(name);
		}
	}
}

class MyThread extends Thread {
	Display d;
	String name;

	MyThread(Display d, String name) {
		this.d = d;
		this.name = name;
	}

	public void run() {
		d.wish(name);
	}
}

public class SynchronizedTest {
	public static void main(String args[]) {
		Display d1 = new Display();
		MyThread t1 = new MyThread(d1, " Guru ");
		MyThread t2 = new MyThread(d1, " Raam ");
		t1.start();
		t2.start();
	}
```

```
}
/*
Output:
        javac SynchronizedTest.java

        java SynchronizedTest
        Good Morning.. Guru
        Good Morning.. Guru
        Good Morning.. Guru
        Good Morning.. Raam
        Good Morning.. Raam
        Good Morning.. Raam
*/
```

→ If we declare wish() method as synchronized then Threads will be executed one by one so that we will get regular output.

```
class Display1 {
        public void wish(String name) {
                for (int i = 0; i < 3; i++) {
                        System.out.print("Good Morning..");
                        try {
                                Thread.sleep(2000);
                        } catch (InterruptedException ie) {
                        }
                        System.out.println(name);
                }
        }
}

class MyThread1 extends Thread {
        Display1 d;
        String name;

        MyThread1(Display1 d, String name) {
                this.d = d;
                this.name = name;
        }

        public void run() {
                d.wish(name);
        }
}

public class AsynchronizedTest {
        public static void main(String args[]) {
                Display1 d1 = new Display1();
                MyThread1 t1 = new MyThread1(d1, " Guru ");
                MyThread1 t2 = new MyThread1(d1, " Raam ");
                t1.start();
                t2.start();
        }
}
/*
```

```
Output:
        javac AsynchronizedTest.java

        java AsynchronizedTest
        Good Morning..Good Morning.. Raam
        Good Morning.. Guru
        Good Morning.. Raam
        Good Morning.. Guru
        Good Morning.. Raam
        Guru
*/
```

→ If we are not declaring **wish()** method as synchronized, then both Threads will be executed simultaneously & we can't expect exact output we will get irregular output.

```
class Display {
        public synchronized void wish(String name) {
                for (int i = 0; i < 3; i++) {
                        System.out.print("Good Morning..");
                        try {
                                Thread.sleep(2000);
                        } catch (InterruptedException ie) {
                        }
                        System.out.println(name);
                }
        }
}

class MyThread extends Thread {
        Display d;
        String name;

        MyThread(Display d, String name) {
                this.d = d;
                this.name = name;
        }

        public void run() {
                d.wish(name);
        }
}

public class SynchronizedTest1 {
        public static void main(String args[]) {
                Display d1 = new Display2();
                Display d2 = new Display2();
                MyThread t1 = new MyThread(d1, " Guru ");
                MyThread t2 = new MyThread(d2, " Raam ");
                t1.start();
                t2.start();
        }
}
/*
Output:
        javac SynchronizedTest.java
```

```
                Good Morning..Good Morning.. Raam
                Good Morning.. Guru
                Good Morning.. Guru
                Good Morning.. Raam
                Good Morning.. Guru
                Raam
        */
```

→ In this case eventhough ***wish()*** method is ***synchronized*** we will get irregular output. Because the Threads are operating on different objects.

Reason: Whenever multiple Threads are operating on same object, then only synchronization play the role. If multiple Threads are operating on multiple objects then there is no impact of synchronization.

Class level Lock:

→ Every class in java has a unique lock.

→ If a Thread wants to execute a static synchronized method then it required class-level lock.

→ While a thread executing static synchronized method, then the remaining Threads are not allowed to execute any static synchronized method of that class simultaneously, but remaining Threads are allowed to execute the following methods simultaneously.

1. normal static methods.
2. normal instance methods.
3.synchornized instance methods.

Ex:

```
class x {
        static synchronized void m1() {
        }

        static synchronized void m2() {
        }

        synchronized void m3() {
        }

        static void m4() {
        }

        void m5() {
        }
}
```

→ Let t1, t2, t3, t4, t5 and t6 are threads, Thread t1 executing a "**static synchronized method m1()**", then it requires class-level lock, remaiing threads i.e.,t2, t3, t4, t5 and t6 are not allowed to execute any static synchronized methods of the class X i.e.,m2().

→ The remaining Threads i.e., t2, t3, t4, t5 and t6 are allowed to execute m3(), m4(), m5() methods executed simultaneously.

Note: There is no link between **object level lock** and **class level lock** both are **independent** of each other.

synchronized block:

→ If very few lines of code requires synchronization, then it is never recommanded to declare entire method as **synchronized**. We have to declare those few lines of code inside synchronized block.

→ The main advantage of synchronized block over synchronized method is it reduces the waiting of the Threads and improves performance of the system.

Ex1:

→ We can declare synchronized block to get current object lock..

```
synchronized(this) {
}
```

→ If Thread got lock of current object, then only it is allowed to execute this block.

Ex2:

→ To get lock of a particular object b we can declare synchronized block as follows:

```
synchronized(b) {
}
```

→ If Thread got lock of "b" then only it is allowed to execute that blocks.

***Ex3:**

→ To get class level lock we can declare synchronized block as follows:

```
synchronized(classname.class) {
}
```

→ If Thread got class level lock of **classname.class**, then only it is allowed to execute that block.

Ex4:

→ **synchronized block** concept is applicable only for **objects** and **classes**, but not for primitives, otherwise we will get CE:unexpected type

```
class SyncUnexpectedTypeTest{
    public static void main(String args[]) {
        int x = 10;
        synchronized (x) {
        }
    }
}
```

```
javac  SyncUnexpectedTypeTest
CE:    unexpected type
       found   : int
       required: reference
            synchronized (x) {
            ^
```

→ Every object in java has a unique Lock, but a Thread can acquire more than one lock at time(from different objects)

```
class x {
        synchronized m1() {
                Y y = new Y();
                y.m2();
        }
}

class Y {
        synchronized m2() {
        }
}
```

Inter Thread communication:

→ Two Threads will communicate with each other by using **wait()**, **notify()**, **notifyAll()** methods. The Thread which requires updation it has to call wait() method. The Thread which is responsible to update it has to call notify() method.

→ **wait()**, **notify()**, **notifyAll()** methods are available on **Object** class but **not** in **Thread** class, Because Threads are required to call these method on any shared object.

→ If a wants to call **wait()**, **notify()**, **notifyAll()** methods compulsory the **Thread** should be owner of the **Object** i.e., the Thread has to get lock of that object i.e, the Thread should be in the **synchronized** area.

→ Hence, we can call **wait()**, **notify()**, **notifyAll()** methods only from **synchronized** area, otherwise we will get **RE:IllegalMonitorStateException**.

→ If a Thread calls **wait()** method it releases the lock immediately and entered into waiting state. Thread releases the lock of only current object but not all locks. After calling **notify()** and **notifyAll()** method Thread releases the lock but may not immediately. Except these **wait()**, **notify()**, **notifyAll()** there is no other case where Thread releases the lock.

```
Syntax: public final void wait() throws InterruptedException
Syntax: public final void wait(long ms) throws InterruptedException
Syntax: public final void wait(long ms, int ns) throws InterruptedException
Syntax: public final native void notify();
Syntax: public final native void notifyAll();
```

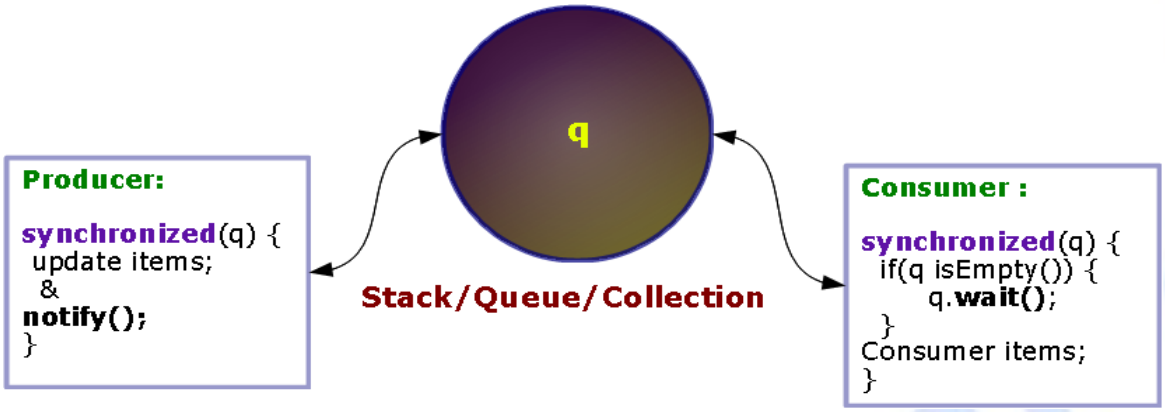
- **wait()** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify().
- **notify()** wakes up the first thread that called wait() on the same object.
- **notifyAll()** wakes up all the threads that called.

Method	Is Thread releases lock?
yield()	No

join()	No
sleep()	No
wait()	Yes
notify(()	Yes
notifyAll()	Yes

Ex: wait() & notify()

```
class ThreadA extends Thread{
	public static void main(String args[]) throws InterruptedException {
		ThreadB b = new ThreadB();
		b.start();
		//Thread.sleep(500);
		synchronized (b) {
		System.out.println("..Main Thread trying to call wait()...");
			b.wait();// b.wait(1000);
			System.out.println("..Main Thread got notification...");
			System.out.println(".ThreadB--> b.total = "+b.total);
		}
	}
}

class ThreadB extends ThreadA {
	int total = 0;

	public void run() {
		synchronized (this) {
		System.out.println("..Child Thread starts notification...");
			for (int i = 1; i < 5; i++) {
				total += i;
			}
		System.out.println("..Child Thread trying to given notification...");
			this.notify();
		}
	}
}

/*
Output:
	java ThreadA
	..Main Thread trying to call wait()...
	..Child Thread starts notification...
	..Child Thread trying to given notification...
	..Main Thread got notification...
	.ThreadB--> b.total = 10
```



*/

Ex:

→ Consumer has to consume items from the Queue.

→ If Queue is Empty, he has to call **wait()** method.

→ Producer has to produce items into the Queue.

→ After producing the items, he has to call **notify()** method, so that all waiting consumers will get notification.

Program:

```
// A correct implementation of a producer and consumer.
class Q {
        int n;
        boolean valueSet = false;

        synchronized int get() {
                if (!valueSet)
                        try {
                                wait();
                        } catch (InterruptedException e) {
                                System.out.println("InterruptedException caught");
                        }
                System.out.println("Consume: " + n);
                valueSet = false;
                notify();
                return n;
        }

        synchronized void put(int n) {
                if (valueSet)
                        try {
                                wait();
                        } catch (InterruptedException e) {
                                System.out.println("InterruptedException caught");
                        }
                this.n = n;
                valueSet = true;
```

```
                System.out.println("Produce: " + n);
                notify();
        }
}

class Producer implements Runnable {
        Q q;

        Producer(Q q) {
                this.q = q;
                new Thread(this, "Producer").start();
        }

        public void run() {
                int i = 0;
                while (true) {
                        q.put(i++);
                }
        }
}

class Consumer implements Runnable {
        Q q;

        Consumer(Q q) {
                this.q = q;
                new Thread(this, "Consumer").start();
        }

        public void run() {
                while (true) {
                        q.get();
                }
        }
}

class PCFixed {
        public static void main(String args[]) {
                Q q = new Q();
                new Producer(q);
                new Consumer(q);
                System.out.println("Press Control-C to stop.");
        }
}
/*Output:
java PCFixed
Produce: 0
Press Control-C to stop.
Consume: 0
Produce: 1
Consume: 1
Produce: 2
Consume: 2
Produce: 3
```

```
Consume: 3
Produce: 4
Consume: 4
Produce: 5
Consume: 5
Produce: 6
Consume: 6
Produce: 7
Consume: 7
Produce: 8
Consume: 8
.
.
.
*/
```

notify() & notifyAll(): We can use **notify()** only one waiting Thread, but which waiting Thread will be notified we can't exactly. All remaining Threads have to wait for futher notifications.

→ But in the case of **notifyAll()** all waiting Threads will be notifing, but these will be executed one by one.

*** **Note:** On which object we are calling **wait(), notify() & notifyAll()**, we have to get the lock of that obejct.

```
Stack s1 = new Stack();
Stack s2 = new Stack();
```

Correct:

```
synchronized(s1) {
  s1.wait();
}
```

RE:

```
synchronized(s1) {
s2.wait();
}
RE:IllegalMonitorStateException
```

DaemonThread: The Threads which are executing in the background are called "**Daemon Threads**"
Ex: GC.

→ The main objective of Daemon Threads is to provide support for non-Daemon Threads.
We can check whether the Thread is Daemon or not by using "**isDaemon()**" method.

```
public final boolean isDaemon()
```

→ We can change Daemon nature of a Thread by using **setDaemon()** method.

public final void setDaemon(boolean b)

We can change Daemon nature of a Thread before starting only, if we are trying to change after starting a Thread we will get RE, saying "IllegalThreadStateException".

→ ***main** Thread* is always ***non-Daemon*** & it's ***not possible** to change it's **Daemon nature**.*

Default nature: By default main Thread is always non-daemon, but for all the remaining Threads Daemon nature will be inheriting from parent to child. i.e., if the parent is Daemon, child is also Daemon & if the parent is non-Daemon then child is also non-Daemon.

→ Whenever the last non-Daemon Thread terminates all the Daemon Threads will be terminated automatically.

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
  * @fileName:DaemonThreadTest.java
 */
class MyThread1 extends Thread {
	public void run() {
		for (int i = 1; i < 5; i++) {
			System.out.println("Lazy Thread");

			try {
				Thread.sleep(2000);
			} catch (InterruptedException e) {
			}
		}
	}
}

class DaemonThreadTest {
	public static void main(String[] args) throws InterruptedException {
		MyThread1 t = new MyThread1();
		t.setDaemon(true);
		t.start();
		System.out.println("End of Main");
	}
}
/*
OUTPUT:

java DaemonThreadTest
End of Main
Lazy Thread
*/
```

"**t.setDaemon(true);**" if we are commenting this line in the program then both Main and Child Threads are non-Daemon and hence, both will be executed until their completion

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
  * @fileName:DaemonThreadTest.java
 */

class MyThread2 extends Thread {
	public void run() {
		for (int i = 1; i < 5; i++) {
			System.out.println("Lazy Thread");

			try {
				Thread.sleep(2000);
			} catch (InterruptedException e) {
			}
		}
	}
}

class DaemonThreadTest{
	public static void main(String[] args) throws InterruptedException {
		MyThread2 t = new MyThread2();
		// t.setDaemon(true);
		t.start();
		System.out.println("End of Main");
	}
}

/*
OUTPUT:

java DaemonThreadTest
End of Main
Lazy Thread
Lazy Thread
Lazy Thread
Lazy Thread
*/
```

DeadLock: If two Threads are waiting for each other for ever such type of situation is called "Dead-Lock".

→ There are no resolution Technique for Deadlock, but several prevention techniques are possible.

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
  * @fileName:DeadLock.java
 */

class A {
	public synchronized void foo(B b) {
		System.out.println("Thread:: Starts execution foo()");
		try {
			Thread.sleep(1000);
		} catch (InterruptedException e) {
		}
		System.out.println("Thread:::Trying to call b's last()");
```

```
                b.last();
        }

        public synchronized void last() {
                System.out.println("Inside A this last()");
        }
}

class B {
        public synchronized void bar(A a) {
                System.out.println("Thread:: Starts execution bar()");
                try {
                        Thread.sleep(3000);
                } catch (InterruptedException e) {
                }
                System.out.println("Thread:::Trying to call a's last()");
                a.last();
        }

        public synchronized void last() {
                System.out.println("Inside B this last()");
        }
}

class DeadLock extends Thread {
        A a = new A();
        B b = new B();

        DeadLock() {
                this.start();
                a.foo(b);// executed by Main Thread
        }

        public void run() {
                b.bar(a); // executed by child Thread
        }

        public static void main(String[] args) throws InterruptedException {
                new DeadLock();
        }
}
/*
OUTPUT:
java DeadLock
Thread:: Starts execution foo()
Thread:: Starts execution bar()
Thread:::Trying to call b's last()
Thread:::Trying to call a's last()
-
-
Waiting -->DeadLock
*/
```

*** "**synchronized**" keyword is the only one reason for **Deadlock**, hence, while using this keyword we have to take very much care.

Deadlock vs Starvation:

→ In the case of **Deadlock** *waiting never ends*.

→ A ***long waiting*** of a **Thread** which ends at **certain point of time** is called "**Starvation**".

→ Hence, **a long waiting which never ends** is called "**Deadlock**". Where as a long waiting which ends at certain point of time is called "**Starvation**".

Ex: Least priority Thread has to wait until completing all the Threads, but this long waiting should compulsory ends at certain point of time.

stop(): A Thread can kill or stop another Thread by using stop() method. Then the running Thread will entered into Dead state. It is a deprecated method and hence, not recommended to use.

```
public void stop();
```

suspend() & resume()

→ A Thread can suspend another Thread by using **suspend()** method.

→ A Thread can resume a suspended Thread by using **resume()** method.

→ But, these methods are deprecated methods , hence, not recommended.

```
final void suspend();
final void resume();
```

Ex:

```
// Using suspend() and resume().
class NewThread implements Runnable {
        String name; // name of thread
        Thread t;

        NewThread(String threadname) {
                name = threadname;
                t = new Thread(this, name);
                System.out.println("New thread: " + t);
                t.start(); // Start the thread
        }

        // This is the entry point for thread.
        public void run() {
                try {
                        for (int i = 5; i > 0; i--) {
                                System.out.println(name + ": " + i);
                                Thread.sleep(200);
                        }
                } catch (InterruptedException e) {
                        System.out.println(name + " interrupted.");
                }
                System.out.println(name + " exiting.");
        }
}
```

```
class SuspendResume {
	public static void main(String args[]) {
		NewThread ob1 = new NewThread("One");
		NewThread ob2 = new NewThread("Two");
		try {
			Thread.sleep(1000);
			ob1.t.suspend();
			System.out.println("Suspending thread One");
			Thread.sleep(1000);
			ob1.t.resume();
			System.out.println("Resuming thread One");
			ob2.t.suspend();
			System.out.println("Suspending thread Two");
			Thread.sleep(1000);
			ob2.t.resume();
			System.out.println("Resuming thread Two");
		} catch (InterruptedException e) {
			System.out.println("Main thread Interrupted");
		}
		// wait for threads to finish
		try {
			System.out.println("Waiting for threads to finish.");
			ob1.t.join();
			ob2.t.join();
		} catch (InterruptedException e) {
			System.out.println("Main thread Interrupted");
		}
		System.out.println("Main thread exiting.");
	}
}

/*
Output:
	New thread: Thread[One,5,main]
			New thread: Thread[Two,5,main]
			One: 5
			Two: 5
			One: 4
			Two: 4
			One: 3
			Two: 3
			One: 2
			Two: 2
			One: 1
			Two: 1
			Suspending thread One
			Two exiting.
			Resuming thread One
			One exiting.
			Suspending thread Two
			Resuming thread Two
			Waiting for threads to finish.
			Main thread exiting.
*/
```

Native Thread : Native threads use the operating system's native ability to manage multi-threaded processes - in particular, they use the thread library. When you run with native threads, the kernel schedules and manages the various threads that make up the process.

Green Thread :

Green threads are "user-level threads". They are scheduled by an "ordinary" user-level process, not by the kernel. So they can be used to simulate multi-threading on platforms that don't provide that capability. In the context of Java specifically, green threads are a thing of the past

Green threads emulate multi-threaded environments without relying on any native OS capabilities. They run code in user space that manages and schedules threads; Sun wrote green threads to enable Java to work in environments that do not have native thread support.

Thus, the advantage is that you get thread-like functionality at all. The disadvantage is that green threads can't actually use multiple cores.

There were a few early JVMs that used green threads (IIRC the Blackdown JVM port to Linux did), but nowadays all mainstream JVMs use real threads.

Performance

On a multi-core processor, native thread implementations can automatically assign work to multiple processors, whereas green thread implementations normally cannot. Green threads can be started much faster on some VMs .

When a green thread executes a blocking system call, not only is that thread blocked, but all of the threads within the process are blocked.

Exception Handling

Def: In general, exceptions are run-time errors caused due to logical mistakes occurred during program execution because of wrong input.

Exception is an object. It is an instance of one of the subclass of **java.lang.Throwable** class.

Different exception classes are available in **java.lang** package for representing different logical mistakes. All these classes are subclasses of **java.lang.Throwable class**.

Ex: 1) For handling wrong operation on arrays below exception classes are given.
NegativeArraySizeException
ArrayIndexOutOfBoundException
ArrayStoreException

2) If we divide an int numer with zero
ArithmeticException

3) If we try to convert alpha numeric string data to a number
NumberFormatException ...etc.,

When an exception is raised in the program, program execution is terminated abnormally.

When a logical mistake occurred in the program the JVM creates exception class object that is associated with that logical mistake, and terminates the current method execution by throwing this exception object using "**throw**" keyword. So that an exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions execution.

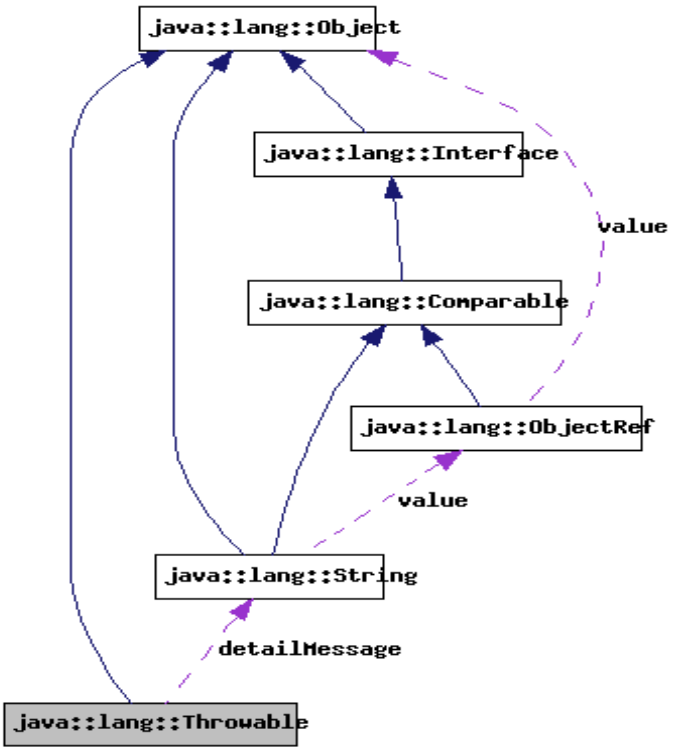
Ex:

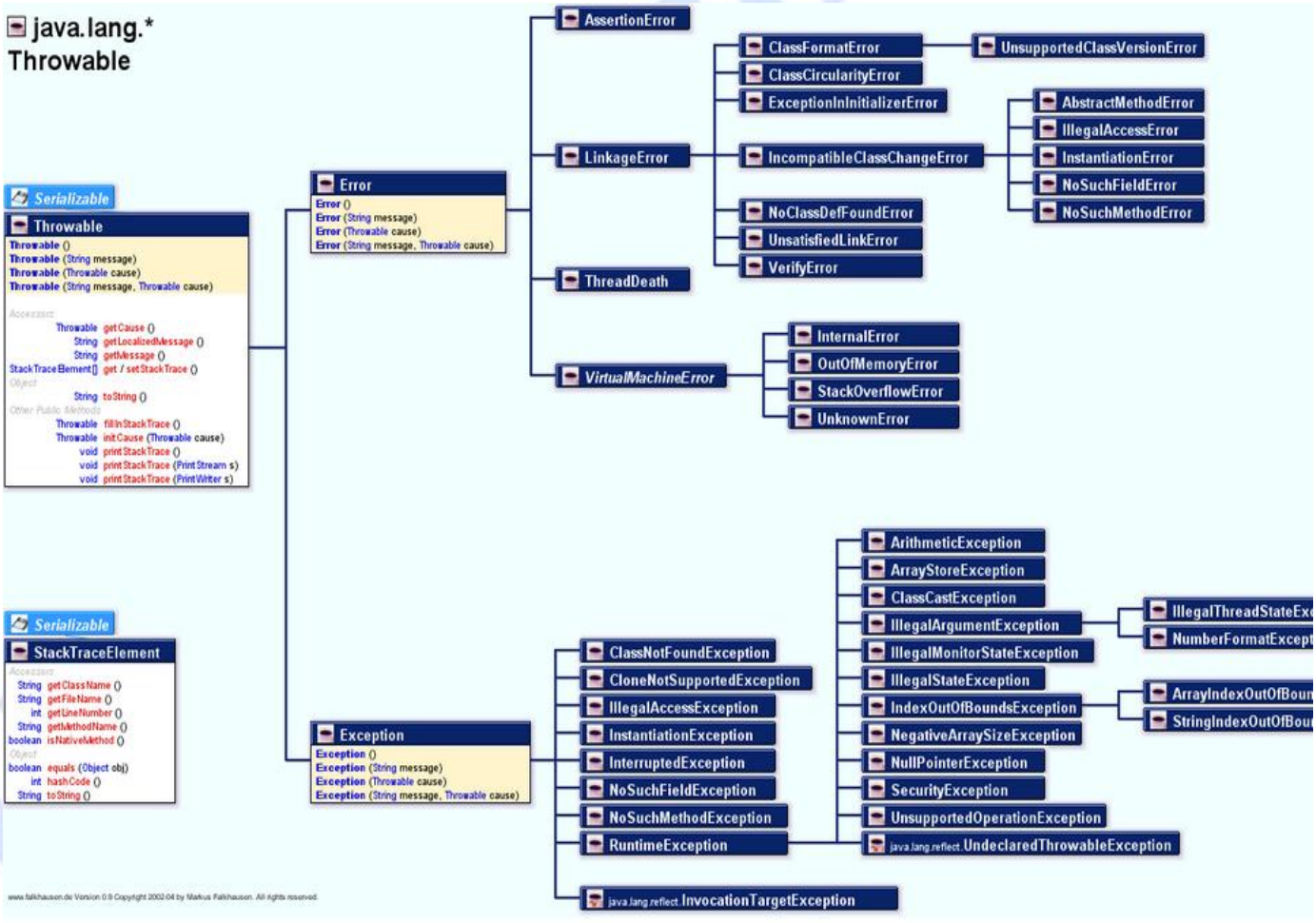
```
/**@author:VENU KUMAR.S  @date:Oct 7, 2012
 * @fileName:NormalExectution.java
 */
class NormalExectution {
	public static void main(String args[]) {
		int a = 10;
		int b = 5;

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

		int c = a / b;
		System.out.println("c = " + c);
	}
}
/*
 OUTPUT:
java NormalException
a = 10
b = 5
c = 2
*/
```

Program with Exception:

```
/**@author:VENU KUMAR.S  @date:Oct7, 2012
 * @fileName:NormalExectutionWithException.java
 */

class NormalExectutionWithException {
	public static void main(String args[]) {
		int a = 10;
		int b = 0;

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

		int c = a / b;
		System.out.println("c = " + c);
	}
}

/*
 OUTPUT:
java NormalException
a = 10
b = 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

```
        at Scjp.NormalExectutionWithException.main(NormalExectutionWithException.java:14)
*/
```


java::lang::Object
java::lang::Interface
value
java::lang::Comparable
java::lang::ObjectRef
value
java::lang::String
detailMessage
java::lang::Throwable


java.lang.*
Throwable
Serializable
Throwable
Throwable ()
Throwable (String message)
Throwable (Throwable cause)
Throwable (String message, Throwable cause)
Accessors
Throwable getCause ()
String getLocalizedMessage ()
String getMessage ()
StackTraceElement[] get / setStackTrace ()
Object
String toString ()
Other Public Methods
Throwable fillInStackTrace ()
Throwable initCause (Throwable cause)
void printStackTrace ()
void printStackTrace (PrintStream s)
void printStackTrace (PrintWriter s)
Error
Error ()
Error (String message)
Error (Throwable cause)
Error (String message, Throwable cause)
AssertionError
LinkageError
ClassFormatError
UnsupportedClassVersionError
ClassCircularityError
ExceptionInInitializerError
IncompatibleClassChangeError
AbstractMethodError
IllegalAccessError
InstantiationError
NoSuchFieldError
NoSuchMethodError
NoClassDefFoundError
UnsatisfiedLinkError
VerifyError
ThreadDeath
VirtualMachineError
InternalError
OutOfMemoryError
StackOverflowError
UnknownError
Serializable
StackTraceElement
Accessors
String getClassName ()
String getFileName ()
int getLineNumber ()
String getMethodName ()
boolean isNativeMethod ()
Object
boolean equals (Object obj)
int hashCode ()
String toString ()
Exception
Exception ()
Exception (String message)
Exception (Throwable cause)
Exception (String message, Throwable cause)
ClassNotFoundException
CloneNotSupportedException
IllegalAccessException
InstantiationException
InterruptedException
NoSuchFieldException
NoSuchMethodException
RuntimeException
ArithmeticException
ArrayStoreException
ClassCastException
IllegalArgumentException
IllegalMonitorStateException
IllegalStateException
IndexOutOfBoundsException
NegativeArraySizeException
NullPointerException
SecurityException
UnsupportedOperationException
java.lang.reflect.UndeclaredThrowableException
java.lang.reflect.InvocationTargetException
NumberFormatExcept
StringIndexOutOfBou
www.falkhausen.de Version 0.9 Copyright 2002-04 by Markus Falkhausen. All rights reserved.

Error & Exception:

1) **Error type exceptions** are thrown due to the problem occurred in **JVM** side logic, like:

→ if there is no memory in JSA to create new stack frame to execute method, then JVM process is killed by throwing Error type exception "**java.lang.StackOverflowError**"

→ if there is no memory in HA to create new object, then JVM process is killed by throwing Error type exception "**java.lang.OutOfMemoryError**".

2) **Exception type exceptions** are thrown due to the problem occurred in java program logic execution like:
if we divide an integer by ZERO, then JVM terminates program execution by throwing Exception type exception "**java.lang.ArithmeticException**".

3) We **cannot catch "Error"** type exception, because **Error** type exception is not thrown in out of application, and once this **Error** type exception is thrown JVM is terminated.

We can **catch Exception** type exceptions, because **Exception** type exception is **thrown** in our application, and more over **JVM** is not directly terminated because of Exception type exceptions, JVM terminated on if the thrown exception is not caught.

Two types of Exception subclasses:

1)Subclasses of RuntimeException: Represents logical mistakes occurred due to operator execution failure. i.e., these exceptions are prepared and thrown by JVM at run-time when operator execution is failed

2)Direct subclasses of Exception: Represents logical mistakes occurred due to condition failure because of wrong input. These exceptions are prepared and thrown by the developer by using "throw" keyword.

Two categories of Exceptions:

1) Checked exceptions: Throwable, **Exception** and its **direct subclasses** are called checked exceptions because if they are thrown by using "**throw**" keyword compiler checks their handling and if they are not caught by using "**try/catch**" or not reported by using "**throws**" keyword, compiler throws.

2) Unchecked exceptions: Error, **RuntimeException** and **their subclasses** are called unchecked exception because these exceptions handling is not checked by compiler when they are thrown by using **throw** keyword i.e., these exception objects catching or reporting is optional.

EXCEPTIONS	DESCRIPTION	CHECKED	UNCHECKED
ArithmeticException	Arithmetic errors such as a divide by zero	-	YES
ArrayIndexOutOfBoundsException	Arrays index is not within array.length	-	YES
ClassNotFoundException	Related Class not found	YES	-
IOException	InputOuput field not found	YES	-
IllegalArgumentException	Illegal argument when calling a method	-	YES
InterruptedException	One thread has been interrupted by another thread	YES	-
NoSuchMethodException	Nonexistent method	YES	-
NullPointerException	Invalid use of null reference	-	YES
NumberFormatException	Invalid string for conversion to number	-	YES

When an exception is thrown by developer using throw keyword, if that exception handling is checked by compiler then that exception is called checked exception, else that is called unchecked exception.

→ **try/catch/finally:** used for catching the raised exception and for executing statements definitely for a try block.

→ **throw/throws:** for throwing an exception manually from a method and for reporting the thrown exception to other programmer.

→ **throw:** This keyword is used to **throw** an **exception manually**. In most of the cases we use it from throwing checked exceptions explicitly.

→ **throw** keyword must follow **Throwable** type of object.
→ it must be used only in method logic.

→ **Rule:** since it is a transfer statement, we cannot place statements after **throw** statement if leads to **CE: "unreachable statement"**.

```
ex: void m1() {
        throw new ArithmeticException();
    }
```

ArithmeticException is run-time exception, so compiler does not check its exception handling.

→ **throws:** This keyword is used to report that **raised exception to the caller**. It is mandatory for

checked exceptions for reporting, if they are not handled.

→ **throws** keyword must follow **Throwable** type of class name.
→ it must be used in method prototype after method parenthesis.

```
ex: void m1(){
        throw new InterruptedException();
    }
```

InterruptedException is a subclass of Exception, so we must catch or report this exception using throws keyword. Since we have not done either of both, compiler throws CE:"unreported exception InterruptedException must be caught or declared to thrown".

Correct syntax of throwing checked exception

```
void m1() throws InterruptedException {
    throw new InterruptedException();
}


void m1() throws ArithmeticException {
    throw new ArithmeticException();
}
```

→ **Rule:** we are not allowed to write **catch** block with checked exception without throwing it from the **try** block, it leads to **CE:"exception never thrown from the corresponding try statement**". The following ex leads to this **CE**

```
void m1() {
    try{
    }
    catch(InterruptedException e){
       System.out.println(e);
    }
}
```

The following program compiles fine:

```
void m1() {
    try{
        throw new InterruptedException();
    }
    catch(InterruptedException e){
       System.out.println(e);
    }

    void m1() throws InterruptedException{
    try{
        throw new InterruptedException();
    }
    catch(InterruptedException e){
       System.out.println(e);
    }
}
```

throws: In our program, if there is any chance of raising **checkedException** compulsory we should handle it, otherwise we will get **CE:** "unreported exception xxxx; must be caught or declare to be thrown".

```
class Test {
        public static void main(String args[]) {
                Thread.sleep(500);
        }
}
```

CE:unreported exception java.lang.InterruptedException; must be caught or declare to be thrown

→ we can handle this by using 1) **try-catch** 2) **throws**

1) using **try-catch**

```
class Test {
        public static void main(String args[]) {
                try {
                        Thread.sleep(500);
                } catch (InterruptedException ie) {
                        System.out.println(ie);
                }
        }
}
```

2) using **throws :**→ we can use throws keyword to delegate the responsibility of exception handling to the caller methods in the case of checked exception, to convince compiler.

```
class Test {
        public static void main(String args[]) throws InterruptedException {
                        Thread.sleep(500);
        }
}
```

→ We can use throws only for throwable types , otherwise we will get CE:incompatable types.

Exception Handling Procedure: 4 steps

→ **step 1) Preparing** exception object appropriate to the current logical mistake.
→ **step 2) Throwing** that exception to the appropriate exception handler.
→ **step 3) Catching** that exception.
→ **step 4) Taking** necessary actions against that exception.

→ We can handle exceptions in java using **1) try 2) catch** blocks.

Exception handling procedure:

→ The first two actions are done by using **try** block and
→ the next two actions are done by using **catch** block.

try: This keyword establishes a block to write a code that causes exceptions and its related statements.
Exception causing statement must be placed in try block to handle and catch exception for stopping abnormal terminations and to display end-user understandable messages.

catch:

→ catch block is used to catch exceptions those are thrown from its corresponding try block. It has logic to take necessary actions on that caught exception.

→ catch block looks like constructor syntax. It does not take accessibility modifiers, normal modifiers, return type. It takes only single parameter of type **Throwable** or its sub-classes.

→ **Throwable** is the super class of all exception classes.

→ Inside catch we can write any statement which is legal in java, including raising an Exception.

Rules:

1) try must follow either one or 'n' number of catch blocks or '1' finally block else it leads to **CE:"try without catch or finally"**.

```
try {
}
```

2)catch must be placed immediately after try block else it leads to **CE:"catch without try"**.

```
try {
} finally {
} catch (Exception e) {
}
```

3)finally must be placed either immediately after try or after try-catch else it leads to **CE: "finally without try"**

4) The catch block parameter must be type of "**java.lang.Throwable**" or its sub-classes else it leads to **CE:"incompatible types**".

```
try {
} catch (String s) {
}
```

5) **try-catch-finally** not allowed at class level directly, because logic is not available at class level.

```
class A {
    try {
    } catch (Exception e) {
    } finally {
    }
}
```

Exception Propagation:

The process of sending exception from called method to calling method is called exception propagation. If an exception is propagated and if that exception is not caught in that calling method, not only called method execution but also calling method execution is terminated abnormally.

```
/**@author:Venu Kumar.S  @date:Oct 7, 2012
 * @fileName:Example.java
 */

class Example {
	public static void main(String args[]) {
		System.out.println("In main method start..");

		try {
			m1();
		} catch (ArithmeticException ae) {
			System.out.println("In main Catch ...." + ae);
		}
		System.out.println("In main method end...");
	}

	static void m1() {
		System.out.println("...In m1 method start");
		System.out.println(10 / 0);
		System.out.println("...In m1 method End");
	}

}
/* OUTPUT: java Example
In main method start..
...In m1 method start
In main Catch ....java.lang.ArithmeticException: / by zero
In main method end...
*/
```

finally Block: Finally establishes a block that definitely executes statements places in it. Statements which are placed in finally block are always executed irrespective of the way the control is coming out from the try block either by completing normally or by return statement or by throwing exception by catching or not catching.

Need of finally block: in this block we should write resource releasing logic or clean up code. Resource releasing logic means unreferencing objects those are created in try block.

→ **syntax:** 1) **try{}**
catch(exception obj){}
finally{}

Program1:

```
/**@author:Venu Kumar.S  @date:Oct 07, 2012
 * @fileName:Tcf.java
 */
class Tcf {
	public static void main(String args[]) {
		try {
			System.out.println("In try Block");
			System.out.println(5 / 0);
```

```
		} catch (ArithmeticException ae) {
			System.out.println("In catch Block");
			System.out.println(ae);
		} finally {
			System.out.println("In finally block");
		}
		System.out.println("After try/catch/finally");
	}
}

/*
 OUTPUT: javac Tcf

In catch Block
java.lang.ArithmeticException: / by zero
In finally block
After try/catch/finally

*/
```

Program2:

```
/**@author:Venu Kumar.S  @date:Oct 07, 2012
 * @fileName:Tcf.java
 */
class Tcf {
	public static void main(String args[]) {
		try {
			System.out.println("In try Block");
			System.out.println(5 / 0);

		} catch (NullPointerException npe) {
			System.out.println("In catch Block");
			System.out.println(npe);
		} finally {
			System.out.println("In finally block");
		}
		System.out.println("After try/catch/finally");
	}
}

/*
 OUTPUT: javac Tcf

In try Block
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at JAVATest.Tcf.main(x.java:15)
In finally block

*/
```

Program3:

```
/**@author:Venu Kumar.S  @date:Oct 07, 2012
 * @fileName:Tcf.java
 */
```

```
class Tcf {
	public static void main(String args[]) {
		try {
			System.out.println("In try Block");

		} catch (Exception e) {
			System.out.println("In catch block");
		} finally {
			System.out.println("In finally block");
		}
		System.out.println("After try/catch/finally");
	}
}

/*
 OUTPUT: javac Tcf

In try Block
In finally block
After try/finally

*/
```

Program4:

```
/**@author:Venu Kumar.S  @date:Oct 07, 2012
* @fileName:Tcf.java
*/
class Tcf {
	public static void main(String args[]) {
		System.out.println(m1());
	}

	static int m1() {
		try {
			System.out.println("In try Block");
			return 10;

		} catch (NullPointerException npe) {
			System.out.println("In catch Block");
			return 20;
		} finally {
			System.out.println("In finally block");
			return 30;
		}
	}
}

/*
 OUTPUT: javac Tcf

In try Block
In finally block
30
```

```
*/
```

→ **Unreachable Code:**If we place return statement in finally block, we can't place a statement after finally block, it leads to CE:Unreachable Statement.

Program5:

```
/**@author:Venu Kumar.S  @date:Oct 07, 2012
 * @fileName:Tcf.java
*/
class Tcf {
	public static void main(String args[]) {
		System.out.println(m1());
	}

	static int m1() {
		try {
			System.out.println("In try Block");
			return 10;

		} catch (NullPointerException npe) {
			System.out.println("In catch Block");
			return 20;
		} finally {
			System.out.println("In finally block");
			return 30;
		}
	System.out.println("After tyr/catch/finally ");//CE:Unreachable Statement
	}
}
```

→ **Syntax: 2)** **try{}**
finally{}

Program4:

```
/**@author:Venu Kumar.S  @date:Oct 07, 2012
 * @fileName:Tcf.java
 */

class Tcf {
	public static void main(String args[]) {
		try {
			System.out.println("In try Block");

		}  finally {
			System.out.println("In finally block");
		}
		System.out.println("After try/finally");
	}
}

/*
 OUTPUT: javac Tcf
```

```
In try Block
In finally block
After try/finally

*/
```

Program5:

```
/**@author:Venu Kumar.S  @date:Oct 07, 2012
 * @fileName:Tcf.java
 */
class Tcf {
	public static void main(String args[]) {
		try {
			System.out.println("In try Block");
			System.out.println(5 / 0);

		}  finally {
			System.out.println("In finally block");
		}
		System.out.println("After try/catch/finally");
	}
}

/*
 OUTPUT: javac Tcf

In try Block
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at JAVATest.Tcf.main(x.java:15)
In finally block
*/
```

Inner finally block: We can also write finally for inner try block that finally block is called inner finally.

1) No Exception is raised in inner or outer try blocks

```
/**@author:Venu Kumar.S  @date:Oct 07, 2012
 * @fileName:Tcf.java
 */
class Tcf {
	public static void main(String args[]) {
		m1();
	}

	static void m1() {
		try {
			System.out.println("In Outer try Block");
			try {
				System.out.println("...In Inner try Block");

			} catch (NullPointerException npe) {
				System.out.println("...In Inner catch Block");
```

```
                } finally {
                        System.out.println("...In Inner finally block");
                }
            System.out.println("After Inner try/catch/finally ");

            } catch (NullPointerException npe) {
                System.out.println("In Outer catch Block");

            } finally {
                System.out.println("In Outer finally block");

            }
            System.out.println("After Outer try/catch/finally ");
        }
}

/*
 OUTPUT: javac Tcf

In Outer try Block
...In Inner try Block
...In Inner finally block
...After Inner try/catch/finally
In Outer finally block
After Outer try/catch/finally
*/
```

2) return statement is placed in inner finally

```
/**@author:Venu Kumar.S  @date:Oct 07, 2012
 * @fileName:Tcf.java
 */
class Tcf {
        public static void main(String args[]) {
            System.out.println(m1());
        }

        static int m1() {
            try {
                System.out.println("In Outer try Block");
                try {
                    System.out.println("...In Inner try Block");
                } catch (NullPointerException npe) {
                    System.out.println("...In Inner catch Block");
                } finally {
                    System.out.println("...In Inner finally block");
                    return 10;
                }
            //System.out.println("After Inner try/catch/finally "); //CE:URS
            } catch (NullPointerException npe) {
                System.out.println("In Outer catch Block");
            } finally {
                System.out.println("In Outer finally block");
            }
```

```
                System.out.println("After Outer try/catch/finally ");
                return 30;
        }
}

/*
 OUTPUT: javac Tcf

In Outer try Block
...In Inner try Block
...In Inner finally block
In Outer finally block
10
*/

/**@author:Venu Kumar.S  @date:Oct 07, 2012
 * @fileName:Tcf.java
 */
class Tcf {
        public static void main(String args[]) {
                System.out.println(m1());
        }

        static int m1() {
                try {
                        System.out.println("In Outer try Block");
                        try {
                                System.out.println("...In Inner try Block");
                        } catch (NullPointerException npe) {
                                System.out.println("...In Inner catch Block");
                        } finally {
                                System.out.println("...In Inner finally block");
                        }

                } catch (NullPointerException npe) {
                        System.out.println("In Outer catch Block");
                } finally {
                        System.out.println("In Outer finally block");
                }
                System.out.println("After Outer try/catch/finally ");
                return 30;
        }
}

/*
 OUTPUT: javac Tcf

In Outer try Block
...In Inner try Block
...In Inner finally block
In Outer finally block
After Outer try/catch/finally
30
*/
```

return vs finally:

→ ***finally*** block dominates ***return*** statement also. Hence, if there is any **return** statement present inside **try** or **catch** block, first **finally** will be executed and then **return** statement will be considered.

Ex:

```
class Test {
	public static void main(String args[]) {
		try {
			System.out.println("try block..");
			return;
		} catch (Exception e) {
			System.out.println("catch block..");
		} finally {
			System.out.println("finally block..");
		}

	}
}

/*Output:
java Test
try block..
finally block..
*/
```

→ ***There is only one situation where the finally block won't be executed is, whenever JVM shut-down. i.e., whenever we are using **System.exit(0)**.

```
class Test {
	public static void main(String args[]) {
		try {
			System.out.println("try block..");
			System.exit(0);
		} catch (Exception e) {
			System.out.println("catch block..");
		} finally {
			System.out.println("finally block..");
		}

	}
}

/*Output:
java Test
try block..
*/
```

Some Common Exceptions: Based on the source, who triggers the exception, all exceptions are divided into 2 types. 1) **JVM Exceptions** 2) **Programmatic Exceptions**

1) **JVM Exceptions:** The Exceptions which are raised automatically by the JVM whenever a particular Event occurs are called JVM Exceptions.

Ex: ArrayIndexOutofBoundsException
NullPointerException
StackOverflowError
NoClassDefFoundException
classCastException
ExceptionInInitializerError

2) **Programmatic Exceptions:** The Exceptions which are raised explicitly either by the programmer or by the API developer are called programmatic Exception.

Ex: IllegalArgumentException
NumberFormatException
IllegalStateException
AssertionError

ArrayIndexOutofBoundsException: It is the child class of RuntimeException and hence it is un-checked. Raised automatically by the the JVM, whenever we are trying to access array element with, out of range index.

```
class ArrayIndexOutofBoundsExceptionTest {
	public static void main(String args[]) {
		int[] a=new int[2];
		System.out.println(a[0]);//0
		System.out.println(a[5]); //RE:java.lang.ArrayIndexOutOfBoundsException:
	}
}

/*Output:
0
java.lang.ArrayIndexOutOfBoundsException:
*/
```

NullPointerException: It is the child class of RuntimeException and hence it is unchecked. Raised automatically by the the JVM, whenever we are trying to perform any operation on null.

```
class NullPointerExceptionTest {
	public static void main(String args[]) {
		String s = null;
		System.out.println(s.length()); //RE:java.lang.NullPointerException
	}
}

/*Output:
java.lang.NullPointerException
*/
```

StackOverflowError: It is the child class of RuntimeException and hence it is unchecked. Raised automatically by the the JVM, whenever we are trying to perform recursive method invocation.

```
class StackOverflowErrorTest {
	public static void main(String args[]) {
		m1();
```

```
        }

        public static void m1() {
            m2();
        }

        public static void m2() {
            m1();
        }

    }
    /*Output:
    java StackOverflowErrorTest
    java.lang.StackOverflowError
    */
```

NoClassDefFoundException: It is the child class of RuntimeException and hence it is unchecked. Raised automatically by the the JVM, whenever JVM unable to find required class.

java Test400
→ If Test400 file not available then we will get RE:java.lang.NoClassDefFoundError:Test400

java.lang.NoClassDefFoundError: Test400

classCastException: It is the child class of RuntimeException and hence it is unchecked. Raised automatically by the the JVM, whenever we are trying to typecast parent object to the child type.

```
    class classCastExceptionTest {
        public static void main(String args[]) {
            String s = new String("guru");
            Object o = (Object) s;
            System.out.println(s);
            System.out.println(o);

        }

    }
    /*Output:
    java classCastExceptionTest
    guru
    guru
    */

    class classCastExceptionTest {
        public static void main(String args[]) {
            Object o = new Object();
            String s = (String) o; //RE:java.lang.ClassCastException: java.lang.Object cannot
                                   //  be cast to java.lang.String
            System.out.println(s + o);
        }

    }
    /*Output:
```

```
java classCastExceptionTest
java.lang.ClassCastException: java.lang.Object cannot be cast to java.lang.String
*/
```

ExceptionInInitializerError: It is the child class of RuntimeException and hence it is unchecked. Raised automatically by the the JVM,if any Exception occurs while performing initialization for static va-triables and while executing static blocks.

```
class ExceptionInInitializerErrorTest {
        static int i=10/0; //RE:java.lang.ExceptionInInitializerError
                           //Caused by: java.lang.ArithmeticException: / by zero

        public static void main(String args[]) {
                System.out.println(i);
        }

}
/*Output:
java ExceptionInInitializerErrorTest
java.lang.ExceptionInInitializerError
Caused by: java.lang.ArithmeticException: / by zero
*/


class ExceptionInInitializerErrorTest {
        static{
                String s = null;
                System.out.println(s.length()); //RE:java.lang.ExceptionInInitializerError
                                                //Caused by: java.lang.NullPointerException
        }
        public static void main(String args[]) {

        }
}
/*
 java ExceptionInInitializerErrorTest

java.lang.ExceptionInInitializerError
Caused by: java.lang.NullPointerException
*/
```

IllegalArgumentException: It is the child class of RuntimeException and hence it is unchecked. Raised explicitly by the programmer or by API developer to indicate that a method has been invoked with invalid argument.

```
class IllegalArgumentExceptionTest {

        public static void main(String args[]) {
         Thread t=new Thread();
         t.setPriority(10);
         t.setPriority(50); //RE:java.lang.IllegalArgumentException

        }
```

```
}
/*
 java IllegalArgumentExceptionTest
java.lang.IllegalArgumentException
*/
```

NumberFormatException: It is the child class of RuntimeException and hence it is unchecked.
Raised explicitly by the programmer or by API developer to indicate that we are trying to convert String to Number type, but the String is not properly formatted.

```
class NumberFormatExceptionTest {

        public static void main(String args[]) {
        Integer i=new Integer("5");
Integer i1=new Integer("Five");//RE:java.lang.NumberFormatException: For input string: "Five"
        }
}
/*
 java NumberFormatExceptionTest

java.lang.NumberFormatException: For input string: "Five"
*/
```

IllegalStateException: It is the child class of RuntimeException and hence it is unchecked.
Raised explicitly by the programmer or by API deceloper to indicate that a method has been invoked at in appropriate time.

Ex1: After starting a Thread we are not allowed to restart the same Thread, otherwise we will get **RE:**java.lang.IllegalThreadStateException

```
class IllegalThreadStateExceptionTest {

        public static void main(String args[]) {
                Thread t = new Thread();
                t.start();
                t.start();//RE:java.lang.IllegalThreadStateException

        }
}
/*
 java IllegalThreadStateExceptionTest

java.lang.IllegalThreadStateException
java.lang.Thread.start
*/
```

Ex2: Once session expires we can't call any method on that object, otherwise, we will get IllegalStateException.

```
HttpSession session = req.getSession();
System.out.println(session.getId());// shows session Id

session.invalidate();
```

```
System.out.println(session.getId()); //RE:IllegalSessionStateException
```

AssertionError: It is the child class of RuntimeException and hence it is unchecked.
Raised explicitly by the programmer or by API developer to indicate that assert statement fails.

```
class AssertionErrorTest {
        static void assertTest() throws Error{

      if (true) throw new AssertionError(); //RE: java.lang.AssertionError
      System.out.println(".AssertionErrorTest.assertTest()");
   }
        public static void main(String args[]) {
                //java.lang.AssertionError
                 try {
                        assertTest();
              } catch (Exception e) {
                 System.out.println("exception");
              }
              System.out.println(".AssertionErrorTest.main().enddd");
        }
}
/*
 java IllegalThreadStateExceptionTest

Exception in thread "main" java.lang.AssertionError
        at JAVATest.AssertionErrorTest.assertTest()
        at JAVATest.AssertionErrorTest.main()
*/
```

User Defined Exceptions

→ A user defined exception is extended by Exception class.

→ Unconventional actions are dealt with user defined exceptions

→ Application demanded exceptions other than java.lang APIs are defined as user defined exceptions.

Ex: When the balance of an account is below zero after withdrawal, an exception can be raised like '**NegativeBalanceException**'

→ As every exception returns a message, a user defined exception should also return a message.

Developer can able to create their own **Exception Class** by **extending Exception** Class from **java.lang** API. And the **toString()** method should be **overridden** in the *user defined exception class* in order to display meaningful information about the exception. Or-else you should create a String parametrized constructor and pass this value to Exception class "super(s)".

The *throw statement* is used to signal the occurrence of the exception within a try block.

Syntax:

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

OR

```
class UserDefinedException extends Exception {
        // code
        public String toString() {
                return "Exception_Message";
        }
}
```

Ex:

Our **Exception Class** by **extending Exception** Class from **java.lang** API. Or-else you should create a String parametrized constructor and pass this value to Exception class by calling "**super(s)**".

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

class Input {
        void method() throws WrongInputException {
                throw new WrongInputException("Wrong input");
        }
}

class TestInput {
        public static void main(String[] args) {
                try {
                        new Input().method();
                } catch (WrongInputException wie) {

                        System.out.println(wie.getMessage());
                }
        }
}
/*
OUTPUT:
java TestInput
Wrong input
*/
```

Ex2:

MyOwnExceptionClass Class by extending **Exception** Class from **java.lang API**. And the **to-String()** method should be overridden in the *user defined exception class* in order to display meaning-ful information about the exception.

```
import java.io.*;

 class MyOwnExceptionClass extends Exception {

            private int price;

            public MyOwnExceptionClass(int price){
               this.price = price;
            }

            public String toString(){
               return "Price should not be in negative, you are entered" +price;
            }

          }

class Client {
        public static void main(String[] args)throws Exception
        {
          //int price = -120;
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
              System.out.println("Enter  Price..0 or +ve number : ");
              int price = Integer.parseInt(br.readLine());

            if(price < 0)
              throw new MyOwnExceptionClass(price);
            else
              System.out.println("Entered Price is :"+price);
            }
          }
```

Output1:

```
 java Client
Enter  Price..0 or +ve number :
5
Entered Price is :5
```

Output2:

```
 java Client
Enter  Price..0 or +ve number :
-1
Exception in thread "main" Price should not be in negative, you are entered-1
```

```
          import java.io.*;

          class MyOwnExceptionClass extends Exception {

                  private int price;
```

```
		public MyOwnExceptionClass(int price) {
			this.price = price;
		}

		public String toString() {
			return "Price should not be in negative, you are entered" + price;
		}

	}

	class Client1 {
		public static void main(String[] args) throws Exception {
			// int price = -120;
		BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
			System.out.println("Enter  Price..0 or +ve number : ");
			int price = Integer.parseInt(br.readLine());
			try {
				if (price < 0)
					throw new MyOwnExceptionClass(price);
				else
					System.out.println("Entered Price is :" + price);
			}

			catch (Exception e) {
				System.out.println(e);
			}
		}
	}
```

Output 1:

```
java Client1
	Enter  Price..0 or +ve number :
	1235
	Entered Price is :1235
```

Output 2:

```
java Client1
	Enter  Price..0 or +ve number :
	-12
	Price should not be in negative, you are entered -12
```

Internationalization -I18N

→ Various countries follow various conventions to represent dates, numbers, etc.

→ Our application should generate locate specific responses like for India people the response should be in terms of rupees(Rs.) & the US people the response should be in-terms of dollar($). The process of designing such type of web application is called "Internationalization (I18N)".

→ We can implement I18N using the following classes:

1) Locale

2) NumberFormat
3) DateFormat

1)**Locale:**

→ A Locale object represents a Geo-graphic location.

→ Locale class is the final class present in java.util

→ It is the direct child class of Object.

→ It implements Clonable, Serializable interfaces

Constructors:

Locale l = new Locale(String language);
Locale l = new Locale(String language, String country);

→ Locale class defines several constants to represent some standard Locales, we can use these Locale directly without creating our own.

Locale.US Locale.ENGLISH
Locale.ITALIAN Locale.UK

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
 * @fileName:LocaleDemo.java
 */
import java.util.*;

class LocaleDemo {
	public static void main(String[] args) throws InterruptedException {
		Locale l1 = Locale.getDefault();
		System.out.println(l1.getCountry() + "....." + l1.getLanguage());
		System.out.println(l1.getDisplayCountry() + "....."
				+ l1.getDisplayLanguage());

		/*
		 * OUTPUT: US.....en United States.....English
		 */

		Locale l2 = new Locale("pa", "iN");
		Locale.setDefault(l2);
		System.out.println(l2.getCountry() + "....." + l2.getLanguage());
		System.out.println(l2.getDisplayCountry() + "....."
				+ l2.getDisplayLanguage());

		/*
		 * OUTPUT:
		 *
		 * IN.....pa India.....Panjabi
		 */

		String[] s3 = Locale.getISOLanguages();
		for (String s4 : s3) {
			System.out.println(s4);
		}

		String[] s4 = Locale.getISOCountries();
		for (String s5 : s4) {
```

```
                System.out.println(s5);
        }

        Locale[] s = Locale.getAvailableLocales();
        for (Locale s1 : s) {
                System.out.println(s1.getDisplayCountry() + "-----"
                                + s1.getDisplayLanguage());
        }

/*
Japan-----Japanese
Peru-----Spanish
-----English
Japan-----Japanese
Panama-----Spanish
Bosnia and Herzegovina-----Serbian
-----Macedonian
Guatemala-----Spanish
United Arab Emirates-----Arabic
Norway-----Norwegian
Albania-----Albanian
-----Bulgarian
Iraq-----Arabic
Yemen-----Arabic
-----Hungarian
Portugal-----Portuguese
Cyprus-----Greek
Qatar-----Arabic
Macedonia-----Macedonian
-----Swedish
Switzerland-----German
United States-----English
Finland-----Finnish
-----Icelandic
-----Czech
Malta-----English
Slovenia-----Slovenian
Slovakia-----Slovak
-----Italian
Turkey-----Turkish
-----Chinese
-----Thai
Saudi Arabia-----Arabic
-----Norwegian
United Kingdom-----English
Serbia and Montenegro-----Serbian
-----Lithuanian
-----Romanian
New Zealand-----English
Norway-----Norwegian
Lithuania-----Lithuanian
Nicaragua-----Spanish
*/
```

```
        }
}
```

2)**NumberFormat:**

→ Various countries follows various Conversions to represent number.

→ By using NumberFormat class we can format the Number according to a particular Locale.

→ NumberFormat class is an abstract class, hence We can't create NumberFormat object directly

→ NumberFormat class present in java.txt.

Creating NumberFormat object for the default Locale:

```
public static Numberformat getInstance();
public static Numberformat getCurrencyInstance();
public static Numberformat getPercentInstance();
public static Numberformat getNumberInstance();
```

Getting NumberFormat object for a specific Locale:

```
public static NumberFormat getCurrencyInstance(Locale l);
```

→ Once we got NumberFormat object we can format and parse numbers by using the following methods:

```
public String format(long l);
public String format(double d);
```

→ To format or convert java specific number form to

```
public Number parse(String s) throws ParseException
```

```
import java.text.*;
import java.util.*;

class NumberFormatTest {
        public static void main(String srga[]) {
                double d = 123456.789;
                Locale india = new Locale("Pa", "IN");
                NumberFormat nf1 = NumberFormat.getCurrencyInstance(india);
                System.out.println("India Notation is..." + nf1.format(d));

                NumberFormat nf2 = NumberFormat.getCurrencyInstance(Locale.US);
                System.out.println("Us Notation is..." + nf2.format(d));

                NumberFormat nf3 = NumberFormat.getCurrencyInstance(Locale.UK);
                System.out.println("UK Notation is..." + nf3.format(d));

        }
}
/*Output:
java NumberFormatTest
India Notation is...INR 123,456.79
```

```
Us Notation is...$123,456.79
UK Notation is...£123,456.79
*/



import java.text.*;
import java.util.*;

class NumberFormatTest {
	public static void main(String srga[]) {
		double d = 123456.789;

		NumberFormat nf1 = NumberFormat.getInstance();
		nf1.setMaximumFractionDigits(2);

		System.out.println("setMaximumFractionDigits(2)...");
		System.out.println("\t123.456789..." + nf1.format(123.456789));
		System.out.println("\t123.4..." + nf1.format(123.4));

		System.out.println("setMinimumFractionDigits(2)...");
		nf1.setMinimumFractionDigits(2);
		System.out.println("\t123.456789..." + nf1.format(123.456789));
		System.out.println("\t123.4..." + nf1.format(123.4));

	}
}
/*Output:
java NumberFormatTest
setMaximumFractionDigits(2)...
	123.456789...123.46
	123.4...123.4
setMinimumFractionDigits(2)...
	123.456789...123.46
	123.4...123.40
*/


/**@author:VenukumarS  @date:Feb 15, 2013
 * @fileName:y.java
 */
package JAVATest;


import java.text.NumberFormat;
import java.util.Locale;

class NumberFormatTest {

	static public void displayNumber(Locale currentLocale) {

		Integer quantity = new Integer(123456);
		Double amount = new Double(345987.246);
		NumberFormat numberFormatter;
```

```
            String quantityOut;
            String amountOut;

            numberFormatter = NumberFormat.getNumberInstance(currentLocale);
            quantityOut = numberFormatter.format(quantity);
            amountOut = numberFormatter.format(amount);
            System.out.println(quantityOut + "   " + currentLocale.toString());
            System.out.println(amountOut + "   " + currentLocale.toString());
    }

    static public void displayCurrency(Locale currentLocale) {

            Double currency = new Double(9876543.21);
            NumberFormat currencyFormatter;
            String currencyOut;

            currencyFormatter = NumberFormat.getCurrencyInstance(currentLocale);
            currencyOut = currencyFormatter.format(currency);
            System.out.println(currencyOut + "   " + currentLocale.toString());
    }

    static public void displayPercent(Locale currentLocale) {

            Double percent = new Double(0.75);
            NumberFormat percentFormatter;
            String percentOut;

            percentFormatter = NumberFormat.getPercentInstance(currentLocale);
            percentOut = percentFormatter.format(percent);
            System.out.println(percentOut + "   " + currentLocale.toString());
    }

    static public void main(String[] args) {

            Locale[] locales = { new Locale("Pa", "IN"), new Locale("fr", "FR"),
                            new Locale("de", "DE"), new Locale("en", "US") };

            for (int i = 0; i < locales.length; i++) {
                    System.out.println();
                    displayNumber(locales[i]);
                    displayCurrency(locales[i]);
                    displayPercent(locales[i]);
            }
    }
}

/*Output:
 java NumberFormatTest
123,456   pa_IN
345,987.246   pa_IN
INR 9,876,543.21   pa_IN
75%   pa_IN

123, 456   fr_FR
```

```
345,987,246   fr_FR
9 ,876, 543,21 €   fr_FR
75 %   fr_FR

123.456   de_DE
345.987,246   de_DE
9.876.543,21 €   de_DE
75%   de_DE

123,456   en_US
345,987.246   en_US
$9,876,543.21   en_US
75%   en_US
*/
```

3)**DateFormat:**

→ Various countries follows various Conversions to represent Date.

→ By using DateFormat class we can format the DATE according to a particular Locale.

→ DateFormat class is an abstract class and present in java.text.

To get DateFormat for default Locale, DateFormat class defines the following methods:

1. public static DateFormat getInstance();
2. public static DateFormat getDateInstance();
3. public static DateFormat getDateInstance(int style);

where style is DateFormat.FULL → 0
DateFormat.LONG → 1
DateFormat.MEDIUM → 2
DateFormat.SHORT → 3

To get DateFormat object for the specific Locale:

public static DateFormat getDateInstance(int style, Locale l);

Default style is MEDIUM and most of the cases default Locale is US

→ Once we got DateFormat object we can format and parse dates by using the following methods:

public String format(Date d);
public Date parse(String s) throws ParseException

→ To convert Locale specific DateForm to java DateForm.

To get Date and Time:

public static DateFormat getDateTimeInstance();
public static DateFormat getDateTimeInstance(int dateStyle, int timeStyle);
public static DateFormat getDateTimeInstance(int dateStyle, int timeStyle, Locale l);

```
import java.text.*;
import java.util.*;
class DateFormatTest {
        public static void main(String srga[]) {
```

```
                System.out.println("Full Date Form.."
                                + DateFormat.getDateInstance(0).format(new Date()));
                System.out.println("Long Date Form.."
                                + DateFormat.getDateInstance(1).format(new Date()));
                System.out.println("Medium Date Form.."
                                + DateFormat.getDateInstance(2).format(new Date()));
                System.out.println("Short Date Form.."
                                + DateFormat.getDateInstance(3).format(new Date()));

                System.out.println("UK Form..."
                                + DateFormat.getDateInstance(0, Locale.UK).format(new Date()));
                System.out.println("US Form..."
                                + DateFormat.getDateInstance(0, Locale.US).format(new Date()));
                System.out.println("Italy Form..."
                                + DateFormat.getDateInstance(0, Locale.ITALY)
                                                .format(new Date()));

                System.out.println("US Form..."
                                + DateFormat.getDateTimeInstance(0, 0, Locale.US).format(
                                                new Date()));

        }

}
/*Output:
Full Date Form..Wednesday, March 6, 2013
Long Date Form..March 6, 2013
Medium Date Form..Mar 6, 2013
Short Date Form..3/6/13
UK Form...Wednesday, 6 March 2013
US Form...Wednesday, March 6, 2013
Italy Form...mercoledì 6 marzo 2013
US Form...Wednesday, March 6, 2013 4:25:45 PM IST
*/
```

Regular Expressions:

Regular Expressions:

Any group of Strings according to a particular Pattern is called "regular Expression".

ex: 1) we can write a regular expression to represent all valid mail-ids & by using that regular expression
we can validate whether the given mail-id is valid or not.
2) we can write a regular expression to represent all valid java identifiers.

The main **application areas** of regular expressions are:
We can
1) implement validation mechanism.
2) develop Pattern matching applications.
3) develop translators like compilers, interpreters etc.
4) use for designing digital circuits.
5) use to develop communication protocols like TCP/IP, UDP etc.

Pattern class: A Pattern obj represents compiled version of regular expression we can create a pattern obj by using compile() of Pattern class.

Pattern p = Pattern.compile(String regularExpression);

Matcher class: A Matcher obj can be used to match char sequence against a regular expression.
Matcher m = p.matcher(String target);

Matcher class methods:

boolean find(): it attempts to find next match & of it is available returns True otherwise False.
int start(): returns starting index of the match
int end(): returns end index of the match
String group(): returns the matched pattern

Character classes:

1) [a-z] → any lower case alphabet symbol
2) [A-Z] → any upper case alphabet symbol
3) [a-zA-Z] → any alphabet symbol
4) [0-9] → any digit
5) [abc] → either a or b or c
6) [^abc] → except a or b or c
7) [0-9a-zA-Z] → any alpha numeric char

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
  * @fileName:RegExDemo.java
 */

import java.util.regex.*;

class RegExDemo {
	public static void main(String args[]) {
		Pattern p = Pattern.compile("ab");
		Matcher m = p.matcher("abbbabbcbdab");

		while (m.find()) {
			System.out.println(m.start() + "----" + m.end() + "-----"
					+ m.group());
		}
	}
}

/*
Output:
java RegExDemo

0----2-----ab
4----6-----ab
10----12-----ab
*/
```

Predefined-character class:

\\s → Space char
\\d → [0-9]
\\w→ [0-9a-zA-Z]
\\. → Any Char

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
  * @fileName:RegExDemo1.java
 */

import java.util.regex.*;

class RegExDemo1 {
	public static void main(String args[]) {
		// Pattern p=Pattern.compile("\\d");//[0-9]
		// Pattern p=Pattern.compile("\\s"); //space
		// Pattern p=Pattern.compile("\\w"); //[0-9a-zA-Z]
		Pattern p = Pattern.compile("."); // any char

		Matcher m = p.matcher("a3z4@ k7#");
		while (m.find()) {
			System.out.println(m.start() + "-----" + m.group());
		}
	}
}

/*
Output:

for "Pattern p=Pattern.compile("\\d");"---> java RegExDemo1
1-----3
3-----4
7-----7

for "Pattern p=Pattern.compile("\\s");"---> java RegExDemo1

5-----

for "Pattern p=Pattern.compile("\\w");"---> java RegExDemo1

0-----a
1-----3
2-----z
3-----4
6-----k
7-----7

for "Pattern p=Pattern.compile(".");"---> java RegExDemo1

0-----a
```

```
1-----3
2-----z
3-----4
4-----@
5-----
6-----k
7-----7
8-----#
*/
```

Qualifiers:

We can use Qualifiers to specify number of characters to match

1) **a** → exactly one 'a'
2) **a+** → at least one 'a'
3) **a*** → any number of 'a's
4) **a?** → at most one 'a'

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
  * @fileName:RegExDemo2.java
 */

import java.util.regex.*;

class RegExDemo2 {
	public static void main(String args[]) {
		// Pattern p=Pattern.compile("a");
		// Pattern p=Pattern.compile("a+");
		// Pattern p=Pattern.compile("a*");
		Pattern p = Pattern.compile("a?");
		Matcher m = p.matcher("abaabaaab");

		while (m.find()) {
			System.out.println(m.start() + "----" + m.group());
		}
	}
}

/*
Output:
Pattern.compile("a") --> java RegExDemo2

0----a
2----a
3----a
5----a
6----a
7----a

Pattern.compile("a+") --> java RegExDemo2

0----a
```

```
2----aa
5----aaa

Pattern.compile("a*") --> java RegExDemo2

C:\Docum
0----a
1----
2----aa
4----
5----aaa
8----
9----


Pattern.compile("a?") --> java RegExDemo2

0----a
1----
2----a
3----a
4----
5----a
6----a
7----a
8----
9----

*/
```

split(): Pattern class contains split() to split given String according to a regular expression.

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
 *   @fileName:RegExDemoSplit.java
 */

import java.util.regex.*;

class RegExDemoSplit {
        public static void main(String args[]) {
                Pattern p = Pattern.compile("\\s");

                String[] s = p.split("Ebix software India Pvt Ltd");

                for (String s1 : s) {
                        System.out.println(s1);
                }
        }
}

/*
Output:
java RegExDemoSplit
```

```
Ebix
software
India
Pvt
Ltd

*/
```

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
  * @fileName:RegExDemoSplit1.java
 */

import java.util.regex.*;

class RegExDemoSplit1
{
public static void main(String args[])
{
Pattern p=Pattern.compile("\\.");

String[] s=p.split("www.google.com");

for(String s1:s)
{
System.out.println(s1);
}
}
}

/*
Output:
java RegExDemoSplit1

www
google
com
*/
```

String class split(): String class also contains **split()** to split given String against to a regular expression.

*** **Note:** "**Pattern class**" **split()** can take target String as arguments, where as "**String class**" **split()** can take regular expression as argument.

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
  * @fileName:RegStringSplit.java
 */

import java.util.regex.*;
```

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

		String s = "www.google.com";
		String[] s1 = s.split("\\.");

		for (String s2 : s1) {
			System.out.println(s2);
		}
	}
}

/*
Output:
java RegStringSplit

www
google
com

*/
```

StringTokenizer:

→ We can use StringTokenizer to divide the target **String into stream of Tokens** according to the specified expression.

→ StringTokenizer class presenting in java.util

→ The *default regular expression* is space.

Ex1:

```
StringTokenizer st = new StringTokenizer("Guru Raam Kumar");
while (st.hasMoreTokens()) {
		System.out.println(st.nextToken());
}

Output:
Guru
Raam
Kumar
```

Ex2:

```
StringTokenizer st = new StringTokenizer("10,00,000", ",");
while (st.hasMoreTokens()) {
		System.out.println(st.nextToken());
}

Output:
10
00
000
```

System Properties:

```
/**@author:Venu Kumar.S  @date:Oct 6, 2012
  * @fileName:SysPropTest.java
 */
import java.util.*;

class SysPropTest {
        public static void main(String args[]) {
                Properties p = System.getProperties();
                p.list(System.out);
        }
}
/*
Output:
java SysPropTest

It shows all system properties.

Note:we can set system property from the command prompt by using -D option
        java -D<name of the property>=<value of the property> class_name
ex: java -Debix=QPS SysPropTest
*/
```

JAR(Java Archive) file

If several dependent files are available then it is never recommended to set each file individually in the class-path we have to group all those "**.class**" file into a single zip file and we have to make that *zip* file available in the class-path. This zip file is nothing but JAR file.

Ex: To develop Servlet all required .class files are available in **servlet_api.jar**. We have to make this JAR file available in the class-path then only Servlet will be compiled.

jar vs war vs ear:

→ **jar: j**ava **ar**chive file. It contains a group of "**.class**" files.

→ **war: w**eb **ar**chive file. It represents a web application which may contain Servlets, JSPs, HTMLs, CSS file, JavaScripts, etc..

→ **ear: e**nterprise **ar**chive file. It represents an enterprise application, which may contains Servlets, JSPs, EJBs, JNS components etc...

→ **Commands:**

1) **Create a jar file:**

```
jar -cvf guru.jar A.class B.class C.class
or
jar -cvf guru.jar *.class
```

2) **To extract a jar file**

```
jar -xvf guru.jar
```

3) **To Display table of contents of a jar file:**

```
jar -tvf guru.jar
```

Ex:

```
public class GuruColorFullCalc {
	public static int add(int x, int y) {
		return x * y;
	}

	public static int multiply(int x, int y) {
		return 2 * x * y;
	}
}
/*
1)javac GuruColorFullCalc.java

2)jar -cvf c:\guruCalc.jar GuruColorFullCalc.class

*/


class GuruJarTest {
	public static void main(String args[]) {
		System.out.println(GuruColorFullCalc.add(10, 20));
		System.out.println(GuruColorFullCalc.multiply(10, 20));
	}
}

/*
```

To Compile:

→ 1) javac GuruJarTest.java

→ 2)javac -cp c: GuruJarTest.java

CE:

```
VenuJarTest.java:5: cannot access GuruColorFullCalc
bad class file: c:\GuruColorFullCalc.class
unable to access file: c:\GuruColorFullCalc.class (The system cannot find the fi
le specified)
Please remove or make sure it appears in the correct subdirectory of the classpa
th.
System.out.println(GuruColorFullCalc.add(10,20));                ^
1 error
```

→ 3)javac -cp c:\guruCalc.jar GuruJarTest.java

OK

To Run:

→ 4)java -cp c:\guruCalc.jar GuruJarTest

```
Exception in thread "main" java.lang.NoClassDefFoundError: GuruJarTest
Caused by: java.lang.ClassNotFoundException: GuruJarTest
	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)
Could not find the main class: GuruJarTest.  Program will exit.

OUTPUT:

→  5) java -cp .;c:\guruCalc.jar GuruJarTest
200
400
*/
```

To Create Executable JAR file for "GuruJarTest"in Java

1) Add GuruJarTest to **MANIFEST.MF**

```
C:\>echo Main-Class: GuruJarTest>manifest.txt
```

2) Create executable jar file.

```
C:\>jar cvfm Guru.jar manifest.txt *.class
added manifest
adding: VenuColorFullCalc.class(in = 319) (out= 229)(deflated 28%)
adding: VenuJarTest.class(in = 483) (out= 338)(deflated 30%)
```

3) Execute the jar file

```
C:\>java -jar Guru.jar
200
400
```

To Create Executable JAR file in Java

1) **Create a directory "my" and change to "my"**

```
C:\>md my
C:\>cd my
```

2) **create Hello.java**

```
public class Hello {
        public static void main(String args[]) {
                System.out.println("Hello! Welcome to Executable Jar");
        }
}
```

3) **compile Hello.java**

```
javac Hello.java
```

4) **Create jar file**

Syntax: jar -cvf JarFilename.jar ClassFileName.class

-c create new archive
-v generate verbose output on standard output
-f specify archive file name

Ex: C:\my>jar -cvf Hello.jar Hello.class
added manifest
adding: Hello.class(in = 436) (out= 300)(deflated 31%)

This command will crate **Hello.jar** which contains **Hello.class** file. this will also create manifest file but without Main-Class entry

Manifest-Version: 1.0
Created-By: 1.6.0-beta2 (Sun Microsystems Inc.)

This jar can not be executed and you will get error when you try to run this jar file:

5) **To execute Jar file**

Syntax: java -jar JarFileName.jar

Ex: C:\my>**java -jar Hello.jar**
Failed to load Main-Class manifest attribute from Hello.jar

To get rid of this error we just need to provide **Main-Class** entry.

Approach 1:

Executable JAR File Example with External Manifest

CreateMANIFEST.MF file by using any text editor e.g. notepad in windows and add following entry in file, remember last line must end with either new line or carriage return:

Manifest-version: 1.0
Main-Class: Hello

Important thing to remember is that we need to specified full classified class name here. suppose if our main class was inside **com/jarexample/Hello** than we should have to specify **com.jarexample.Hello** here, don't put .class extension here its not required. Apart from specifying **Main-Class** you can also specify Java Classpath in Manifest file which is important if your application is depended on external library jars. "Classpath" entry supersede both -cp and CLASSPATH environment variable.

1)**To add Main-class file execute the following command**

syntax: echo Main-Class: ClassFileName>manifest.txt

Ex: C:\my>**echo Main-Class: Hello>manifest.txt**

2)Execute following **jar command to create executable jar**

Syntax: jar -cvfm JarFileName.jar manifest.txt **ClassFileName.class**

Ex: C:\my>**jar -cvfm Hello.jar** manifest.txt **Hello.class**
jar -cvfm Hello.jar manifest.txt Hello.class
added manifest

adding: Hello.class(in = 436) (out= 300)(deflated 31%)

here -m include manifest information from specified manifest file and remember specify name of manifest file after jar name. Now we have an executable jar file in java which you run by command specified earlier.

3)**Executing Java Program from JAR file**

Ex: C:\my>java -jar Hello.jar
Hello! Welcome to Executable Jar

Approach-2

Creating Executable JAR File By entry point

This seems to an **easy way to create executable jars in Java**, as you need not have to create manifest file explicitly and it will be create by jar command itself along with Main-Class entry. What you need to provide is a new jar option "-e" and you main class name while running jar command. here is **example of jar command** with entry option:

1) **jar command to create executable jar**

syntax: jar -cvfe jarFileName.jar ClassFileName ClassFileName.class

-e specify application entry point for stand-alone application

bundled into an executable jar file

Ex: C:\my>**jar -cvfe Hello.jar Hello Hello.class**
added manifest
adding: Hello.class(in = 436) (out= 300)(deflated 31%)

jar -e for entry point and entry point or main class name should come after jar file name and before directory or file needs to be included in JAR. You can now run your executable jar file by issuing **"java -jar" command** as shown in following example:

2)**To execute Java Program from Jar file**

Executing jar program from jar archive is very easy one thing required is **jar must be executable** and must have *Main-Class entry in* MANIFEST.MF. here is a Java command example for running java program from jar file:

Syntax: java -jar JarFileName.jar

here we have specified jar file name with *-jar option* and it will run main class declared as "Main-Class" attribute in manifest file.

Ex:C:\my>**java -jar HelloWorld.jar**
Hello! Welcome to Executable Jar

How to view contents of a JAR file in Java

jar command in Java allows you to view files and directories inside of a jar file without extracting or unzipping original jar. "-t" jar option is used to list files from jar archive.

Syntax: **jar -tvf JarFileName.jar**

Ex:C:\my>**jar -tvf Hello.jar**
0 Fri Mar 08 14:22:02 IST 2013 META-INF/
90 Fri Mar 08 14:22:02 IST 2013 META-INF/MANIFEST.MF
436 Fri Mar 08 14:15:14 IST 2013 Hello.class

here "**-t**" to list table of contents for archive.

How to extract contents of JAR File

Use **jar option "-x" for extracting files** from JAR files.

Syntax: **jar -xvf JarFileName.jar**

Ex: C:\my>jar -xvf Hello.jar
created: META-INF/
inflated: META-INF/MANIFEST.MF
inflated: Hello.class

-x extract named (or all) files from archive

Wrapper classes

→ The main objective of wrapper classes are

i) To wrap primitives into object form, so that we can handle primitive just objects.
ii) To define several utility methods for the primitives.

Constructors of Wrapper classes or creation of Wrapper objects:

→ All most all wrapper classes contains two constructors. One can take corresponding primitive as argument & the other can take String as argument.

```
Integer I = new Integer(10);
Integer I = new Integer("10");

Double D = new Double(10.5);
Double D = new Double("10.5");
```

→ If the String is not properly formatted then we will get RE:NumberFormatException

```
Integer I = new Integer("Ten"); --> RE:NumberFormatException
```

→ Float class contains 3 constructors one can take float primitive, 2nd one can take String and 3rd one can take double arg.

```
Float f=new Float(10.5f);
Float f=new Float("10.5f");
Float f=new Float(10.5); --> double arg.
```

→ Character class contains only one constructor which can take char primitive as arg.

```
Character ch=new Character('a');
```

→ Boolean class contains two constructors on can take boolean primitive as the arg & other can take

String as arg.

If we are passing primitive as arg the only allowed values are true, false.

```
Boolean B = new Boolean(true);
Boolean B = new Boolean(false);
```

If we are passing String arg to the Boolean constructor the the case is not important & content is not important.

```
Boolean b = new Boolean("true"); → true
```

If the content case insensitive string , it takes true. Otherwise it is treated as false.

```
Boolean b = new Boolean("True"); → true
Boolean b = new Boolean("TRUE"); → true

Boolean b = new Boolean("xyz"); → false
Boolean b = new Boolean("yes"); → false
```

Wrapper classes corresponding constructor arg:

Byte → byte or String
Short → Short or String
Integer → int or String
Long → long or String
Float → float or String or double
Double → double or String
Character → char
Boolean → boolean or String

```
Boolean b1 = new Boolean("yes");
Boolean b1 = new Boolean("no");

System.out.println(b1.equals(b2)); → true
System.out.println(b1==b2); → false
System.out.println(b1); → false
System.out.println(b2); → false
```

```
class TestBoolean {
    public static void main(String args[]) {
        Boolean b1=new Boolean("yes");
        Boolean b2=new Boolean("no");

        System.out.println(b1.equals(b2)); // true
        System.out.println(b1==b2); //false
        System.out.println(b1); // false
        System.out.println(b2); // false
    }
}
```

```
/*OUTPUT:
true
false
false
false
*/
```

→ In every Wrapper class **toString()** is overridden to return it's content.

→ In every Wrapper class **equals()** is overridden for content comparison.

valueOf(): This method can be used for creating Wrapper object as alternative to constructor

→ Every Wrapper class except Character class contains a static valueOf() for converting for String to the Wrapper object.

Syntax: **public static Wrapper valueOf(String s);**

```
Integer i = Integer.valueOf("10");
Boolean b = Boolean.valueOf("True");
```

→ Every integral type Wrapper class(**Byte, Short, Integer, Long**) contains the following **valueOf()** to convert specified Radix String form to corresponding Wrapper object.

Syntax2: **public static Wrapper valueOf(String s, int radix);**

```
Integer i1 = Integer.valueOf("1010",2);
System.out.println(i1); →  10

Integer i2 = Integer.valueOf("1111",2);
System.out.println(i2); →  15
```

→ Every Wrapper class including Character class contains the following **valueOf()** to convert primitive to corresponding Wrapper object.

Syntax3: **public static Wrapper valueOf(primitive p);**

```
Integer i = Integer.valueOf(10);
Character ch = Character.valueOf('a');
Boolean B = Boolean.valueOf(true);
```

xxxValue(): This method is used to convert Wrapper object to primitivies.

→ Every number type Wrapper class contains the following six **xxxValue()** methods.

```
public byte byteValue();
public short shortValue();
public int intValue();
public long longValue();
public float floatValue();
public double doubleValue();
```

```
Double d=new Double(130.456);
```

```
System.out.println(d.byteValue()); --> -126
System.out.println(d.shortValue()); --> 130
System.out.println(d.intValue()); --> 130
System.out.println(d.floatValue()); -->130.45
```

charValue(): Character class contains Char value method to convert Character object to the char primitive.

Syntax: **public char charValue();**

```
Character ch=new Character('@');
char ch1=ch.charValue();
System.out.println(ch1); → @
```

booleanValue(): Boolean class contains this method to find boolean primitive for the given boolean object.

Syntax: **public boolean booleanValue();**

```
Boolean b=Boolean.valueOf("xyz");
boolean b1=b.booleanValue();
System.out.println(b1); → false
```

parseXxx(): This method is used to convert Strinf to corresponding primitive.

1) Every Wrapper class except Character class contains the following parseXxx() to convert String to corresponding primitive.

Syntax: **public static primitive parseXxx(String s);**

```
int i=Integer.parseInt("10");
double d= Double.parseDouble("3.45");
long l=Long.parseLong("10l");
boolean b=Boolean.parseBoolean("xyz"); → false
```

2) Every Integeral type Wrapper class contains the following prseXxx() to convert specified radix String to corresponding primitive.

Syntax: **public static primitive parseXxx(String s, int radix);**

```
int i=Integer.parseInt("1111",2);
System.out.println(i); → 15
```

toString(): We can use **toString()** to convert Wrapper object or primitive to String.

→ Every Wrapper class contains the following toString() to convert Wrapper object to String type.

Syntax: **public String toString();**

It is the overriding version of Object class **toString()**

```
Integer i=new Integer(10);
System.out.println(i.toString()); → 10
```

→ Every wrapper class contains a static toString() to convert primitive to Strinf form.

Syntax: **public static String toString(primitive p);**

```
String s=Integer.toString(10);
String s=Boolean.toString(true);
```

→ Integer & Long classes contains toString() to convert primitive to specified radix String form.

Syntax: **public Static String toString(primitive p, int radix);**

```
String s=Integer.toString(15,2);
System.out.println(s); →  1111
```

→ Integer & Long classes contains the following **toXxxString()**

Syntax: **public static String toBinaryString(primitive p);**
public static String toOctalString(primitive p);
public static String toHexString(primitive p);

```
String s=Integer.toHexString(123);
System.out.println(s); →  7b
```

valueOf(): String, primitive to Wrapper object.
xxxValue(): Wrapper object o tprimitive.
parseXxx(): String to primitive.
valueOf(): String, primitive to Wrapper object.
toString(): primitive, Wrapper object to String.

java.io.*

File:

→ This will not create any physical file, first it will check is there any file named with "abc.txt" is available or not.

→ If it is available then the File object simply pointing to that file. Otherwise it represents just name of the file without creating any physical file.

```
File f=new File("abc.txt");
System.out.println(f.exists()); →  false

f.createNewFile(); // creates physical file with the name of abc.txt
System.out.println(f.exists()); →  true
```

→ A Java file object can represent a directory also.

```
File f=new File("guru");
System.out.println(f.exists()); →  false

f.mkdir(); //it creates a dir with the name of guru
System.out.println(f.exists()); →  true
```

Constructors:

1)**File f=new File(String name);** → creates a java file object to represent name of a file or directory

2)**File f=newfile(String subdir, String name);** → To create a file or directory present in some other sub-directory

3)**File f=new File(Filesubdir, String name);**

Ex: 1) File f=new File("abc.txt");
f.createNewFile(); → creates abc.txt file in current working directory

Ex: 2) File f1=new File("Guru");
f1.mkdir(); → it creates a directory with the name of Guru in current working directory

Ex: 3) File f2=new File("Guru","abc.txt");
f2.createNewFile(); → creates new file abc.txt in Guru directory

```
File f2=new File(f1,"abc.txt");
f1.createNewFile();
```

Important methods of File class:

1) **boolean exists():** → returns true if the physical file or directory presents otherwise false.

2) **boolean createNewFile():** → This method will check whether the specified file is already available or not. if it is already available then this method returns false without creating new file. if it is not already available then this method returns true after creating new file.

3) **boolean mkdir():** → This method will check whether the specified directory is already available or not. if it is already available then this method returns false without creating new directory . if it is not already available then this method returns true after creating new directory.

4) **boolean isFile():** → It returns true if it is a file, otherwise false

5) **boolean isDirectory():** → It returns true if it is a directory, otherwise false

6)**String[] list():** → It returns the names of all files & sub-directories present in the specified directory.

7) **boolean delete():** → It deletes a file or directory

8) **long length():** → returns the number of characters present in the specified file.

program: To print the names of all files & directories present in "d:\guru"

```
import java.io.*;

class Test {
        public static void main(String args[]) {
                File f = new File("D:\\guru");
                String[] s = f.list();
                for (String s1 : s) {
                        System.out.println(s1);
```

```
                }
            }
        }
```

FileWriter: FileWriter object is used to write character data to the File.

To overriding a file contents.

1) FileWriter fw=new FileWriter(String name);

2) FileWriter fw=new FileWriter(File f);

To append a file contents:

3) FileWriter fw=new FileWriter(String name, boolean append);

4) FileWriter fw=new FileWriter(File f, boolean append);

→ If the specified file is not already available then the above constructors will create that file.

Methods:

1) **write(int ch):** → To write a single character to the file.

2) **write(char[] ch):** → To write an array of characters to the file.

3) **write(String s):** → To write a string to the file.

4) **flush():** → To give the guarantee that last character of the data also return to the file.

5) **close():** → To close the FileWriter

Program:

```
import java.io.*;

class FileWriterTest {
        public static void main(String args[]) throws IOException {
                FileWriter fw = new FileWriter("we.txt", true); // appending we.txt
                fw.write(100); // adding a single char 'd'

                char[] ch = { 'x', 'y', 'z' };
                fw.write(ch); // adding array chars

                fw.write('\n');

                fw.write("Chandra\nGuru"); // adding string

                fw.flush();
                fw.close();
```

```
        }
    }
```

→ If **throws IOException** is not included it shows

CE: Unhandled exception type IOException
CE:unreported exception java.io.IOException

```
1.import java.io.*;
2.
3.class FileWriterTest {
4.      public static void main(String args[]) {
5.              FileWriter fw = new FileWriter("we.txt", true); // appending we.txt
6.              fw.write(100); // adding a single char 'd'
7.
8.              char[] ch = { 'x', 'y', 'z' };
9.              fw.write(ch); // adding array chars
10.
11.             fw.write('\n');
12.
13.             fw.write("Chandra\nGuru"); // adding string
14.
15.             fw.flush();
16.             fw.close();
17.
18.     }
19.}
```

→ From the above program line number 5, 6, 9, 11, 13, 15, & 16 stating that
CE:unreported exception **java.io.IOException**; must be caught or declared to be thrown

3) FileReader: It is used to read character data from the file

```
FileReader fr=new FileReader(String name);
FileReader fr=new FileReader(File f);
```

Methods:

1) **int read():** → It attempts to read next character from the file and return its Unicode value. If the next character is not available the it returns "-1".

2) **int read(char[] ch):** → It attempts to read enough characters from the file into the character array and returns the number of characters which are copied from file to the char[].

3) **close():** To close the FileReader

Program:

```
import java.io.*;

class FileReaderTest {
```

```
        public static void main(String args[]) throws IOException {
                File f = new File("we.txt");

                FileReader fr = new FileReader(f);

                System.out.println(fr.read()); // prints unicode of first character

                char[] ch = new char[(int) (f.length())];
                fr.read(ch); // file data is copied to charater array ch

                for (char ch1 : ch) {
                        System.out.print(ch1);
                }

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

                FileReader fr1 = new FileReader(f);
                int i = fr1.read();
                while (i != -1) {
                        System.out.print((char) i);
                        i = fr1.read();
                }
                fr.close();

        }
}
```

Usage of FileWriter & FileReader is not recommended because:

1) While writing data by **FileWriter** we have to insert line separators manually which is a bigger risk to the programmer.

2) By using **FileReader** we can read data character by character which is not convenient to the programmer.

To resolve these problems **BufferedWriter** & **BufferedReader** classes are introduced.

BufferedWriter: It is used to write character data to the file.

BufferedWriter bw = new BufferedWriter(Writer w);
BufferedWriter bw = new BufferedWriter(Writer w, int buffersize);

→ BufferedWriter never communicates directly with the file compulsory it should communicate via some Writer object only.

Methods:

write(int ch);
write(char[] ch)
write(String s)
flush()
close()
newLine();

program:

```
import java.io.*;

class BufferedWriterTest {
        public static void main(String args[]) throws IOException {
                File f = new File("we.txt");
                FileWriter fw = new FileWriter(f);

                BufferedWriter bw = new BufferedWriter(fw);
                bw.write(100);
                bw.newLine();

                char[] ch = { 'x', 'y', 'z' };
                bw.write(ch); // adding array chars
                bw.newLine();

                bw.write("KumaraGuru");
                bw.newLine();

                bw.flush();

                bw.close();
        }
}
```

→ Whenever we are closing BufferedWriter automatically underlying writers will be closed.

BufferedReader: The main advantage of BufferedReader over FileReader is we can read the data line by line instead of reading character by character. This approach improves performance of the system by reducing the number of read operations.

BufferedReader br = new BufferedReader(Reader r);
BufferedReader br = new BufferedReader(Reader r, int buffersize);

→ BufferedReader never communicates directly with the file compulsory it should communicate via some Reader object only.

Methods:

int read();
int read(char[] ch);
close();
String readLine(); → It attempts to find the nextLine & if the nextLine is available then it returns it, otherwise it returns **null**.

Program:

```
import java.io.*;

class BufferedReaderTest {
        public static void main(String args[]) throws IOException {
                FileReader fr = new FileReader("we.txt");
```

```
                BufferedReader br = new BufferedReader(fr);
                String line = br.readLine();
                while (line != null) // while((line=br.readLine())!=null)
                {
                        System.out.println(line);
                        line = br.readLine();
                }
                br.close();
        }
}
```

→ Whenever we are closing BufferedReader,automatically underlying Readers will be closed.

PrintWriter: This is the most enhanced writer to write a character data to a file. By using FileWriter & BufferedWriter we can write only character data but by using PrinterWriter we can write any primitive data types to the file.

```
PrintWriter pw=new PrintWriter(String name);
PrintWriter pw=new PrintWriter(File f);
PrintWriter pw=new PrintWriter(Writer w);
```

Methods:

```
write(int ch);
write(cahr[] ch);
writer(String s);
flush();
close();

print(char ch)
print(int I)
print(long l)
print(String s)

println(char ch)
println(int i)
println(long l);
println(String s)
```

Program:

```
import java.io.*;

class PrintWriterTest {
        public static void main(String args[]) throws IOException {
                FileWriter fw = new FileWriter("we.txt");
                PrintWriter pw = new PrintWriter(fw);

                pw.write(100);
                pw.println(100);// 100

                pw.println(true); // true
                pw.println('x');// x
                pw.println("guru"); // guru
                pw.flush();
```

```
                pw.close();

            }
        }
```

→ Readers & Writers are meant for handling character data(any primitive type.

→ To handle binary data like images, movies, jar files.., we should go for streams.

→ we can use InputStream to read binary data & OutpuStream to write a Binary data.

→ we can use ObjectInputStream & ObjectOutputStream to read & write objects to a file respectively(Serialization)

→ The most enhanced writer to write character data is PrintWriter where as the most enhanced Reader to read character data is BufferedReader.

Object → Writer(Abstract Class) → OutputStreamWriter → FileWriter
→ BufferedWriter
→ PrintWriter

→ Reader(Abstract Class) → InputStreamReader → FileReader
→ BufferedReader

To give input to Java program with the help of BufferedReader:

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
datatype var=Wrapper.parseXxx(br.readLine());
```

Ex:

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
int i=Integer.parseInt(br.readLine());
```

Program:

```
import java.io.*;

class Input {
    public static void main(String args[]) throws IOException {

        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

        System.out.println("Enter a String ...");
        String s = br.readLine();

        System.out.println("Enter Double value...");
        double d = Double.parseDouble(br.readLine());

        System.out.println("String=" + s + "..." + "Double=" + d);
```

```
        }
}
```

Output:

```
Enter a String ...
Guru
Enter Double value...
3.56
String=Guru...Double=3.56
```

To give input to a java program:

We are using Scanner class to get input from user. This program firstly asks the user to enter a string and then the string is printed, then an integer and entered integer is also printed and finally a float and it is also printed on the screen. Scanner class is present in java.util package so we import this package in our program. We first create an object of Scanner class and then we use the methods of Scanner class. Consider the statement

```
Scanner a = new Scanner(System.in);
```

Scanner is the class name, a is the name of object, new keyword is used to allocate the memory and **System.in** is the input stream.

nextInt to input an integer
nextFloat to input a float
nextLine to input a string

Example1:

```
import java.util.Scanner;

class GetInputFromUser {
        public static void main(String args[]) {
                int i;
                float f;
                String s;

                Scanner in = new Scanner(System.in);

                System.out.println("Enter a string");
                s = in.nextLine();

                System.out.println("Enter an integer");
                i = in.nextInt();

                System.out.println("Enter a float");
                f = in.nextFloat();

                System.out.println("i=" + i + "..." + "s=" + s + "..." + f);
        }
}
```

Output:

```
Enter a string
Guru
Enter an integer
3
Enter a float
2.75
i=3...s=Guru...2.75
```

Example2:

```
import java.util.Scanner;

 class ScannerDemo {
   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);

            System.out.print("Username: ");
      String username = scanner.nextLine();

           System.out.print("Password: ");
      String password = scanner.nextLine();

      System.out.print("What is 2 + 2: ");
      int result = scanner.nextInt();

      if (username.equals("Guru")
            && password.equals("kumar") && result == 4) {
         System.out.println("Welcome to Java Application");
      } else {
         System.out.println("Invalid username or password or result, " +
               "access denied!");
      }
   }
}
```

Output1:

```
Username: Guru
Password: kumar
What is 2 + 2: 4
Welcome to Java Application
```

Output2:

```
Username: Guru
Password: Raam
What is 2 + 2: 4
Invalid username or password, access denied!
```

Garbage Collection

In java programmer is responsible only for creation of objects and he is not responsible for destruction of useless objects.

→ The main objective of Garbage Collector is to "***destroy useless objects***".

Various ways to make an object eligible for GC:

→ Eventhough programmer is not responsible to destroy useless object, it is always a good programming practice to make an object eligible for GC if it is no longer required.

→ An object is said to be eligible for GC, if it doesn't contain any references.

→ The various passible ways to make an object eligible for GC:

1) **nullyfying the reference variable:**

→ If an object is no longer required then assign null to all its references, then automatically that object eligible for GC.

Ex:

```
Student s1 = new Student();
Student s2 = new Student();
s1 = null; // → s1 is eligible for GC
s2 = null; // → s1 is eligible for GC

/**@author:VenukumarS  @date:Oct 15, 2012
 * @fileName:NullyfingTest.java
 */

class NullyfingTest {
        NullyfingTest i;

        public static void main(String args[]) {
                NullyfingTest t1 = new NullyfingTest();
                NullyfingTest t2 = new NullyfingTest();
                NullyfingTest t3 = new NullyfingTest();
                t1.i = t2;
                t2.i = t3;
                t3.i = t1;

                t1 = null;
                t2 = null;
                t3 = null;
        }
}
```

Note:

→ If an object doesn't have any refe then it is always eligible for GC.

→ Eventhough object having the ref, still it is eligible for FC sometimes.

Mathod for requesting JVM to run GC:

→ Whenwver we are making an object eligible for GC it may not be destroyed by GC immediately when ever JVM runs GC then only that object will be destroyed.

→ We can request JVM to run GC, programmatically whether JVM accepts our request or not there is no gaurantee.

1) **By system class:**

→ **System** class contains a static method **GC** → **System.gc();**

2) **By runtime class:**

→ By using **Runtime** object a java application can communicate with JVM.

→ Runtime class is a Singleton class hence we can't create Runtime object by using constructor.

→ We can create a Runtime object by using factory method get Runtime.

Runtime r=Runtime.getRuntime();

→ Once we got Runtime object we can apply the following method on that object.

a) **freeMemory()** → returns free memory in the heap.

b) **totalMemory()** → returns total memory of the heap(size)

c) **gc()** → for requesting JVM to Run garbage collector.

```
/**@author:VenukumarS  @date:Oct 15, 2012
 * @fileName:RuntimeTest.java
 */
import java.util.Date;
class RuntimeTest {
	public static void main(String args[]) {
		Runtime r = Runtime.getRuntime();
		System.out.println(".totalMemory()..= "+r.totalMemory());
		System.out.println(".freeMemory()..= "+r.freeMemory());
		for (int i = 1; i <= 1000; i++) {
			Date d = new Date();
			d = null;
		}
		System.out.println("\n.totalMemory()..= "+r.totalMemory());
		System.out.println(".freeMemory()..= "+r.freeMemory());
		r.gc();
		System.out.println("\n.totalMemory()..= "+r.totalMemory());
		System.out.println(".freeMemory()..= "+r.freeMemory());

	}
}
/*Output:
	java RuntimeTest
	.totalMemory()..= 16252928
	.freeMemory()..= 16055504

	.totalMemory()..= 16252928
	.freeMemory()..= 15942688

	.totalMemory()..= 16318464
	.freeMemory()..= 16103160
```

```
*/
```

*** **Note:** ***gc()*** present in the **System** class is a **static** method, where as **gc()** present in the Run-time is instatnce mehtod & ercommended to user ***System.gc()***.

****

finalization :

→ Just **before destroying** any object, **garbage collector** always calls ***finalize()*** method to perform clean-up activities on that object.

→ ***finalize()*** declare in **Object** class with following declaration.
protected void finalize() throws Throwable;

Case 1: Garbage collector always calls **finalize()** on the object which is **eligible** for **GC** just before destruction, Then the corresponding class **finalize()** will be executed. If **String** object eligible for **GC** then **String** class **finalize()** will be executed, but not **Test finalize()** method.

```
/**@author:Venu Kumar.S  @date:Oct 15, 2012
 * @fileName:FinalizeTest.java
 */
cclass FinalizeTest {
	public static void main(String args[]) {
		String s = new String("Guru");
		s = null;
		System.gc();
		System.out.println("..End of main()...");
	}

	public void finalize() {
		System.out.println("finalize() method called...");
	}

}

/*Output:
	javac FinalizeTest
	java FinalizeTest
	..End of main()...
*/
```

→ From the program Sting object is eligible for **GC**, Hence, **String** class **finalize()** method got executed which has empty implementation.

→ If we are replacing **String** object with **FinalizeTest** object, then **FinalizeTest** class **finalize()** will be executed.

Case 2:

→ We can call **finalize()** explicitly in this case it will be executed just like a normal method call & **Object** won't be destroyed.

→ But before destuction of an object **GC** always call **finalize()**

```
/**@author:Venu Kumar.S  @date:Oct 15, 2012
 * @fileName:FinalizeTest1.java
```

```
 */
class FinalizeTest1 {
	public static void main(String args[]) {
		FinalizeTest1 t = new FinalizeTest1();
		t.finalize();
		t.finalize();
		t = null;
		System.gc();
		System.out.println(".End of main...");
	}

	public void finalize() {
		System.out.println(".finalize() method called...");
	}

}

/*Output:
	javac FinalizeTest1
	java FinalizeTest1
		.finalize() method called...
		.finalize() method called...
		.End of main...
		.finalize() method called...
*/
```

→ From the program ***finalize()*** got executed 3 times, 2 times explicitly by the programmer **one time** by the **Garbage Collector**.

Note:

→ Before destruction of **Servlet** object web container always calls destroy method, to perform clean-up activities.

→ It is possible to call **destroy()** explicitly from **init()** & **service()** in this case it will be executed just like a normal method call and **Servlet** object won't be destroyed.

Case 3:

→ If we are calling **finalize()** explicitly & while executing that **finalize()**, if **any exception raised** & **uncaught**, then the program will be **terminated abnormally**.

→ If **GC** calls **finalize()** and while executing that **finalize()**, if any exception is uncaught, then JVM simply ignores that uncaught exception & rest of the program will be executed normally.

```
/**@author:Venu KumarsS  @date:Oct 15, 2012
 * @fileName:FinalizeTest2.java
 */
class FinalizeTest2 {
	public static void main(String args[]) {
		FinalizeTest2 t = new FinalizeTest2();
		t.finalize();
		t = null;
		System.gc();
		System.out.println(".End of main...");
	}
```

```
		public void finalize() {
			System.out.println(".finalize() method called...");
			System.out.println(5 / 0);
		}

	}

	/*Output:
		javac FinalizeTest2
		java FinalizeTest2
			.finalize() method called...
		Exception in thread "main" java.lang.ArithmeticException: / by zero
		at FinalizeTest2.finalize(System.out.println(5 / 0);)
		at FinalizeTest2.main(t.finalize();)
	*/
```

→ from the program, if we are not comment "**t.finalize();**", then we are calling the **finalize()** explicily and the program will be terminated abnormally.

→ If we are commenting "**t.finalize();**", then GC calls **finalize()** & the raised **ArithmeticException** is ignored by JVM. Hence, the output is ..

→ On any object **GC** calls **finalize()** only once.

```
/**@author:Venu KumarS  @date:Oct 15, 2012
 * @fileName:FinalizeTest4.java
 */
class FinalizeTest4 {
	 static FinalizeTest4 s;

	public static void main(String args[]) throws Exception {
		FinalizeTest4 f = new FinalizeTest4();
		System.out.println(f.hashCode());
		f = null;
		System.gc();
		Thread.sleep(5000);
		System.out.println(s.hashCode());
		s = null;
		System.gc();
		Thread.sleep(5000);
		System.out.println(".End of main()...");
	}

	public void finilize() {
		System.out.println("..finalize method is called...");
		s = this;
	}

}
```

→ We can't tell exact algorithm followed by **GC**, but most of the cases it is **Mark & Sweep algorithm**.

Memory Leak:

→ If an object having the reference then it is not eligible for GC, eventhough we are not using that object in our program. Still it is destroyed by the GC, such type of object is called "**memory leak**".

→ Memory Leak is a useless object which is not eligible for GC.

→ We can resolve memory leaks by making useless objects for GC explicitly & by **invoking GC** progarmmatically.

Memory leak monitoring tools: JpROBE, IBM Tivoli, HP J meter.

Assertions

→ Very common way of debugging is using **System.out.println()**. But, the problem with *System.out.println()* is after fixing the problem compulsory we should delete these *System.out.println()*, otherwise these *System.out.println()* will be executed at runtime and effects performance & disturbs logging. To resolve this problem, "**Assertions**" concept is introduced in **Java1.4v**. Hence, the main objective of **assertions is to perform debugging**.

→ The main advantage of assertions over *System.out.println()* is after fixing the problem it is not required to delete assert statements because assertions will be disabled automatically at runtime, based on our requirement we can enable & disable assert statements & by default assertions are disabled.

→ **Assertions** concept is applicable for **development** & **test** environment , but **not** for **production environment**.

→ **assert** is a *keyword* introduced in Java**1.4**v. But, **assert** is an *identifier* in Java**1.3**v.

Types of Assertions statements: 1) Simple version 2) Augmented version

1) **Simple version:**

Syntax: assert(b); → where, b is boolean

→ If **b** is **true**, then our **assumption satisfied** & rest of the program will be executed normally.

→ If **b** is **false**, then our **assumption fails**, the program will be terminated by raising runtime exception saying **assertionError**. so, that we can able to fix the problem.

```
/**@author:Venu Kumar.S  @date:Oct 15, 2012
 * @fileName:SimpleAssertTest.java
 */

class SimpleAssertTest {
        public static void main(String args[]) {
                int x = 10;
                assert (x > 10);
                System.out.println("SimpleAssertTest.x...= " + x);
        }
}

/*Output:
        javac SimpleAssertTest.java

        java SimpleAssertTest
```

```
                SimpleAssertTest.x...= 10

        java -ea SimpleAssertTest
                Exception in thread "main" java.lang.AssertionError

        -ea → enable assertions for every non-system class
*/

/**@author:Venu Kumar.S  @date:Oct 15, 2012
 * @fileName:SimpleAssertTest1.java
 */

class SimpleAssertTest1 {
        public static void main(String args[]) {
                int x = 10;
                assert (x >= 10);
                System.out.println("SimpleAssertTest.x...= " + x);
        }
}

/*Output:
        javac SimpleAssertTest.java

        java SimpleAssertTest
                SimpleAssertTest.x...= 10

        java -ea SimpleAssertTest
                SimpleAssertTest.x...= 10

        -ea → enable assertions for every non-system class
*/
```

2)**Augmented Version:** We can augment some description by using augmented version to the **AssertionError**.

Syntax: **assert(b):description_msg;** where, b → is boolean

description_msg → any description, can be any type, but recommended to use String type.

Ex:

```
/**@author:Venu Kumar.S  @date:Oct 15, 2012
 * @fileName:AugmentAssertTest.java
 */
class AugmentAssertTest {
        public static void main(String args[]) {
                int x = 10;
                assert (x > 10) : "..x value should be >10, but it is not";
                System.out.println(".AugmentAssertTest.x...= "+x);
        }
}

/*Output:
        javac AugmentAssertTest.java
```

```
        java AugmentAssertTest
                .AugmentAssertTest.x...= 10

        java -ea AugmentAssertTest
        Exception in thread "main" java.lang.AssertionError: ..x value should be >10, but it is not

        -ea → enable assertions for every non-system class
*/

/**@author:Venu Kumar.S  @date:Oct 15, 2012
 * @fileName:AugmentAssertTest1.java
 */
class AugmentAssertTest1 {
        public static void main(String args[]) {
                int x = 10;
                assert (x <= 10) : "..x value should be <=10, but it is not";
                System.out.println(".AugmentAssertTest.x...= "+x);
        }
}

/*Output:
        javac AugmentAssertTest1.java
        java AugmentAssertTest1
                .AugmentAssertTest1.x...= 10

        java -ea AugmentAssertTest1
                .AugmentAssertTest1.x...= 10

        -ea → enable assertions for every non-system class
*/
```

→ **Syntax: assert(e1):e2;**

→ **e2** will be evaluated iff **e1** is false. i.e ., if **e1** is true, then **e2** won't be evaluated.

```
/**@author:Venu Kumar.S  @date:Oct 15, 2012
 * @fileName:AugmentAssertTest2.java
 */
class AugmentAssertTest2 {
        public static void main(String args[]) {
                int x = 10;
                assert (x > 10) : ++x;
                System.out.println(x);
        }
}

/*Output:
        javac AugmentAssertTest2.java
        java AugmentAssertTest2
                .AugmentAssertTest2.x...= 10
        java -ea AugmentAssertTest2
                Exception in thread "main" java.lang.AssertionError: 11
        -ea → enable assertions for every non-system class
```

```
*/
```

→ **assert(e1):e2;**

→ In the place **e2** we can take a method call also, but void type method calls are not allowed, if **e2** return type is void, then we will get **CE:"void type not allowed here"**.

```
/**@author:Venu Kumar.S  @date:Oct 15, 2012
 * @fileName:AugmentAssertTest3.java
 */
class AugmentAssertTest3 {
	public static void main(String args[]) {
		int x = 10;
		assert (x > 10) : m1();
		System.out.println(".AugmentAssertTest3.x...="+x);
	}

	public static int m1() {
		return 9999;
	}
}

/*Output:
	javac AugmentAssertTest3.java

	java AugmentAssertTest3
		.AugmentAssertTest3.x...= 10

	java -ea AugmentAssertTest3
		Exception in thread "main" java.lang.AssertionError: 9999

   -ea → enable assertions for every non-system class
*/
```

Inappropriate usage of assert():

1) It is always inappropriate to mix programming logic with assert statement, because there is no guarantee of execution of assert statement at runtime.

Ex: inappropriate way

```
	public static void withdraw(int x) {
			assert (x < 100);
		}
```

Ex: appropriate way

```
	public static void withdraw(int x) {
			if(x<100){
				throw new IllegalArgumentException;
			}
	}
```

2) It is always inappropriate to use assertions for validating ***public method*** arguments.
It is always *inappropriate* to use *assertions* for validating command-line arguments, because

these are arguments to ***public main()***.

Appropriate way to use assert():

1) In our program if there is any place where the control not allowed to reach then it is the best place to use assert statement.

Ex:

```
switch(x) {
        case 1: System.out.println("One");
                break;

        case 2: System.out.println("Two");
                break;

        case 3: System.out.println("Three");
                break;

        case 4: System.out.println("Four");
                break;
        default: assert(false); //RE: AssertionException
}
```

2) It is always appropriate to use assertions for validating private method arguments.

AssertionError:

→ It is the child class of Error, hence it is unchecked.

→ It is legal to catch **AssertionError** by using **try-catch** block, but it is waste kind of activity.

Ex:

```
/**@author:Venu Kumar.S  @date:Oct 15, 2012
 * @fileName:AssertErrorTest.java
 */
        class AssertErrorTest {
        public static void main(String args[]) {
                int x = 10;
                try {
                        assert (x > 10);
                } catch (AssertionError e) {
System.out.println("..It waste kind of activity.. Iam catching AssertionError.. ");
                }
                System.out.println(".AssertErrorTest.x..= "+x);
        }
}

/*Output:
        javac AssertErrorTest.java

        java  AssertErrorTest
                .AssertErrorTest.x..= 10
```

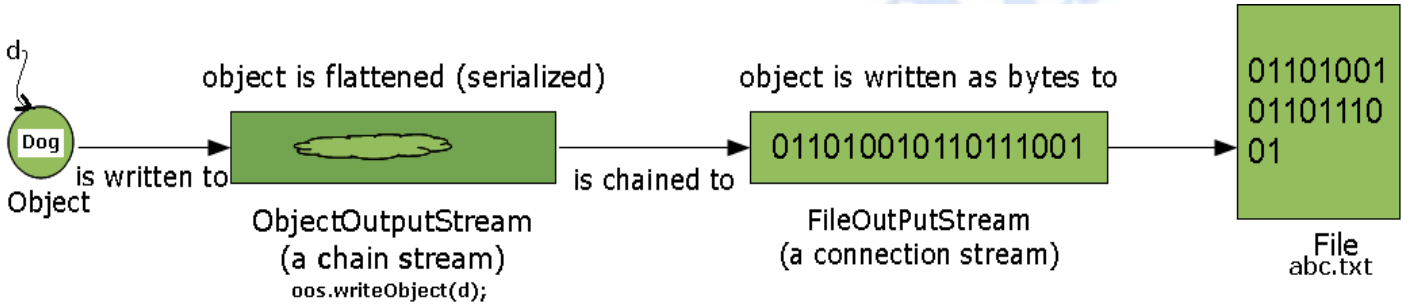
```
        java -ea  AssertErrorTest
                ..It waste kind of activity.. Iam catching AssertionError..
        .AssertErrorTest.x..= 10

    -ea → enable assertions for every non-system class
*/
```

Serialization & DeSerialization

Serialization:

→ The process of writing state of an object to a file is called Serialization. But, it is a process of converting an object from java supported form to either file supported form or network supported form.

→ By using "**FileOutputStream**" and "**ObjectOutputStream**" classes we can achieve Serialization.



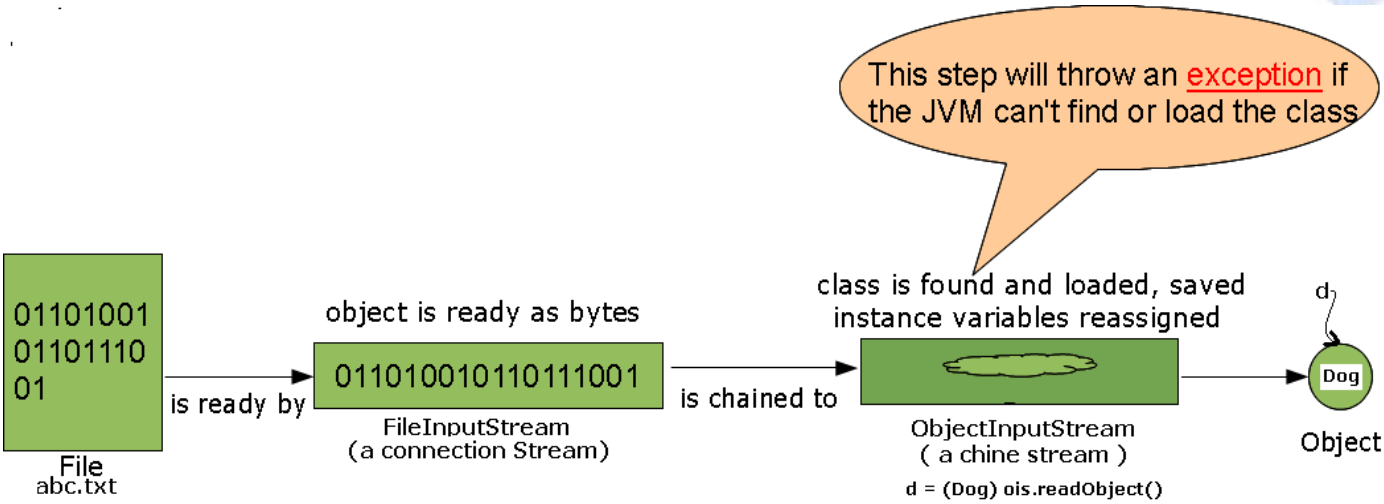
Ex: Serialization:

```
import java.io.*
class Dog implements Serializable {
        int i = 10;
        int j = 20;
}

class SerializationDemo {
        public static void main(String args[]) throws IOException {
                Dog d = new Dog();

                // Serialization
                try {
                        FileOutputStream fos = new FileOutputStream("abc.txt");
                        ObjectOutputStream oos = new ObjectOutputStream(fos);
                        oos.writeObject(d);
                        oos.close();
                        fos.close();
                } catch (IOException i) {
                        i.printStackTrace();
                }
        }
}
```

Deserialization:

→ The process of reading state of an object from a file is called Deserialization. But, it is the process of converting an object from either network supported form or File supported form to java supported form.

→ By using "**FileInputStream**" and "**ObjectInputStream**" classes we can achieve De-Serialization.



Ex: DeSerialization:

```
import java.io.*;
        class Dog implements Serializable {
                int i = 10;
                int j = 20;
        }

        class DeSerializationDemo {
                public static void main(String args[]) throws IOException,
                                ClassNotFoundException {
                        Dog d = null;

                        // De-Serialization
                        try {
                                FileInputStream fis = new FileInputStream("abc.txt");
                                ObjectInputStream ois = new ObjectInputStream(fis);
                                d = (Dog) ois.readObject();
                                ois.close();
                                fis.close();
                        } catch (IOException i) {
                                i.printStackTrace();
                                return;
                        } catch (ClassNotFoundException c) {
                                System.out.println("Dog class not found");
                                c.printStackTrace();
                                return;
                        }
                        System.out.println(d.i + "...." + d.j);
```

```
		}
	}
```

Program:

```
import java.io.*;

class Dog implements Serializable {
	int i = 10;
	int j = 20;
}

class SerializableDemo {
	public static void main(String args[]) throws Exception {
		Dog d1 = new Dog();

		// Serialization

		FileOutputStream fos = new FileOutputStream("abc.txt");
		ObjectOutputStream oos = new ObjectOutputStream(fos);
		oos.writeObject(d1);
		oos.flush();
		oos.close();
		// De-Serialization

		FileInputStream fis = new FileInputStream("abc.txt");
		ObjectInputStream ois = new ObjectInputStream(fis);
		Dog d2 = (Dog) ois.readObject();
		ois.close();
		System.out.println(d2.i + "...." + d2.j);

	}
}

/*OUTPUT:
 10....20
 */
```

Serialization & De-Serialization Example:

```
import java.io.*;

class Dog implements Serializable {
	int i = 10;
	int j = 20;
}
```

Serialization Example:

```
import java.io.*;
class SerializationDemo {
	public static void main(String args[]) throws IOException {
		Dog d1 = new Dog();
```

```
		// Serialization
		try {
			FileOutputStream fos = new FileOutputStream("abc.ser");
			ObjectOutputStream oos = new ObjectOutputStream(fos);
			oos.writeObject(d1);
			oos.close();
			fos.close();
		} catch (IOException i) {
			i.printStackTrace();
		}
	}
}
```

javac SerializationDemo.java

java SerializationDemo

De-Serialization Example:

```
import java.io.*;

class DeSerializationDemo {
	public static void main(String args[]) throws IOException,
			ClassNotFoundException {
		Dog d2 = null;

		// De-Serialization
		try {
			FileInputStream fis = new FileInputStream("abc.ser");
			ObjectInputStream ois = new ObjectInputStream(fis);
			d2 = (Dog) ois.readObject();
			ois.close();
			fis.close();
		} catch (IOException i) {
			i.printStackTrace();
			return;
		} catch (ClassNotFoundException c) {
			System.out.println("Dog class not found");
			c.printStackTrace();
			return;
		}
		System.out.println(d2.i + "...." + d2.j);
	}
}
```

javac DeSerializationDemo
java DeSerializationDemo
10....20

→ We can perform Serialization only for Serializable objects.

→ An object is said to be Serializable iff the corresponding class implements Serialiazable interface.

→ Serializable interface is a marker interface, which is present in java.io package and doesn't contain any methods.

→ If we are trying to serialize a non-serializable object we will get **RE: NotSerializableException**.

transient keyword:

→ At the time of **Serialization** if we don't want to Serialize the value of a particular variable to meet the security constraint we have to declare those variables with "**transient**" keyword.

→ At the time of **Serialization** JVM ignores original value of transient variable and saves default value.

transient Vs static: → Static variables are not part of object hence they won't participate in Serialization process. Due to this declaring a static variable as transient there is no impact.

transient Vs final: → final variables will be participated into serialization directly by their values, hence declaring a final variable with transient there is no impact.

Ex:

	Declaration	Output
1)	int i=10; int j=20;	10...20
2)	transient int i=10; int j=20;	0...20
3)	transient final int i=10; transient int j=20;	10...0
4)	transient int i=10; transient static int j=20;	0...20

Object Graph in Serialization:

→ Whenever we are trying to Serialize an object the set of all objects which are reachable from that object will be Serialized automatically this group of objects is called "**Object Graph**".

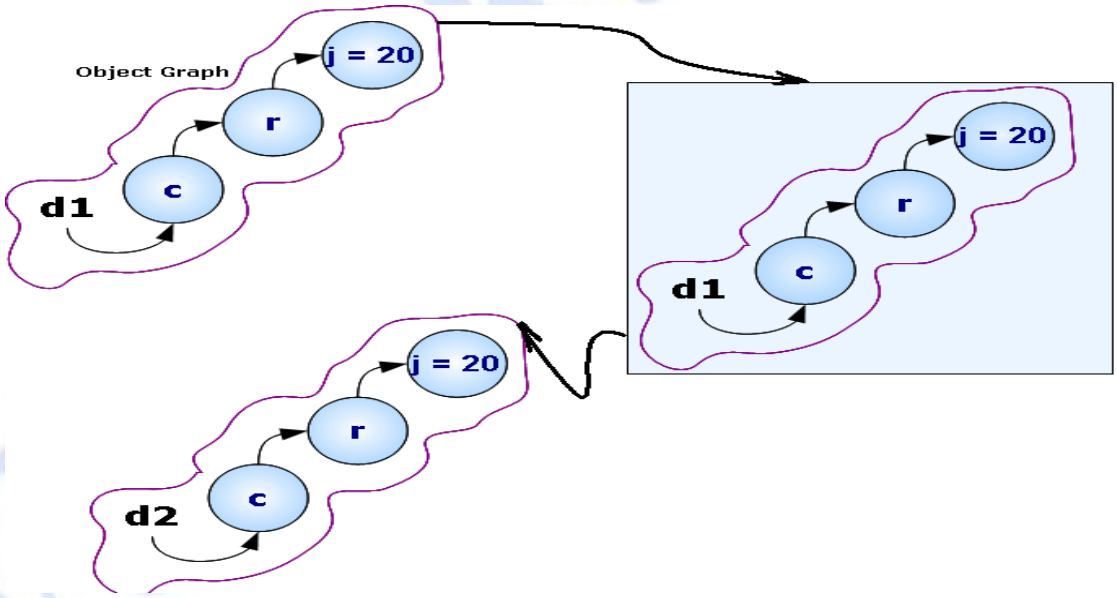
In Object Graph every object should be Serializable otherwise we will get "**SerializableException**".

```
import java.io.*;

class Dog implements Serializable {
        Cat c = new Cat();
}

class Cat implements Serializable {
        Rat r = new Rat();
}

class Rat implements Serializable {
        int j = 20;
```

```
}

class ObjGraphSerializeDemo {
        public static void main(String args[]) throws Exception {
                Dog d1 = new Dog();

                // Serialization

                FileOutputStream fos = new FileOutputStream("abc.ser");
                ObjectOutputStream oos = new ObjectOutputStream(fos);
                oos.writeObject(d1);
                oos.flush();
                oos.close();
                // De-Serialization

                FileInputStream fis = new FileInputStream("abc.ser");
                ObjectInputStream ois = new ObjectInputStream(fis);
                Dog d2 = (Dog) ois.readObject();
                ois.close();
                System.out.println("d2.c.r.j = " + d2.c.r.j); // 20

        }
}

/*OUTPUT:
d2.c.r.j = 20
*/
```

→ From the above program whenever we are Serializing a **Dog** object automatically Cat & Rat objects will be Serialized. Because these are the part of ObjectGraph of Dog.

Among Dog, Cat & Rat, if at least one class is not Serializable then we will get NotSerializable Exception



Customized Serialization:

In the default Serialization there may be a chance of loss of information because of "**transient**" keyword.

```
import java.io.*;
class Account implements Serializable {
        String username = "guru";
        transient String password = "kumar";
}

class SerializeDemo1 {
        public static void main(String args[]) throws Exception {
                Account a1 = new Account();

                System.out.println(a1.username + "...." + a1.password); // guru....kumar
                FileOutputStream fos = new FileOutputStream("abc.ser");
                ObjectOutputStream oos = new ObjectOutputStream(fos);
                oos.writeObject(a1);
                oos.flush();
                oos.close();

                FileInputStream fis = new FileInputStream("abc.ser");
                ObjectInputStream ois = new ObjectInputStream(fis);
                Account a2 = (Account) ois.readObject();
                ois.close();
                System.out.println(a2.username + "...." + a2.password); // guru....null
        }
}


/*OUTPUT:
guru....kumar
guru....null
*/
```

→ From the above example before Serialization Account object can provide proper username & password, but after Deserialization Account object can't provide the Original password. Hence, during default Serialization there may be a chance of loss of information due to "transient" keyword. we can recover this loss of information by using "Customized Serialization".

→ We can implement Customized Serialization by using the following two methods. We have to define these methods in the corresponding class of Serialized object.

1) **private void writeObject(ObjectOutputStream oos) throws IOException** : This method will be executed automatically at the time of Serialization it is a call back method.

2) **private void readObject(ObjectInputStream ois) throws IOException, ClassNotFoundException** : This method will be executed automatically at the time of deserialization it is a call back method.

Program1:

```
import java.io.*;

class Account implements Serializable {
        String username = "guru";
```

```
        transient String password = "kumar";

        private void writeObject(ObjectOutputStream oos) throws Exception {
                System.out.println(Account.class.getName() + ".writeObject");
                oos.writeObject(username); // write username
                oos.writeObject(password); // write password
        }

        private void readObject(ObjectInputStream o) throws IOException,
                        ClassNotFoundException {
                System.out.println(Account.class.getName() + ".readObject");

                username = (String) o.readObject(); // read username
                password = (String) o.readObject(); // read password
        }

}

class CustSerializeDemo {
        public static void main(String args[]) throws Exception {
                Account a1 = new Account();
                System.out.println(a1.username + "...." + a1.password); // guru....kumar
                FileOutputStream fos = new FileOutputStream("abc.ser");
                ObjectOutputStream oos = new ObjectOutputStream(fos);
                oos.writeObject(a1);

                FileInputStream fis = new FileInputStream("abc.ser");
                ObjectInputStream ois = new ObjectInputStream(fis);
                Account a2 = (Account) ois.readObject();
                System.out.println(a2.username + "...." + a2.password); // guru....kumar
        }
}

/*OUTPUT:
 java CustSerializeDemo
guru....kumar
JAVATest.Account.writeObject
JAVATest.Account.readObject
guru....kumar
*/
```

Program2: with validation

```
import java.io.*;

class Account implements Serializable {
        String username = "guru";
        transient String password = "kumar";

        Account(String username, String password) {
                this.username = username;
                this.password = password;
                validate();
        }
```

```
	private void writeObject(ObjectOutputStream oos) throws Exception {
		System.out.println(Account.class.getName() + ".writeObject");
		oos.writeObject(username); // write any obejct here
		oos.writeObject(password);
	}

	private void readObject(ObjectInputStream o) throws IOException,
			ClassNotFoundException {
		System.out.println(Account.class.getName() + ".readObject");

		username = (String) o.readObject();
		password = (String) o.readObject();
		validate();
	}

	private void validate() {
		if (username == null || username.length() == 0 || password == null
				|| password.length() == 0) {

			throw new IllegalArgumentException();
		}
	}
}

class CustSerializeDemo {
	public static void main(String args[]) throws Exception {
		BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

		System.out.println("Enter UserName ...:");
		String username = br.readLine();

		System.out.println("Enter Password ...:");
		String password = br.readLine();

		Account a1 = new Account(username, password);

		System.out.println(a1.username + "...." + a1.password);
		FileOutputStream fos = new FileOutputStream("abc.ser");
		ObjectOutputStream oos = new ObjectOutputStream(fos);
		oos.writeObject(a1);

		oos.flush();
		oos.close();

		FileInputStream fis = new FileInputStream("abc.ser");
		ObjectInputStream ois = new ObjectInputStream(fis);
		Account a2 = (Account) ois.readObject();
		ois.close();
		System.out.println(a2.username + "...." + a2.password);
	}
}
```

Output1:

```
Enter UserName ...:
Guru
Enter Password ...:
Kumar
Guru....Kumar
Account.writeObject
Account.readObject
Guru....Kumar
```

Output2:

```
Enter UserName ...:

Enter Password ...:
kumar
Exception in thread "main" java.lang.IllegalArgumentException
        at JAVATest.Account.validate(y.java:39)
        at JAVATest.Account.<init>(y.java:16)
        at JAVATest.CustSerializeDemo.main(y.java:54)
```

Serialization with respect to Inheritance:

case 1: If the parent class implements Serializable then every child class is by default Serializable i.e., Serializable nature is inheriting from parent to child
(Parent → child).

Ex:

```
class Animal implements Serializable {
        int x = 10;
}

class Dog extends Animal {
        int y = 20;
}
```

→ We can Serialize **Dog** object even though dog class doesn't implement Serializable interface explicitly , because it's parent class Animal is Serializable.

Case 2: Even though parent class doesn't implement Seriazable & if the child is Serializable then we can Serialize child class object. At the time of Serialization JVM ignores the original values of instance variables which are coming from non-Serializable parent & store default values.

→ At the time of deserialization JVM checks is any parent class is non-Serializable or not, JVM creates a separate object for every non-Serializable parent & share its instance variables to the current object.

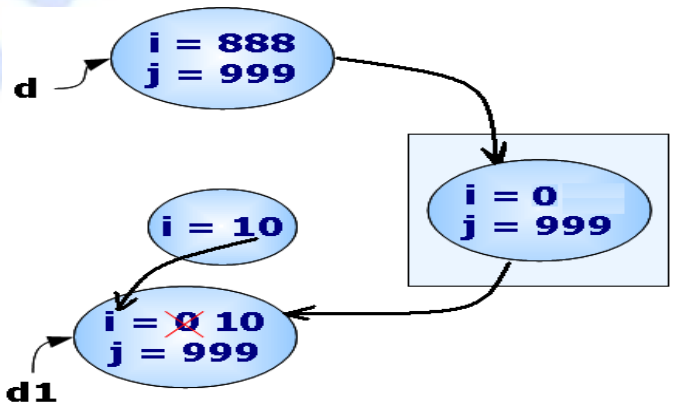
→ For this JVM always calls no argument constructor of the non-Serializable parent. If the non-Serializable doesn't have no argument constructor then we will get RuntimeException.

Ex:

```
import java.io.*;
```

```
class Animal {
	int i = 10;

	Animal() {
		System.out.println("Animal Constructor called..");
	}
}

class Dog extends Animal implements Serializable {
	int j = 20;

	Dog() {
		System.out.println("Dog Constructor called..");
	}
}

class SerializeInheritanceTest {
	public static void main(String args[]) throws Exception {
		Dog d = new Dog();
		d.i = 888;
		d.j = 999;

		FileOutputStream fos = new FileOutputStream("abc.txt");
		ObjectOutputStream oos = new ObjectOutputStream(fos);
		oos.writeObject(d);

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

		FileInputStream fis = new FileInputStream("abc.txt");
		ObjectInputStream ois = new ObjectInputStream(fis);
		Dog d1 = (Dog) ois.readObject();

		System.out.println("d1.i..." + d1.i+"  d1.j..." + d1.j);
	}
}
```

```
/*OUTPUT:
java SerializeInheritanceTest

Animal Constructor called..
Dog Constructor called..
Deserialization started..
Animal Constructor called..
d1.i...10  d1.j...999
*/
```



To get elapsed Time:

```
/**@author:Venu Kumar.S  @date:Oct 5, 2012
 * @fileName:TimeTest1.java
 */
```

```
        class TimeTest1 {
          public static void main(String[] args) {

            long startTime = System.currentTimeMillis();

            long total = 0;
            for (int i = 0; i < 10000000; i++) {
              total += i;
            }


            long stopTime = System.currentTimeMillis();
            long elapsedTime = stopTime - startTime;
            long min = elapsedTime/(60 * 1000);
            long hours = min/60;
            min = min%60;
            long days = hours/24;
            hours = hours%24;

               System.out.println("Elapsed Time in milliSeconds..:" + elapsedTime);
               System.out.println("Elapsed Time in min.." + min);
               System.out.println("Elapsed Time in hours.." + hours);
               System.out.println("Elapsed in days.." + days);


          }
        }

/*
 Output:
        java TimeTest1
        Elapsed Time in milliSeconds..:16
        Elapsed Time in min..0
        Elapsed Time in hours..0
        Elapsed in days..0
*/


        /**@author:Venu Kumar.S  @date:Oct 5, 2012
         * @fileName:MethodExecutionTimer.java
         */
public class MethodExecutionTimer {

        public static void main(String[] args) {

                long start = System.currentTimeMillis();

        System.out.println(".MethodExecutionTimer.heavyMethod().. Execution starts...");

                heavyMethod();

        System.out.println(".MethodExecutionTimer.heavyMethod().. Execution Ends...");

                long end = System.currentTimeMillis();
```

```
		System.out.println("Method execution total time" + " in seconds ==> "
					+ (end - start) / 1000 + " seconds");

	}

	public static void heavyMethod() {

		for (int i = 1; i < 5; i++) {

			try {

			Thread.sleep(i * 1000); // * 1000 makes it second 1*1000 = 1 second

			} catch (InterruptedException ex) {

				ex.printStackTrace();

			}

		}

	}

}

/*
Output:
	.MethodExecutionTimer.heavyMethod().. Execution starts...
	.MethodExecutionTimer.heavyMethod().. Execution Ends...
	Method execution total time in seconds ==> 10 seconds
*/
```

NetWorking Socket Programming

The term network programming refers to writing programs that execute across multiple devices (computers), in which the devices are all connected to each other using a network.

The **java.net** package of the J2SE APIs contains a collection of classes and interfaces that provide the low-level communication details, allowing you to write programs that focus on solving the problem at hand.

The **java.net** package provides support for the two common network protocols:

TCP: TCP stands for Transmission Control Protocol, which allows for reliable communication between two applications. TCP is typically used over the Internet Protocol, which is referred to as TCP/IP.

UDP: UDP stands for User Datagram Protocol, a connection-less protocol that allows for packets of data to be transmitted between applications.

Socket Programming: This is most widely used concept in Networking and it has been explained in

very detail.

URL Processing: This would be covered separately. Click here to learn about URL Processing in Java language.

Socket Programming:

Sockets provide the communication mechanism between two computers using TCP. A client program creates a socket on its end of the communication and attempts to connect that socket to a server.

When the connection is made, the server creates a socket object on its end of the communication. The client and server can now communicate by writing to and reading from the socket.

The **java.net.Socket** class represents a socket, and the java.net.ServerSocket class provides a mechanism for the server program to listen for clients and establish connections with them.

The following steps occur when establishing a **TCP** connection between two computers using sockets:

The server instantiates a **ServerSocket object**, denoting which port number communication is to occur on.

The server invokes the **accept()** method of the ServerSocket class. This method waits until a client connects to the server on the given port.

After the **server is waiting**, a **client instantiates a Socket object**, specifying the server name and port number to connect to.

The constructor of the Socket class attempts to connect the client to the specified server and port number. If communication is established, the client now has a Socket object capable of communicating with the server.

On the **server side, the accept()** method returns a reference to a new socket on the server that is connected to the client's socket.

After the connections are established, communication can occur using I/O streams. Each socket has both an OutputStream and an InputStream. The client's OutputStream is connected to the server's InputStream, and the client's InputStream is connected to the server's OutputStream.

TCP is a **two-way communication protocol**, so data can be sent across both streams at the same time. There are following usefull classes providing complete set of methods to implement sockets.

ServerSocket Class Methods:

The **java.net.ServerSocket** class is used by server applications to obtain a port and listen for client requests

The ServerSocket class has four constructors:

1. **public ServerSocket(int port) throws IOException :**Attempts to create a server socket bound to the specified port. An exception occurs if the port is already bound by another application.

2. **public ServerSocket(int port, int backlog) throws IOException** :Similar to the previous constructor, the backlog parameter specifies how many incoming clients to store in a wait queue.

3.**public ServerSocket(int port, int backlog, InetAddress address) throws IOException :** Similar to the previous constructor, the InetAddress parameter specifies the local IP address to bind to.

The InetAddress is used for servers that may have multiple IP addresses, allowing the server to specify which of its IP addresses to accept client requests on.

4.**public ServerSocket() throws IOException :**Creates an unbound server socket. When using this constructor, use the bind() method when you are ready to bind the server socket
If the ServerSocket constructor does not throw an exception, it means that your application has successfully bound to the specified port and is ready for client requests.

The common methods of the ServerSocket class:

1.**public int getLocalPort()** : Returns the port that the server socket is listening on. This method is useful if you passed in 0 as the port number in a constructor and let the server find a port for you.

2.**public Socket accept() throws IOException :**Waits for an incoming client. This method blocks until either a client connects to the server on the specified port or the socket times out, assuming that the time-out value has been set using the setSoTimeout() method. Otherwise, this method blocks indefinitely.

3.**public void setSoTimeout(int timeout) :** Sets the time-out value for how long the server socket waits for a client during the accept().

4.**public void bind(SocketAddress host, int backlog):**Binds the socket to the specified server and port in the SocketAddress object. Use this method if you instantiated the ServerSocket using the no-argument constructor.

When the ServerSocket invokes **accept()**, the method does not return until a client connects. After a client does connect, the ServerSocket creates a new Socket on an unspecified port and returns a reference to this new Socket. A TCP connection now exists between the client and server, and communication can begin.

Socket Class Methods:

The **java.net.Socket** class represents the socket that both the client and server use to communicate with each other. The client obtains a Socket object by instantiating one, whereas the server obtains a Socket object from the return value of the accept() method.

The Socket class has five constructors that a client uses to connect to a server:

1.**public Socket(String host, int port) throws UnknownHostException, IOException :**
This method attempts to connect to the specified server at the specified port. If this constructor does not throw an exception, the connection is successful and the client is connected to the server.

2.**public Socket(InetAddress host, int port) throws IOException :**
This method is identical to the previous constructor, except that the host is denoted by an InetAddress object.

3.**public Socket(String host, int port, InetAddress localAddress, int localPort) throws IOException :**Connects to the specified host and port, creating a socket on the local host at the specified address and port.

4.**public Socket(InetAddress host, int port, InetAddress localAddress, int localPort) throws IOException:** This method is identical to the previous constructor, except that the host is denoted by an InetAddress object instead of a String
5.**public Socket() :**Creates an unconnected socket. Use the connect() method to connect this socket to a server.

When the Socket constructor returns, it does not simply instantiate a Socket object but it actually attempts to connect to the specified server and port.

Some methods of interest in the Socket class are listed here. Notice that both the client and server have a Socket object, so these methods can be invoked by both the client and server.

1.**public void connect(SocketAddress host, int timeout) throws IOException**
This method connects the socket to the specified host. This method is needed only when you instantiated the Socket using the no-argument constructor.

2.**public InetAddress getInetAddress() :**This method returns the address of the other computer that this socket is connected to.

3.**public int getPort() :**Returns the port the socket is bound to on the remote machine.

4.**public int getLocalPort():** Returns the port the socket is bound to on the local machine.

5.**public SocketAddress getRemoteSocketAddress():**Returns the address of the remote socket.

6.**public InputStream getInputStream() throws IOException :**
Returns the input stream of the socket. The input stream is connected to the output stream of the remote socket.

7.**public OutputStream getOutputStream() throws IOException :**Returns the output stream of the socket. The output stream is connected to the input stream of the remote socket

8.**public void close() throws IOException :**Closes the socket, which makes this Socket object no longer capable of connecting again to any server InetAddress Class Methods:
This class represents an Internet Protocol (IP) address. Here are following useful methods which you would need while doing socket programming:

1.**static InetAddress getByAddress(byte[] addr) :** Returns an InetAddress object given the raw IP address.

2.**static InetAddress getByAddress(String host, byte[] addr) :**Create an InetAddress based on the provided host name and IP address.

3.**static InetAddress getByName(String host) :**Determines the IP address of a host, given the host's name.

4.**String getHostAddress()** : Returns the IP address string in textual presentation.

5.**String getHostName()** :Gets the host name for this IP address.

6.**static InetAddress InetAddress getLocalHost():**Returns the local host.

7.**String toString() :**Converts this IP address to a String.

Socket Client Example:

The following GreetingClient is a client program that connects to a server by using a socket and sends a greeting, and then waits for a response.

// **GreetingClient.java**

```
import java.net.*;
import java.io.*;

public class GreetingServer extends Thread {
	private ServerSocket serverSocket;

	public GreetingServer(int port) throws IOException {
		serverSocket = new ServerSocket(port);
		serverSocket.setSoTimeout(10000);
	}

	public void run() {
		while (true) {
			try {
				System.out.println("Waiting for client on port "
					+ serverSocket.getLocalPort() + "...");
				Socket server = serverSocket.accept();
				System.out.println("Just connected to "
					+ server.getRemoteSocketAddress());
				DataInputStream in = new DataInputStream(server
					.getInputStream());
				System.out.println(in.readUTF());
				DataOutputStream out = new DataOutputStream(server
					.getOutputStream());
				out.writeUTF("Thank you for connecting to "
					+ server.getLocalSocketAddress() + "\nGoodbye!");
				server.close();
			} catch (SocketTimeoutException s) {
				System.out.println("Socket timed out!");
				break;
			} catch (IOException e) {
				e.printStackTrace();
				break;
			}
		}
	}

	public static void main(String[] args) {
		int port = Integer.parseInt(args[0]);
		try {
			Thread t = new GreetingServer(port);
			t.start();
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
}
```

Compile client and server and then start server as follows:

```
java GreetingServer 6066
```

Waiting for client on port 6066...

Check client program as follows:

java GreetingClient localhost 6066

Connecting to localhost on port 6066
Just connected to localhost/127.0.0.1:6066
Server says Thank you for connecting to /127.0.0.1:6066
Goodbye!

||Jai Hind||