So Called

CORE JAVA

- Satya Kaveti

Small Codes

Programming Simplified

A SmlCodes.Com Small presentation

In Association with Idleposts.com

For more tutorials & Articles visit SmlCodes.com

So called CORE JAVA

If you discover any errors on our website or in this tutorial, please notify us at support@smlcodes.com or smlcodes@gmail.com

First published on Aug 2016, Published by **SmlCodes.com**

Author Credits

Name : **Satya Kaveti**

Email : satyakaveti@gmail.com

Website : smlcodes.com, satyajohnny.blogspot.com

Digital Partners

Table of Content

I.Introduction to Java

Java is a platform independent programming language which is introduced by **James Gosling** and his team mates in the year 1991.

First they want to develop programming language for the **Setup boxes and small embedded systems** in the year of 1991.they named it as "**Green talk",** because the file extension is '.**gt'**. After that they renamed as **"Oak",** it's a tree name**.** But they faced some trademark issues in 1995 they renamed it as "Java"

The first beta version of java released in 1995.

1.1 History of Java

JDK Alpha and Beta (1995)

JDK 1.0 (23rd Jan, 1996) -

JDK 1.1 (19th Feb, 1997)

- AWT event model
- Inner classes
- JavaBeans
- JDBC
- RMI,Reflection
- JIT (Just In Time) compiler for Windows

J2SE 1.2 (8th Dec, 1998) – Playground

- **strictfp** keyword
- Swing graphical API
- Sun's JVM was equipped with a JIT compiler for the first time
- Java plug-in
- Collections framework

J2SE 1.3 (8th May, 2000) - Kestrel

- HotSpot JVM
- Java Naming and Directory Interface (JNDI)
- Java Platform Debugger Architecture (JPDA)
- JavaSound
- Synthetic proxy classes

J2SE 1.4 (6th Feb, 2002) – Merlin

- assert keyword
- Regular expressions
- Exception chaining
- Internet Protocol version 6 (IPv6) support
- New I/O; NIO
- Logging API
- Image I/O API
- Integrated XML parser and XSLT processor (JAXP)
- Integrated security and cryptography extensions (JCE, JSSE, JAAS)
- Java Web Start
- Preferences API (java.util.prefs)

J2SE 5.0 (30th Sep, 2004) – Tiger

- Generics
- Annotations
- Autoboxing/unboxing
- Enumerations
- Varargs
- Enhanced for each loop
- Static imports
- New concurrency utilities in java.util.concurrent
- Scanner class for parsing data from various input streams and buffers.

Java SE 6 (11th Dec, 2006) – Mustang

- Scripting Language Support
- Performance improvements
- JAX-WS
- JDBC 4.0
- Java Compiler API
- JAXB 2.0 and StAX parser
- Pluggable annotations
- New GC algorithms

Java SE 7 (28th July, 2011) – Dolphin

- JVM support for dynamic languages
- Compressed 64-bit pointers
- Strings in switch
- Automatic resource management in try-statement

- The diamond operator
- Simplified varargs method declaration
- Binary integer literals
- Underscores in numeric literals
- Improved exception handling
- ForkJoin Framework
- NIO 2.0 having support for multiple file systems, file metadata and symbolic links
- WatchService
- Timsort is used to sort collections and arrays of objects instead of merge sort
- APIs for the graphics features
- Support for new network protocols, including SCTP and Sockets Direct Protocol

Java SE 8 (18th March, 2014) - Code name culture dropped

- Lambda expression support in APIs
- Functional interface and default methods
- Optionals
- Nashorn – JavaScript runtime which allows developers to embed JavaScript code within applications
- Annotation on Java Types
- Unsigned Integer Arithmetic
- Repeating annotations
- New Date and Time API
- Statically-linked JNI libraries
- Launch JavaFX applications from jar files
- Remove the permanent generation from GC

Java SE 9 Expected: September 22, 2016

- Support for multi-gigabyte heaps
- Better native code integration
- Self-tuning JVM
- Java Module System
- Money and Currency API
- jshell: The Java Shell
- Automatic parallelization

1.2 Features of Java

1. Simple – No Pointers Dude!

Compare with previous Object oriented language C++ they removed many confusing and/or rarely-used features e.g., explicit pointers, operator overloading etc.So now no confusions, clean syntax makes java as **Simple**

2. Object-oriented – All about java Basics

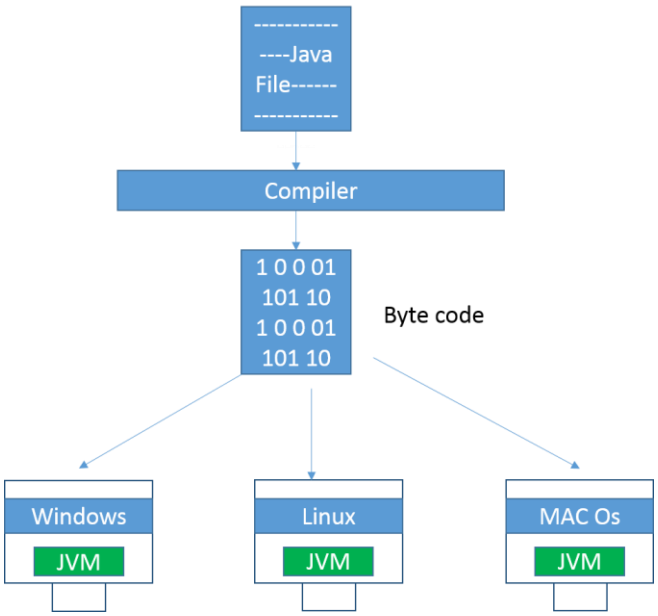
Java based on OOP.below are concepts of OOPs are:

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

3. Platform Independent – OS doesn't matter!

A platform is the hardware or software environment in which a program runs. There are two types of platforms software-based and hardware-based. Java provides software-based platform.

Java code can be run on multiple platforms e.g.Windows,Linux,Sun Solaris,Mac/OS etc. Java code is compiled by the compiler and converted into bytecode.This bytecode is a platform independent code because it can be run on multiple platforms



4. Secured – U can Hack OS, but you can't hack Java Byte code

The Java platform is designed with security features built into the language and runtime system such as static type-checking at compile time and runtime checking (security manager), which let you creating applications that can't be invaded from outside. You never hear about viruses attacking Java applications.

5. Robust – Strong, Error Free always

Robust simply means strong. Java uses strong memory management. There are lack of pointers that avoids security problem. There is automatic garbage collection in java. There is exception handling and type checking mechanism in java. All these points makes java robust.

6. Architecture-neutral – 64-bit, 32-bit, xxx-bit doesn't matter I will work

The language like JAVA can run on any of the processor irrespective of their architecture and vendor

7. Portable

We may carry the java bytecode to any platform.

8. High-performance

Java is faster than traditional interpretation since byte code is "close" to native code still somewhat slower than a compiled language (e.g., C++)

9. Distributed

We can create distributed applications in java. RMI and EJB are used for creating distributed applications. We may access files by calling the methods from any machine on the internet.

10. Multi-threaded

A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it shares the same memory. Threads are important for multi-media, Web applications etc.

1.3 Say "Hello" to Java

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

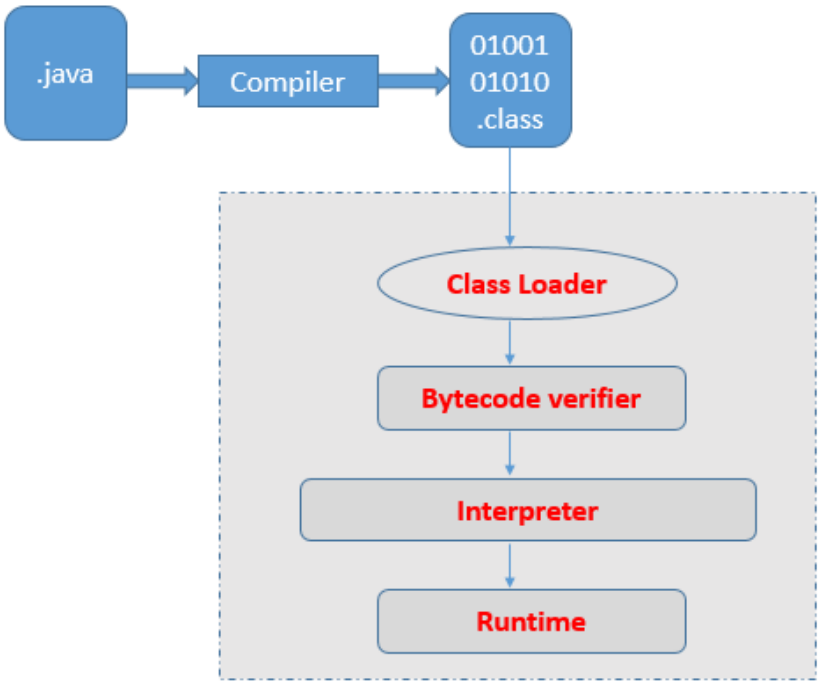
Output

```
Hello Java
```

How it Works

Things needs to understand

- **class** keyword is used to declare a class in java.
- **public** keyword is an access modifier, it means it is visible to all.
- **static** is a keyword, The main method is executed by the JVM, so it doesn't require to create object to invoke the main method. So it saves memory.
- **void** it means it doesn't return any value.
- **main** represents startup of the program.
- **String[] args** is used for command line argument.



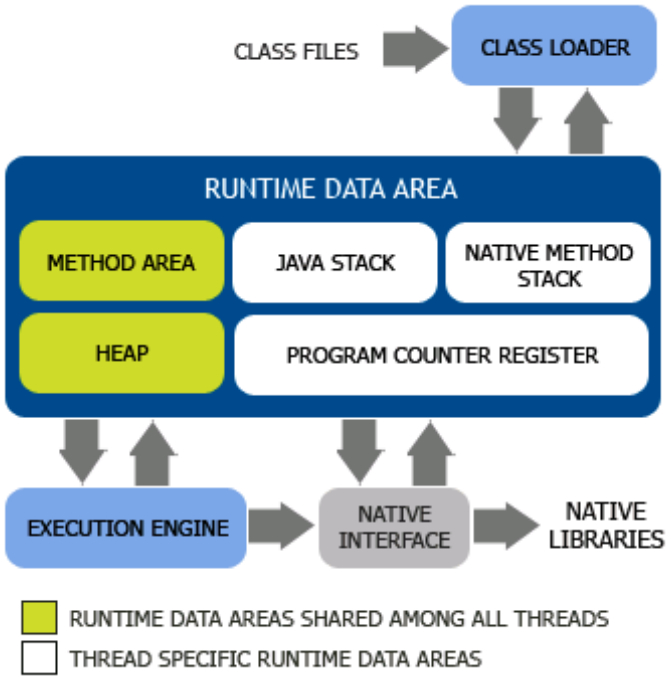
Java complier will convert Java source code to Byte code

Classloader: is the subsystem of JVM that is used to load class files.

Bytecode Verifier: checks the code fragments for illegal code that can violate access right to objects.

Interpreter: read bytecode stream then execute the instructions

1.4 JVM Architecture



1) **Classloader:** load class files.

Runtime Area

2) **Method Area:** per-class area. Constant pool, field and method data, the code for methods.

3) **Heap:** It is the runtime data area in which objects are allocated.

4) **Stack:** It holds local variables and partial results, and plays a part in method invocation and return. Each thread has a private JVM stack, created at the same time as thread. A new frame is created each time a method is invoked. A frame is destroyed when its method invocation completes.

5) **Program Counter Register:** contains the address of the Java virtual machine instruction currently being executed.

6) Native Method Stack: contains all the native methods used in the application.

Execution Engine

1) A virtual processor

2) **Interpreter:** Read bytecode stream then execute the instructions.

3) **Just-In-Time(JIT) compiler** :It is used to improve the performance.JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation.

1.5 JVM, JRE, JDK Differences

JVM : It's a Specification

JRE : Practical implementation of JVM

JDK : JRE + Development Tools

1.6 Interview Questions

Q1) No. of ways to write main () method in java?

- ☑ public static void main(String[] args)
- ☑ public static void main(String []args)
- ☑ public static void main(String args[])
- ☑ public static void main(String... args)
- ☑ static public void main(String[] args)
- ☑ public static final void main(String[] args)
- ☑ final public static void main(String[] args)
- ☑ final strictfp public static void main(String[] args)

Below are invalid

- ☒ public void main(String[] args)
- ☒ static void main(String[] args)
- ☒ public void static main(String[] args)
- ☒ abstract public static void main(String[] args)

Q2) Can We Overload main() method?

☑ YES ☑ Compile ☑ Execute

Yes, We can overload main() method. A Java class can have any number of main() methods. But to run the java class, class should have main() method with signature as "public static void main(String[] args)". If you do any modification to this signature, compilation will be successful. But, you can't run the java program. You will get run time error as main method not found

```
public class OverloadMain {
	public static void main(String[] args) {
		System.out.println("Execution starts from this method");
	}
	void main(int args) {
		System.out.println("Another main method");
	}
	double main(int i, double d) {
		System.out.println("Another main method");
		return d;
	}

}
```

```
Output
Execution starts from this method
```

Q3) Can we declare main() method as private or protected or no access modifier?

☒ NO ☑ Compile ☒ Execute → ***main() method can't accessible to JVM***.

No, main() method must be public. You can't define main() method as private or protected or with no access modifier.

This is because to make the main() method accessible to JVM. If you define main() method other than public, compilation will be successful but you will get run time error as no main method found.

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

```
Output
Error: Main method not found in class intro1.OverloadMain, please define the main method as:
   public static void main(String[] args)
```

Can We Declare main() Method As Non-Static?

☒ NO ☑ Compile ☒ Execute → ***JVM unable to call main method.***

main() method must be declared as static, so that JVM can call main() method without instantiating it's class. If you remove 'static' from main() method signature, compilation will be successful but program fails at run time.

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

Output

```
D:\demo>java Hello
Error: Main method is not static in class Hello, please define the main method as:
   public static void main(String[] args)
```

Can we change return type of main() method?

☒ NO ☑ Compile ☒ Execute → ***Overload fine, removing Not Fine***

```
D:\demo>java Hello
Error: Main method is not static in class Hello, please define the main method as:
   public static void main(String[] args)
```

Can we run java class without main() method? If I use static block in place of main?

☒ NO ☑ Compile ☒ Execute → ***JVM will searches only main() not static***

```
Error: Main method not found in class Hello, please define the main method as:
   public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application
```

Few more on main()

- The main() method is a static method
- You can overload main() method in Java.
- You cannot override main() method in Java
- You can make the **main method final** in Java
- You can make the main method synchronized in Java.
- You cannot call a non-static method from main in Java.

Is JVM, a compiler or interpretor ?

Ans. It's an interpreter.

What are various types of Class loaders used by JVM ?

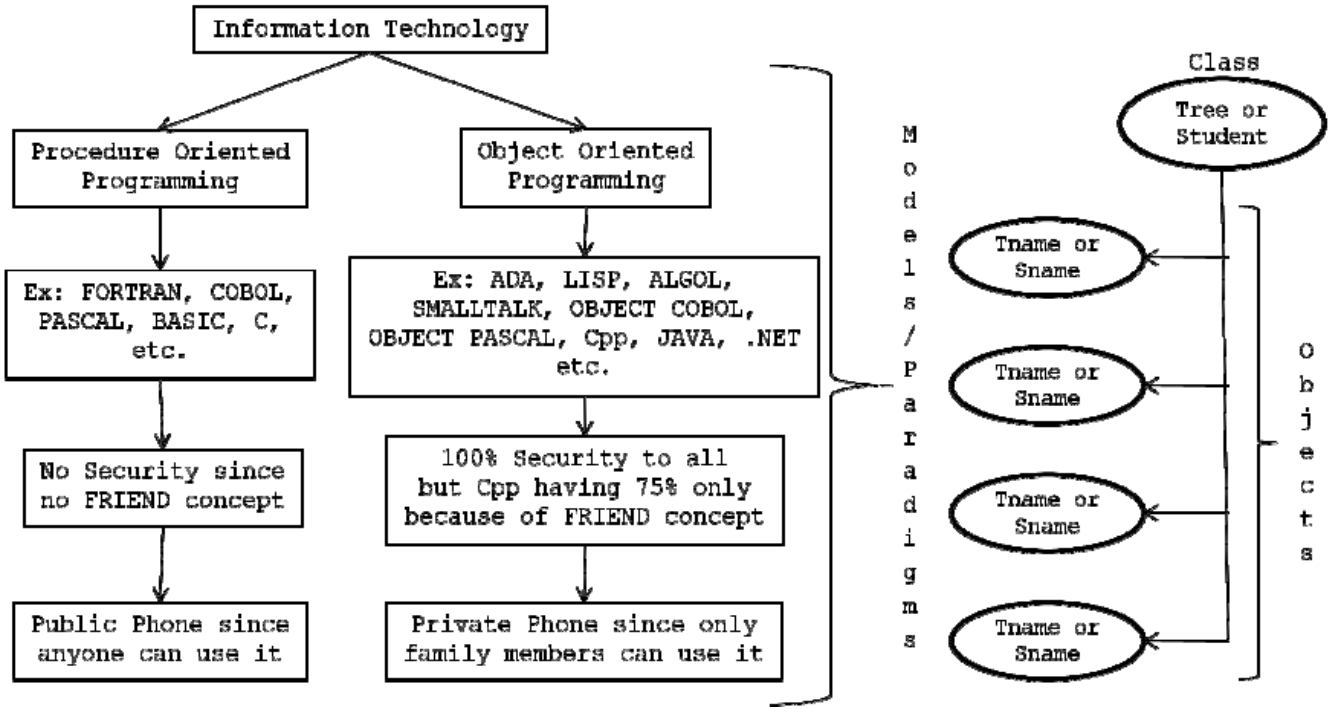
Bootstrap - Loads JDK internal classes, java.* packages.

Extensions - Loads jar files from JDK extensions directory - usually lib/ext directory of the JRE

System - Loads classes from system classpath.

II.Java Object Oriented Concepts

We have two types of programming models are available. They are **procedure oriented** programming language and **object oriented** programming language.



When we represent the data in object oriented programming language we get the security

2.1 The 7 OOP's concepts are

1. **Class**
2. **Object**
3. **Inheritance**
4. **Polymorphism**
5. **Abstraction**
6. **Encapsulation**
7. **Dynamic Binding**

1. Class

A class is the Template/blueprint from which individual objects are created.

"A class is a way of binding the data and associated methods in a single unit".

Class Contains Five elements:-

Class
{

1. **Variables**
2. **Static blocks**
3. **Instance blocks**
4. **Constructors**
5. **Methods**

}

```
package oops2;

public class Student {
	int sno;
	String name;

	public void getData() {
		System.out.println("S.No : " + sno + "\n Student: " + name);
	}
}
```

- Whenever we define a class there is **no memory space** for data members of the class.
- Memory Space will be created for the data members of the class when we create object
- Memory space for the **data members** will be creating on **Heap memory** (Dynamic memory).
- Memory space for **methods** will be creating on **stack memory** (that too when we call the methods).
- All **constants** are available in **associative memory** (retrieving data from Associative memory is negligible).

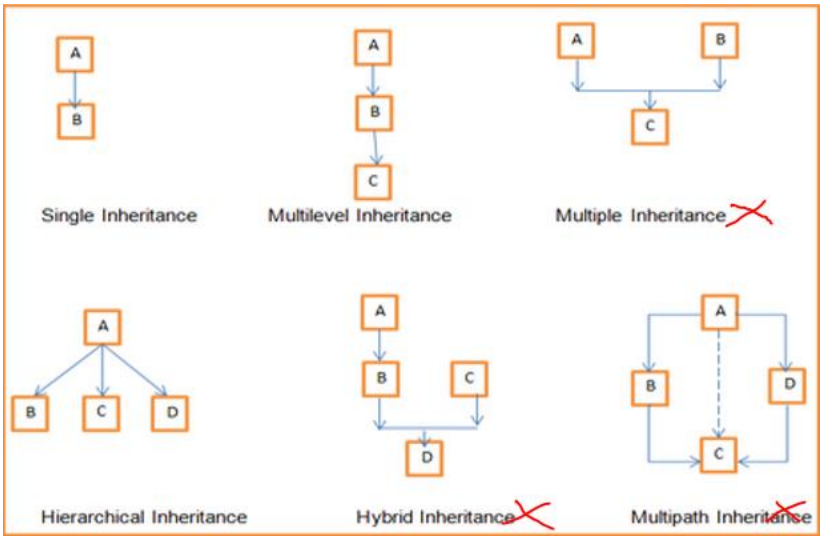
2. Object

Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors - wagging, barking, and eating. An object is an instance of a class. In order to create a memory space in JAVA we must use an operator called **new**.

This **new** operator is known as dynamic memory allocation operator

3. Inheritance

When one object acquires all the properties and behaviors of parent object i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.



4. Polymorphism

When one task is performed by different ways i.e. known as polymorphism. For example: to convense the customer differently, to draw something e.g. shape or rectangle etc.

"Method overloading and method overriding to achieve polymorphism".

1. Abstraction

Hiding internal details and showing functionality is known as abstraction.

"Abstract class and interface to achieve abstraction".

2. Encapsulation

Binding (or wrapping) code and data together into a single unit is known as encapsulation.

A java class is the example of encapsulation.

```java
class student{
	private int sno;
	private String name;
	public int getSno() {
		return sno;
	}
	public void setSno(int sno) {
		this.sno = sno;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
}

public class Main {
	public static void main(String[] args) {
		student ob = new student();
		ob.setName("SATYA"); //Binding code and Data Together
		ob.setSno(101);
		System.out.println(ob.getSno()+" : "+ob.getName());
	}
}
```

```
101 : SATYA
```

3. Dynamic Binding

Dynamic binding is a mechanism of binding an appropriate version of a derived class which is inherited from base class with base class object

2.2 Data types

Type Name	Description	Size	Range	Sample Declaration & Initialization
boolean	true or false	1 bit	{true, false}	boolean myBool = true;
char	Unicode Character	2 bytes	\u0000 to \uFFFF	char myChar = 'a';
byte	Signed Integer	1 byte	-128 to 127	int myInt = 100;
short	Signed Integer	2 bytes	-32768 to 32767	short myShort = 1000;
int	Signed Integer	4 bytes	-2147483648 to 2147483647	int myInt = 100000;
long	Signed Integer	8 bytes	-9223372036854775808 to 9223372036854775807	long myLong = 0;
float	IEEE 754 floating point	4 bytes	±1.4E-45 to ±3.4028235E+38	float myFloat = 10.0f;
double	IEEE 754 floating point	8 bytes	±4.9E-324 to ±1.7976931348623157E+308	double myDouble = 20.0;

2.3 Variables

Whenever we develop any JAVA program that will be developed with respect to class only. In a class we can use 'n' number of data members and 'n' number of methods.

In JAVA, we can use two types of data members or variables. They are

1. **Local variables**
2. **Instance variables**
3. **Static variables**

1. Local

- Local variables are declared in methods, constructors, or blocks.
- Access modifiers (public, private etc.) cannot be used for local variables.
- **No default value** for local variables, they should be initialized before use

2. Instance Variables

"One Copy for each Object"

- Non-static variables will have one copy each per object. Each instance of a class will have one copy of non-static variables.
- Instance variables can be **accessed only by the instance methods**.
- **Instance variables are allocated at compile time**

3. Class/static variables

"Only one copy for class"

- A static variable is associated with the class has only one copy per class but not for each object. An instance of a class does not have static variables.
- Static variables can be **accessed by static or instance methods**
- **Memory is allocated when the class is loaded in context area at run time**

2.4 Final Keyword

"Constant is an identifier whose value cannot be changed during execution of the program".

In JAVA to make the identifiers are as constants, we use a keyword called **final**.Final is a keyword which is playing an important role in three levels. They are at

1. ***Variable level*** **:** cannot change its value once it is initialized.
2. ***Method level*** **:** cannot be overridden in the sub class.
3. ***Class level*** **:** cannot be extended

1. Variable Level

If you make any variable as final, **you cannot change the value of final variable** (It will be constant)

```
public class FinalDemo {

	final int a = 100;

	void data() {
		a = 200;
		System.out.println("a : " + a);
	}

	public static void main(String[] args) {
		FinalDemo demo = new FinalDemo();
		demo.data();
	}

}
```

```
Output  [COMPLE TIME ERROR]
---------------
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
	The final field FinalDemo.a cannot be assigned
```

2. Method Level

If you make any method as final, ***you cannot override it***

```
class A {
	public final void get() {
		System.out.println("A Class Final Method");
	}
}

public class FinalatMethod extends A {

	public void get() {
		System.out.println("Im Override...");
	}

	public static void main(String[] args) {
		new FinalatMethod().get();
	}
}
```

```
Output {COMIPLE TIME ERROR}
Exception in thread "main" java.lang.VerifyError: class final3.FinalatMethod overrides final method
get.()V
```

3. Class Level

If you make any class as final, you cannot extend it. That means **Inheritance not Possible**

```
final class A {
	public final void get() {
		System.out.println("A Class Final Mehod");
	}
}

public class FinalatMethod extends A {

	public void get() {
		System.out.println("Im Override...");
	}

	public static void main(String[] args) {
		new FinalatMethod().get();
	}
}
Output : The type FinalatMethod cannot subclass the final class A
```

More on Final Keyword

1. Is final method inherited?

Ans) ***Yes***, final method is inherited but **you cannot override it**

2. What is blank or uninitialized final variable, Can we initialize blank final variable?

We can initialize blank final variable, but only in constructor

```
public class FinalDemo {
    final int a;
    public FinalDemo() {
    a =100;
    }
}
```

3. Static blank final variable, how can we initialize?

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block

```
public class FinalDemo {
	static final int a;
	static {
	a =100;
	}
}
```

4. What is final parameter, can we change the value of it?

If you declare any parameter as final, you cannot change the value of it.

```
public class FinalDemo {
        int cube(final int n){
                n=n+2;//can't be changed as n is final
                n*n*n;
               }
}
```

5. Can we declare a constructor final?

No, because constructor is never inherited.

6. Can interface be final?

NO

2.5 Constructor

A constructor is a special member method which will be called by the JVM automatically for placing user/programmer defined values instead of placing default values. Constructors are meant for initializing the object

RULES/PROPERTIES/CHARACTERISTICS of a constructor

1. Constructor **name must be similar to name of the class**.

2. Constructor should **not return any value even void** also (if we write the return type for the constructor then that constructor will be treated as ordinary method).

3. Constructors **should not be static** since constructors will be called each and every time whenever an object is creating.

4. Constructor **should not be private** .an object of one class is created in another class (constructor can be private provided an object of one class created in the same class).

5. Constructors will **not be inherited** at all.

6. Constructors are **called automatically whenever an object is creating**.

Types of Constructors

1. ***Default constructor (no-arg constructor)***
2. ***Parameterized constructor***

1. Default constructor

A default constructor is one which will not take any parameters

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

- If there is no constructor in a class, compiler automatically creates a default constructor.
- Default constructor provides the default values to the object like 0, null etc. depending on the type.

2. Parameterized constructor

A constructor that have parameters is known as parameterized constructor

- Parameterized constructor is used to provide different values to the distinct objects
- Constructor overloading is possible by changing no. of parameters

```
public class Demo {
	public Demo(int a, int b) {
		System.out.println(a + ", " + b);
	}
	public Demo(String a) {
		System.out.println(a);
	}
	public static void main(String[] args) {
		new Demo();
	}
}
```

```
The constructor Demo() is undefined
```

Whenever we define/create the objects with respect to both parameterized constructor and default constructor, it is mandatory for the JAVA programmer to define both the constructors

1. Does constructor return any value?

yes, that is current class instance (You cannot use return type yet it returns a value)

2. Can constructor perform other tasks instead of initialization?

Yes, like object creation, starting a thread, calling method etc. You can perform any operation in the constructor as you perform in the method.

3. ***If we place return type in constructor prototype will it leads to Error?***

No, because compiler and JVM considers it as a method.

4. ***If class has explicit constructor, will it has default constructor?***

No. compiler places default constructor only if there is no explicit constructor.

5. ***What is the use of private constructor?***

Private constructors are used to restrict the instantiation of a class. When a class needs to prevent other classes from creating its objects then private constructors are suitable for that

2.6 Static Keyword

The static keyword is mainly used for Memory Management. We will use static in below areas

1. ***Variable (also known as class variable)***
2. ***Method (also known as class method)***
3. ***Block***
4. ***Class nested within another Class***

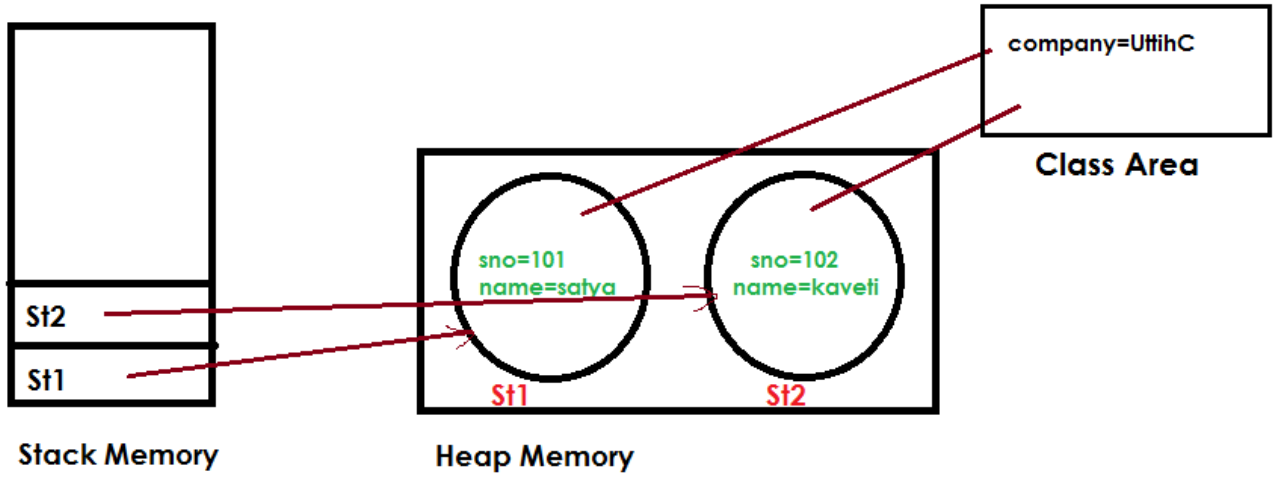
1. Static Variable

Java static property is shared to all objects.The static variable can be used to refer the common property of all objects (that is not unique for each object). It is one for Class.

Ex. company name of employees, college name of students etc.

```
public class StVariable {
	int sno;
	String name;
	static String company = "UttihC";
	StVariable(int sno, String name) {
		this.sno = sno;
		this.name = name;
	}
	public void getData() {
		System.out.println(sno + " : " + name + " : " + StVariable.company);
	}
	public static void main(String[] args) {

		StVariable st1 = new StVariable(101, "Satya");
		StVariable st2 = new StVariable(102, "Kaveti");
		st1.getData();
		st2.getData();

	}
}
```

```
101: Satya : UttihC
102 : Kaveti : UttihC
```



Static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value.

In Real world we use Static variable in ***Counter*** Programs

```
public class Counter {

	int a;
	static int b;

	public void normalCounter() {
		a++;
		System.out.println("normalCounter : " + a);
	}

	public void staticCounter() {
		b++;
		System.out.println("staticCounter : " + b);
	}

	public static void main(String[] args) {
		Counter c1 = new Counter();
		Counter c2 = new Counter();
		Counter c3 = new Counter();

		c1.normalCounter();
		c2.normalCounter();
		c3.normalCounter();
		System.out.println("------------------------");
		c1.staticCounter();
		c2.staticCounter();
		c3.staticCounter();
	}
}
```

```
normalCounter : 1
normalCounter : 1
normalCounter : 1
------------------------
staticCounter : 1
staticCounter : 2
staticCounter : 3
```

2. Static Method

- A static method belongs to the class..
- A static method can be invoked without the need for creating an instance of a class.
- Static method can access static data member and can change the value of it.
- The ***static method cannot use non static data member or call non-static method directly***. ***this and super cannot be used in static context***

```
public class StMethod {
	int a = 100;

	static void dispaly() {
		StMethod method = new StMethod();
		System.out.println(method.a);
	}

	public static void main(String[] args) {
		dispaly();
	}
}
```

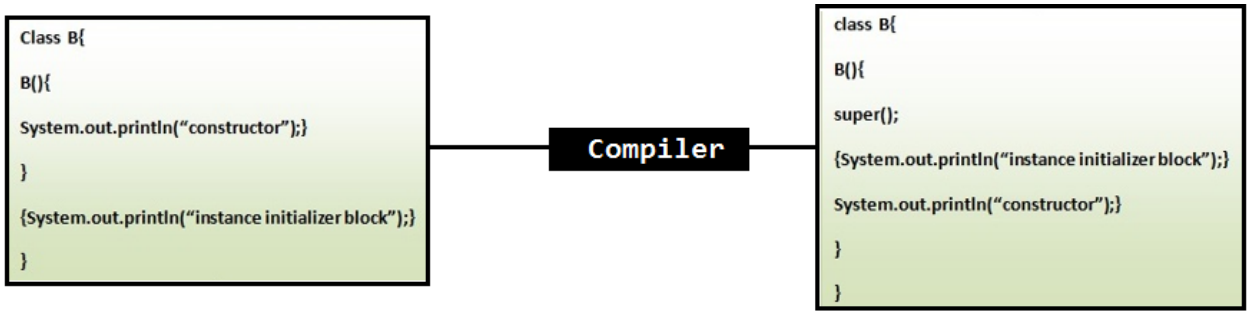
```
100
```

3. Static Block

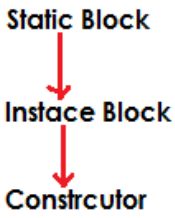
- Is used ***to initialize the static data members***.
- It is ***executed before main*** method at the time of class loading

4. Instance initializer block

- Instance Initializer block is used to initialize the instance data member.
- It run each time when object of the class is created
- There are three places in java where you can perform operations:
  - ***i. method***
  - ***ii. constructor***
  - ***iii. block***
- It will ***invoke Before Constructor***
- The java compiler copies the code of instance initializer block in every constructor
- The instance initializer block is created when instance of the class is created.
- The instance ***initializer block is invoked after the parent class constructor is invoked*** (i.e. after super() constructor call).
- The instance initializer block comes in the order in which they appear



Constructor VS Static VS Instance Block




```
class A {
	A() {
		System.out.println("Class-A :  Costrcutor");
	}
	static {
		System.out.println("Class-A :  Static Block");
	}

	{
		System.out.println("Class-A :  Instance Block");
	}

}

public class Order extends A {
	Order() {
		System.out.println("Order :  Costrcutor");
	}

	static {
		System.out.println("Order :  Static Block");
	}

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

		A ob = new A(); //only A Class
		System.out.println("A========Completed");

		Order o1 = new Order(); //Order class Contains A Class DataMembers
		System.out.println("Order1 ========== Completed");

		Order o2 = new Order(); //Order class Contains A Class DataMembers
		System.out.println("Order2 ========== Completed");
	}
}
```

```
Class-A :  Static Block
Order :  Static Block
Class-A :  Instance Block
Class-A :  Costrcutor
A========Completed
Class-A :  Instance Block
Class-A :  Costrcutor
Order :  Instance Block
Order :  Costrcutor
Order1 ========== Completed
Class-A :  Instance Block
Class-A :  Costrcutor
Order :  Instance Block
Order :  Costrcutor
Order2 ========== Completed
```

2.7 This

This refers to the current Class object

1. this keyword can be used to refer **current class instance variable**.
2. **this()** can be used to invoke **current class constructor**.
3. this keyword can be used to **invoke current class method (implicitly)**
4. this can be **passed as an argument in the method** call.
5. this can be **passed as argument in the constructor** call.
6. this **keyword can also be used to return the current class instance**.

```java
public class Student {
	int sno;
	String name;
	public Student() {
		System.out.println("Constrcutor");
	}

	public Student(int sno, String name) {
	this();// 2. this() can be used to invoke current class constructor
	this.sno = sno;
	this.name = name; // 1. this keyword used to refer current class instance variable
	}

	public void getData() {
		this.title();//3.this keyword used to invoke current class method
		System.out.println("S.No : " + sno + "\t Name : " + name);
	}

	public void title() {
		System.out.println("\nSTUDENT DATA \n======================");
	}

	public static void main(String[] args) {
		Student ob1 = new Student(101, "SATYA");
		Student ob2 = new Student(103, "JOHNNY");
		ob1.getData();
		ob2.getData();
	}
}
```

```
Constrcutor
Constrcutor
STUDENT DATA
======================
S.No : 101       Name : SATYA

STUDENT DATA
======================
S.No : 103       Name : JOHNNY
```

More on this

- **Call to this () must be the first statement in constructor.**

- **this ()** is used for calling current class default constructor from current class parameterized Constructors.

- **this (…)** is used for calling current class parameterized constructor from other category Constructors of the same class.

- Whenever we use either this () or this (…) in the current class constructors, that statements Must be used as **first statement only**.

- Whenever we refer the data members which are similar to method parameters, the JVM Gives first preference to method parameters whereas whenever we write a keyword **this** before the Variable name of a class then the JVM refers to data members of the class

- ***Can we use "this" within static method? Why?***
  Ans. No. Even though "this" would mean a reference to current object id the method gets called using object reference but "this" would mean an ambiguity if the same static method gets called using Class name.

2.8 Method Overloading

If a class have multiple methods by same name but different parameters, it is known as **Method Overloading**

Possible ways of method overloading

- ***By changing number of arguments***
- ***By changing the data type***

Impossible ways of Method overloading

- ***changing the return type of the method***

```
public class Overload {

	public void sum(int x,int y)
	{
		System.out.println(x+y);
	}

	public void sum(double x, double y)
	{
		System.out.println(x+y);
	}
	public static void main(String[] args) {
		Overload o = new Overload();
		o.sum(10.34, 12.45);
		o.sum(1, 4);
	}

}
```

```
22.79
5
```

Why Method Overloading is not possible by changing the return type of method?

Because there may occur ambiguity. Let's see how ambiguity may occur

```
public class Overload {

	public void sum(int x,int y)
	{
		System.out.println(x+y);
	}

	public int sum(int x, int y)
	{
		System.out.println(x+y);
	}
	public static void main(String[] args) {
		Overload o = new Overload();

		o.sum(1, 4);
	}

}
```

```
Overload.java:10: error: method sum(int,int) is already defined in class Overload
	public int sum(int x, int y)
	           ^
```

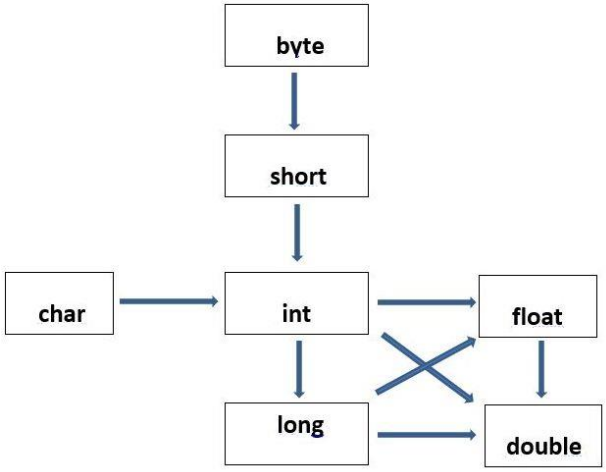
Can we overload main () method?

Yes, by method overloading. You can have any number of main methods in a class by method overloading

Method Overloading and Type Promotion

One type is promoted to another implicitly if no matching data type is found.

byte can be promoted to short, int, long, float or double. The short datatype can be promoted to int,long,float or double. The char datatype can be promoted to int,long,float or double and so on.




```
package ex;

public class Overload {

	public void sum(int x, int y)
	{
		System.out.println("INT : "+(x+y));
	}

	public void sum(double x, double y)
	{
		System.out.println("DOUBLE : "+(x+y));
	}


	public void sum(float x, float y)
	{
		System.out.println("FLOAT : "+(x+y));
	}


	public static void main(String[] args) {
		Overload o = new Overload();
		o.sum(1, 4.5f); //int, float
		o.sum(67.89, 14.5f); //double, float
	}

}
```

```
FLOAT: 5.5
DOUBLE: 82.39
```

2.9 Relationships

Relationship in java → reusing data members from one class to another class

Based on reusing the data members from one class to another class in JAVA we have ***three types*** of relationships

1. ***IS – A***
2. ***HAS –A***
3. ***Uses - A***

1. Inheritance (IS-A)

Is-a relationship is one in which **data members of one class is obtained into another class through the concept of inheritance**

Inheritance is the technique which allows us to ***inherit the data members and methods from base class to derived class.***

Base class : is one which always gives its features to derived classes.
Derived class : is one which always takes features from base class.

A **Derived class** is one which contains some of **features** of its **own plus** some of the data members from base class.

Keynotes

- **Final classes cannot be inherited**.
- **One class can extend only one class at a time**. Since, JAVA does not support multiple inheritance.
- Whatever the data members are coming from base class to the derived class, the **base class members are logically declared in derived class**, the base class **methods are logically defined in derived class**.
- **Private data members and private methods** of the base class will **not be inherited** at all.

Example

```
class B{
public int a=100;
private int b = 200;
public void sum() {
	System.out.println("Sum : "+(a+b));
}
}
```

```
public class A extends B{
	/*public int a=100;		====> Logically Present
	public void sum() {
		System.out.println("Sum : "+(a+b));
	}*/

	public static void main(String[] args) {
		A ob = new A();
		ob.sum();
		System.out.println(ob.a);
		//System.out.println(ob.b);			==> ERROR
	}

}
```

Why multiple inheritance is not supported in java?

Consider a scenario where A, B and C are three classes. The C class inherits A and B classes. If A and B classes have same method and you call it from child class object, there will be ambiguity to call method of A or B class.

Since compile time errors are better than runtime errors, java renders compile time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error now.

```
class A {
	void msg() {
		System.out.println("Hello");
	}
}

class B {
	void msg() {
		System.out.println("Welcome");
	}
}

class C extends A,B {// suppose if it were

	public static void main(String args[]) {
		C obj = new C();
		obj.msg();// Now which msg() method would be invoked?
	}
}
```

2. Has – A

Has-a relationship is one in which an **object of one class is created as a data member in another class**.

```
class Emp{
    int a =100;
}

public class Demo {
private Emp emp;
 public static void main(String[] args) {

}
}
```

3. Uses – A

Uses-a relationship is one in which an **Object of one class is created inside a method of another class.**

```
class Emp {
    int a = 100;
}

public class Demo {
    public void get() {
        Emp ob = new Emp();
    }

}
```

2.10 Super

Super keyword is used for differentiating the base class features with derived class features

Super **keyword is placing an important role in three places.**

i. ***Variable level***
ii. ***Method level***
iii. ***Constructor level***

1. Super at Variable level

Whenever we inherit the base class members into derived class, there is a possibility that base class members are similar to derived class members

To **distinguish the base class members with derived class** members we use Super keyword to variable

```
class B{
public int a=100;
}

public class A extends B{
```

```
        int a = 500;
        public void show()
        {
               System.out.println("NORMAL a : "+a);
               System.out.println("SUPER a  : "+super.a);
        }
       public static void main(String[] args) {
              new A().show();
       }
}
```

```
NORMAL a: 500
SUPER a: 100
```

2. Super at Method Level

Whenever we inherit the base class methods into the derived class, there is a possibility that **base class methods are May similar to derived methods.**

To **differentiate** the base class methods with derived class methods in the derived class, the base class **methods must be preceded by a keyword super**.

```
package ex;

class B{
        public void show()  {
               System.out.println("SUPER CLASS Show()");
        }
}

public class A extends B{
        public void show()  {
               System.out.println("SUBCALSS Show()");
        }

        public void test() {
              show();
              super.show();
       }
       public static void main(String[] args) {
              new A().test();
       }

}
```

```
SUBCALSS Show()
SUPER CLASS Show()
```

Hello Master, this is called method Overriding!!

3. Super at Constructor level

- Whenever we develop any inheritance application, we use to create always object of bottom most derived class.
- When we create an object of bottom most derived class, it in turns calls its immediate super class default constructor and it in turns calls its top most super class default constructor.
- Therefore, in JAVA environment, constructors will be called always from bottom to top and the execution starts from top to bottom

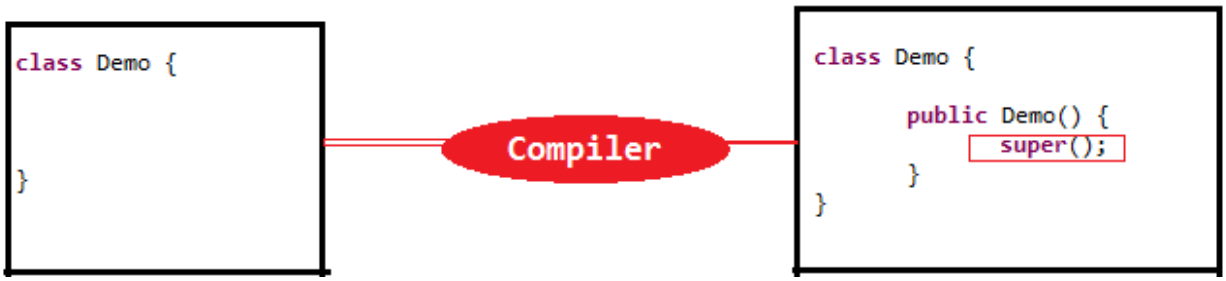
Super (): is used for **calling super class default constructor** from default constructor or from parameterized constructor of derived class.

Super (...): is used for **calling super class parameterized constructor** either from default constructor or from parameterized constructor of derived class. It is always mandatory

```
class B {

	public B() {
		System.out.println("SUPER CONSTRCTOR ");
	}
}

public class A extends B {
	public A() {
		super();
		System.out.println("A CONSTRCUTOR ");
	}

	public static void main(String[] args) {
		new A();
	}

}
```

```
SUPER CONSTRCTOR
A CONSTRCUTOR
```

Above program is only for example purpose. Actually super () is added in each class constructor automatically by compiler.



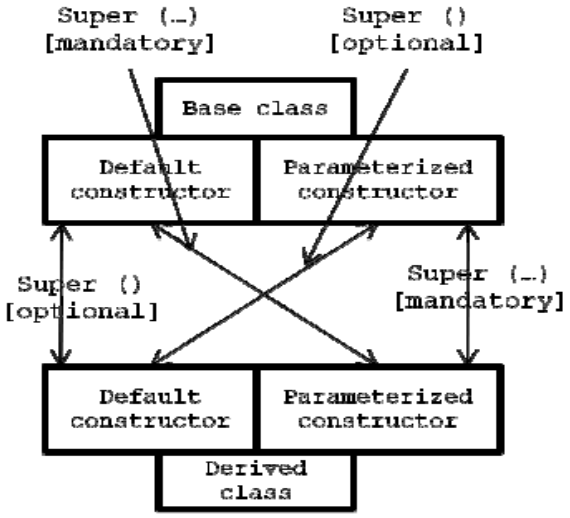
CASES

1. Whenever we want to call **default constructor** of base class **from default constructor** of derived class **using super ()** in default constructor of derived class is **optional**

2. Whenever we want to **call the super class parameterized** class from parameterized class of the derived class using **super (...)** in parameterized class of derived class is **mandatory**.

3. Whenever we want to call default constructor of base class from parameterized class of derived class using super () in parameterized class of derived class is optional

4. Whenever we want to **call parameterized class of base class from default constructor** of derived class using super (...) in default constructor of derived class is mandatory




```
class B {

	public B(int a) {
		System.out.println("SUPER CONSTRCTOR ");
	}
}

public class A extends B {

	public static void main(String[] args) {
		new A();
	}

}
```

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
	Implicit super constructor B() is undefined for default constructor. Must define an explicit
constructor
```

2.11 Method Overriding

Method overriding is used for runtime polymorphism

If subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as method overriding.

Rules

- Method must have ***same name as in the parent class***
- Method must have same ***parameter as in the parent class***.
- ***Must be IS-A relationship (inheritance).***
- ***You must not reduce the visibility of a method while overriding.***

Example: Tax functionality to get Tax%. But Tax varies according to States. For example, AP, UP and MP states tax% are 5%, 7% and 9%

```
class Tax {
	public int getTax() {
		return 0;
	}
}

class AP extends Tax {
	public int getTax() {
		return 5;
	}
}

class UP extends Tax {
	public int getTax() {
	return 7;
	}
}

class MP extends Tax {
	public int getTax() {
		return 9;
	}
}
public class Test {
	public static void main(String[] args) {
		AP ap = new AP();
		MP mp = new MP();
		UP up = new UP();
		System.out.println("AP : " + ap.getTax());
		System.out.println("MP : " + mp.getTax());
		System.out.println("UP : " + up.getTax());
	}
}
```

```
AP : 5
MP : 9
UP : 7
```

More on Overriding

Can we override static method?

No, static method cannot be overridden. Because static method is bound with class whereas instance method is bound with object. Static belongs to class area and instance belongs to heap area

Can we override java main method?

No, because main is a static method.

Can we change access modifiers in derived class (public to private, etc?)

If you are overriding any method, overridden method (i.e. declared in subclass) must not be more restrictive. That means you can change private to public, not public to private

Can we override protected method of super class as public method in the sub class?

Yes. You can increase the visibility of overriding methods but can't reduce it.

Can we change the return type of overriding method from Number type to Integer type?

Yes. You can change as Integer is a sub class of Number type.

Can we override a super class method without throws clause as a method with throws clause in the sub class?

Yes, but **only with unchecked type** of exceptions

Can we override private methods?

No, they are not at all inherited to sub class.

Method Overlaoding	Method Overriding
Method overloading is used to increase the readability of the program.	Method overriding is used to provide the specific implementation of the method that is already provided by its super class.
Method overloading is performed within class.	overriding occurs in two classes that have IS-A (inheritance) relationship.
In case of method overloading, parameter must be different.	In case of method overriding, parameter must be same.
Method overloading is the example of compile time polymorphism.	Method overriding is the example of run time polymorphism.
In java, method overloading can't be performed by changing return type of the method only. Return type can be same or different in method overloading. But you must have to change the parameter.	Return type must be same or covariant in method overriding

2.12 Abstraction

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

There are two ways to achieve abstraction in java

- ***Abstract class (0 to 100%)***
- ***Interface (100%)***

1. Abstract Class

In JAVA we have two types of classes. They are concrete classes and abstract classes.

Concrete class: is one which ***contains fully defined methods***

Abstract class: is one which ***contains some defined methods and some undefined methods***

Notes

- A class that is declared as abstract is known as **abstract class**
- A method , declared as abstract and does not have implementation is known as **abstract method**

```
abstract class A {     //abstract class
       abstract int sum();//abstract method
       public void test()
       {
              System.out.println("I Normal");
       }
}
```

- We **cannot create abstract class object directly** but we can create indirectly. An **object of abstract** class is **equal** to an object of that class which **extends** that **abstract class**

- Abstract classes should not be final, since, they are always reusable.

- Abstract classes are basically used to ***implement polymorphism***;

- ***If there is any abstract method in a class, that class must be abstract***

- ***If you are extending any abstract class that have abstract method, you must either provide the implementation of the method or make this class abstract***

- ***We can also declare any class as Abstract class, even that class contains all concrete methods***

```java
abstract class Tax {
	abstract int getTax();
}

class AP extends Tax {
	@Override
	public int getTax() {
		return 5;
	}
}

class UP extends Tax {
	@Override
	public int getTax() {
		return 7;
	}
}

class MP extends Tax {
	@Override
	public int getTax() {
		return 9;
	}
}

public class Test {
	public static void main(String[] args) {
		Tax ap = new AP();
		Tax mp = new MP();
		Tax up = new UP();
		System.out.println("AP : " + ap.getTax());
		System.out.println("MP : " + mp.getTax());
		System.out.println("UP : " + up.getTax());

	}
}
```

```
AP : 5
MP : 9
UP : 7
```

2. Interfaces

An Interface is a collection of **`public static final`** data members and public abstract methods

Notes

- We use **implements** keyword to implement interface by any class
- Interfaces are basically used to develop user defined data types.
- Interface is a keyword which is used for developing user defined data types
- we cannot create interface object directly but we can create indirectly

- all **variables** by default belongs to (no need to write explicitly)

  `public static final xxx data members`

  All variables must be initialized (bcoz final, otherwise it will be compilation error).

- All methods by default belongs to (no need to write explicitly)

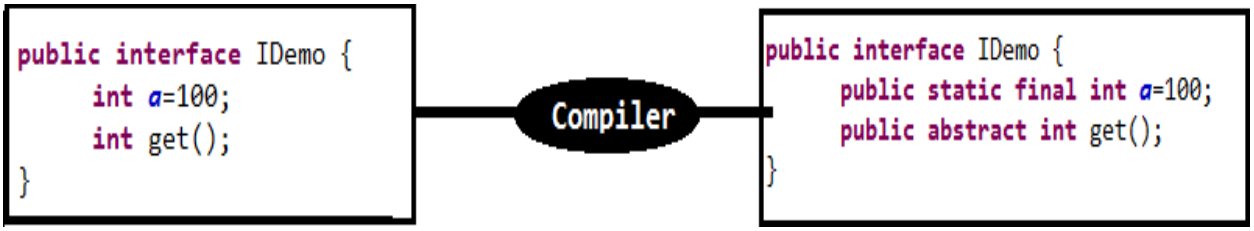
```
public abstract xxx xxxMethods()
```

- An object of base interface contains the details about those methods which are declared in that interface only but it does not contain details about those methods which are specially available in either in derived classes or in derived interfaces.

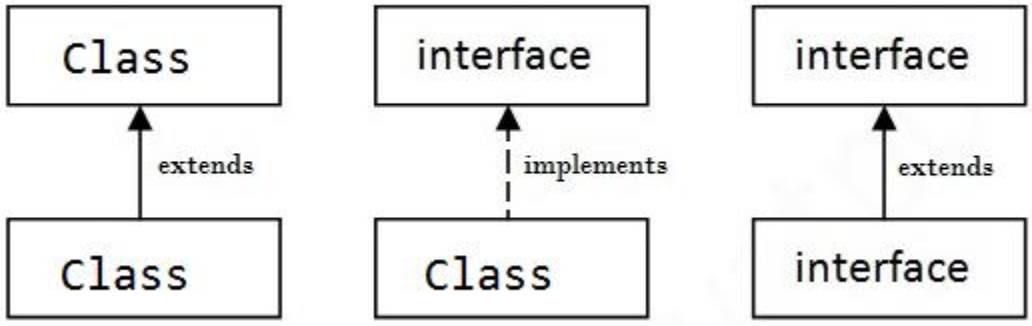
- Interfaces **should not be final**.(because implement's not possible)

- An interface does **not contain Constructors**. (because no need of create objects)

The java compiler adds public and abstract keywords before the interface method and public, static and final keywords before data members



Relationship between classes and interfaces



Multiple inheritance is not supported through class in java but it is possible by interface, why?

Yes, because there is no ambiguity as implementation is provided by the implementation class

```
interface Student {
	void show();
}

interface Master {
	void show();
}

public class Test implements Student, Master {

	@Override
	public void show() {
		System.out.println("Two Interfaces are Same");
	}

	public static void main(String[] args) {
		Student ob1 = new Test();
		Master ob2 = new Test();

		ob1.show();
		ob2.show();
	}
}
```

```
Two Interfaces are Same
Two Interfaces are Same
```

3. Nested Interface

An interface declared within another interface or class is known as nested interface

Nested interface must be public if it is declared inside the interface but it can have any access modifier if declared within the class. Nested interfaces are declared static implicitly

```
interface Student{
	  void getName();
	  interface Address{
	   void getAddress();
	  }
	}

public class Test implements Student.Address {

	@Override
	public void getAddress() {
		System.out.println("HYDERABAD");
	}
	public static void main(String[] args) {
		 Student.Address ob = new Test();
		 ob.getAddress();
	}
}
```

```
HYDERABAD
```

Internal code generated by the java compiler for nested interface Address

```
public static interface Student$Address{

        public abstract void getAddress();

}
```

4. Marker Interface

An Empty interface in java called Marker Interface

Marker interface in Java is interfaces with no field or methods or in simple word empty interface in java is called marker interface. It is used to convey to the JVM that the class implementing an interface of this category ***will have some special behavior***.

Example of market interface: ***Serializable, Clonnable*** and ***Remote interface***

Abstract class	Interface
1) Abstract class can have **abstract and non-abstract methods.**	Interface can have **only abstract methods**.
2) Abstract class **doesn't support multiple inheritance.**	Interface **supports multiple inheritance**.
3) Abstract class can **have final, non-final, static and non-static variables.**	Interface has **only static and final variables**.
4) Abstract class can **have static methods, main method and constructor**.	Interface **can't have static methods, main method or constructor**.
5) Abstract class can provide the implementation of interface.	Interface can't provide the implementation of abstract class.
6) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.

More on Abstract and Interfaces

1. Can we define a class inside the interface?

Yes, if we define a class inside the interface, java compiler creates a static nested class. Let's see how we can define a class within the interface:

```
interface M{
  class A{}
}
```

2. Can abstract class have static methods in Java?

Yes, abstract class can declare and define static methods, nothing prevents from doing that. But, you must follow guidelines for making a method static in Java, as it's not welcomed in a object oriented design, because static methods cannot be overridden in Java. It's very rare, you see static methods inside abstract class, but as I said, if you have very good reason of doing it, then nothing stops you.

3. Can abstract class contains main method in Java?

Yes, abstract class can contain main method, it just another static method and you can execute Abstract class with main method, until you don't create any instance

4. Class C implements Interface I containing method m1 and m2 declarations. Class C has provided implementation for method m2. Can i create an object of Class C?

No not possible. Class C should provide implementation for all the methods in the Interface I. Since Class C didn't provide implementation for m1 method, it has to be declared as abstract. Abstract classes can't be instantiated.

5. Can we declare abstract methods as private? Justify your answer?

No. Abstract methods cannot be private. If abstract methods are allowed to be private, then they will not be inherited to sub class and will not get enhanced.

2.13 Polymorphism

Polymorphism is derived from 2 greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are ***two types of polymorphism*** in java. They are,

1. **Compile time polymorphism** (static binding or ***method overloading***)
2. **Runtime polymorphism** (dynamic binding or ***method overriding***)

1. What is Type?

1) Variables have a type

Each variable has a type, it may be primitive and non-primitive.

```
int data=30;
```

Here data variable is a type of int.

2) References have a type

```
class Dog{
 public static void main(String args[]){
  Dog d1;//Here d1 is a type of Dog
 }
}
```

3) Objects have a type

An object is an instance of particular java class, but it is also an instance of its superclass.

```
class Animal{}

class Dog extends Animal{
 public static void main(String args[]){
  Dog d1=new Dog();
 }
}
```

Here d1 is an instance of Dog class, but it is also an instance of Animal.

2. Static Binding

When type of the object is determined at compiled time (by the compiler), it is known as static binding.

If there is any ***private, final or static method*** in a class, there is ***static binding***

```
public class Student {
	void show(){
		System.out.println("Student Data");
	}
	public static void main(String[] args) {
		Student student = new Student();
		student.show();
	}
}
```

3. Dynamic Binding

When type of the object is determined at run-time, it is known as dynamic binding

```
class Game{
	public void msg() {
		System.out.println("NULL GAME");
	}
}
class Cricket extends Game{
	public void msg() {
	System.out.println("CRICKET GAME");
	}
}
public class Test {
	public static void main(String[] args) {
		Game ob = new Cricket();
		ob.msg();
	}
}
```

```
CRICKET GAME
```

In the above example object type cannot be determined by the compiler, because the instance of Cricket is also an instance of Game. So compiler doesn't know its type, only its base type.

4. Compile time polymorphism (*static binding or method overloading*)

- Here ***type of the object is determined at compiled time*** (by the compiler)
- It is also known as ***Static binding, early binding*** and ***overloading*** as well.
- We can achieve compile time polymorphism by ***method overloading***
- Overloading is compile time polymorphism where more than one methods share the same name with different parameters or signature
- It provides ***fast execution*** because known early at compile time.
- ***Less flexible***, because as all things execute at compile time.

```
public class Test2 {

	void sum(int x, int y)
	{
		System.out.println((x+y));
	}

	void sum(int x, int y, int z)
	{
		System.out.println((x+y+z));
	}
		void sum(float x, int y)
	{
		System.out.println((x+y));
	}
	public static void main(String[] args) {
		Test2 t = new Test2();
		t.sum(10, 20);
```

```
                t.sum(10.5f, 21);
                t.sum(10,20,20);
        }
}
```

```
30
31.5
50
```

5. Runtime polymorphism (*dynamic binding or method overriding*)

- It is also known as ***Dynamic binding, late binding and overriding*** as well.
- Overriding is run time polymorphism having ***same method*** with same parameters or signature, but associated in a class & ***its subclass***.
- It provides ***slow execution*** as compare to early binding , because it is ***known at runtime***
- ***More flexible as all things execute at run time.***

```
class Tax {
    int getTax() {
            return 0;
    }
}
class AP extends Tax {
    public int getTax() {
            return 5;          }
}
class UP extends Tax {
    public int getTax() {
            return 7;
    }
}
class MP extends Tax {
    public int getTax() {
            return 9;
    }
}
public class Test {
    public static void main(String[] args) {
            Tax ap = new AP();
            Tax mp = new MP();
            Tax up = new UP();
            System.out.println("AP : " + ap.getTax());
            System.out.println("MP : " + mp.getTax());
            System.out.println("UP : " + up.getTax());
    }
}
```

```
AP : 5
```

```
MP : 9
UP : 7
```

Runtime polymorphism can't be achieved by data members.

```
class Car {
	int mileage = 140;
}

public class Zen extends Car {
	int mileage = 30;
	public static void main(String[] args) {
		Car c = new Zen();
		System.out.println(c.mileage);
	}
}
```

If you think you override mileage variable, it must give output as 140. But it will give O/P 30. Because overriding variable not possible.

2.14 Access Modifiers

There are two types of modifiers in java

1. **Access modifiers**
2. **Non-access modifiers.**

1. Access Modifiers

There are 4 types of java access modifiers:

- private
- default
- protected
- public

Let's understand the access modifiers by a simple table.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

2. Non- Access Modifiers

Non-access modifiers do not change the accessibility of variables and methods, but they do **provide them special properties.** Below are the Non Access Modifiers available in Java.

- ***Final***
- ***Abstract***
- ***Static***
- ***Strictfp***
- ***Native***
- ***Synchronized***
- ***Transient***

1. Final

- **Final Class** : A Class when set to final **cannot be extended** by any other Class.
- **Final Method** : A Method when set to final **cannot be overridden** by any subclass.
- **Final Variable** : When a variable is set to final, **its value cannot be changed**.

2. Abstract Class

Abstract Class, **Abstract methods** for Data Abstraction

3. Static

Static Modifiers are used to create class variable and class methods which can be accessed without instance of a class.

4. Stictfp

- Java **strictfp** keyword ensures that you will get the same result on every platform if you perform operations in the floating-point variable.

- The precision may differ from platform to platform that is why java programming language have provided the **strictfp** keyword, so that you get same result on every platform. So, now you have better control over the floating-point arithmetic

- The strictfp keyword **can be applied** *on* ***methods, classes and interfaces.***

```
strictfp class A{}//strictfp applied on class

strictfp interface M{}//strictfp applied on interface

class A{
strictfp void m(){}//strictfp applied on method
}
```

The strictfp keyword **cannot** be applied on ***abstract methods, variables or constructors***

5. **Transient modifier**
   - Transient variables **cannot** participate in **serialization** process.
   - An instance variable is marked transient to indicate the JVM to skip the particular variable when serializing the object containing it.

```
public transient int limit = 55;   // will not persist
public int b; // will persist
```

6. **Synchronized modifier**

   When a method is synchronized it can be accessed by only one thread at a time. We will discuss it in detail in Thread.

```
public synchronized void showDetails(){
.......
}
```

7. **Volatile modifier**
   - Volatile modifier is used in **multi-threaded** programming.
   - If you declare a field as volatile it will be signal to the threads that its **value must be read from the main memory rather than their own stack**.
   - Because volatile field is common to all threads and it will be updated frequently by multiple threads. ***Example will explain in Threads*** ☺

2.15 Covariant Return Type

From Java5 onwards ***changing the return type of overridden method in sub class is also acceptable***.

```
class Car {
	public Car getObject() {
		return this;
	}
}
class Zen extends Car {
	@Override // Noraml way of Ovveriding
	public Car getObject() {
		return super.getObject();
	}
}

public class Benz extends Car {
	@Override // by chnaging return type
	public Benz getObject() {
		return this;
	}
}
```

As you can see in the above example, the return type of the getObject() method of Car class is Car. But the return type of the getObject () method of Benz class is Benz. ***Both methods have different return type but it is method overriding***. This is known as covariant return type.

2.16 Wrapper class

Wrapper class in java provides the mechanism to **convert primitive into object and object into primitive**.Wrapper classes are basically ***used for converting the string data into fundamental data type***. Each and every wrapper class contains the following generalized parse methods.

```
public static Xxx parseXxx (String);
```

Primitive Type	Wrapper class	String to Primitive	Primitive to Wrapper
boolean	Boolean	P s booean parseBoolean(str)	Boolean.valueOf(bo)
char	Character	p s char parseChar (str)	Char. valueOf (char)
byte	Byte	p s byte parseByte (str)	Byte. valueOf (byte)
short	Short	p s short parseShort (str)	Short. valueOf (short)
int	Integer	p s int parseInt (str)	Integer. valueOf (int)
long	Long	p s long parseLong (str)	Long. valueOf (long)
float	Float	p s float parseFloat (str)	Float. valueOf (float)
double	Double	p s double parseDouble(str)	Double. valueOf (dou)

```
public class Wrapper {
	int i = 100;
	long l = 1000l;
	float f = 239.78f;
	double d = 10032.78d;
	public void show() {
		Integer integer = Integer.valueOf(i);
		Double double1 = Double.valueOf(d);
		Float float1 = Float.valueOf(f);
		System.out.println("Primitive to Wrapper");
		System.out.println("------------------------");
		System.out.println(integer+ ":" +float1 +":" + double1);
		System.out.println("Wrapper to Primitive");
		System.out.println("------------------------");
		System.out.println(Integer.parseInt(integer.toString()));
		System.out.println(Double.parseDouble(double1.toString()));
		System.out.println(Float.parseFloat(float1.toString()));
	}
	public static void main(String[] args) {
		new Wrapper().show();
	}
}
```

```
Primitive to Wrapper
-----------------------
100: 239.78: 10032.78

Wrapper to Primitive
-----------------------
100
10032.78
239.78
```

2.17 Object Class

The **java.lang.Object** class is the parent class of all the classes in java by default

1. Why it is Default Super Class to All Classes

Reusability

- Every object has 11 common properties.
- These properties must be implemented by every class developer
- So to reduce burden on developer SUN developed a class called Object by implementing all these 11 properties with 11 methods
- All these methods have generic logic common for all sub classes. if this logic is not satisfying subclass requirement then subclass can override it

Runtime Polymorphism

- You can refer any object whose type ***don't know as object class*** object.

2. Methods

11 methods are divided into 2 types as **Final methods**, **Overridable methods**

Final Methods [6]

1. public final Class **getClass()**
2. public final void **wait()** throws InterruptedException
3. public final void **wait(long timeout)** throws InterruptedException
4. public final void **wait(long timeout,int nanos)** throws InterruptedException
5. public final void **notify()**
6. public final void **notifyAll()**

Overridable methods [5]

7. public String **toString()**
8. public boolean **equals(Object obj)**
9. public int **hashcode()**
10. protected Object **clone()** throws CloneNotSupportedException
11. protected void **finalize()** throws Throwable

In above classes except **wait(long timeout,int nanos), wait(long timeout), equals(Object obj), toString(), *remaining all are native methods*.**

Complied Object Class

```
package java.lang;
public class Object {

       public final native Class<?> getClass();
       public native int hashCode();
       public final native void notify();
    public final native void notifyAll();
    protected native Object clone() throws CloneNotSupportedException;
    public final native void wait(long timeout) throws InterruptedException;
    private static native void registerNatives();

  static {
        registerNatives();
    }

    public boolean equals(Object obj) {
        return (this == obj);
    }

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

    public final void wait(long timeout, int nanos) throws InterruptedException {
        if (timeout < 0) {
            throw new IllegalArgumentException("timeout value is negative");
        }
        if (nanos < 0 || nanos > 999999) {
            throw new IllegalArgumentException(
                                "nanosecond timeout value out of range");
        }
        if (nanos >= 500000 || (nanos != 0 && timeout == 0)) {
            timeout++;
        }
        wait(timeout);
    }

    public final void wait() throws InterruptedException {
        wait(0);
    }
    protected void finalize() throws Throwable {
    }
}
```

3. Final Methods [6]

We have 6 final methods. **wait()** has **3 versions**, **notify** has **2 versions**.

1) Public native final Class getClass()

This method returns the runtime class of this Object

```
public class Native {
	public static void main(String[] args) {
		Date d = new Date();
		System.out.println("Class : "+d.getClass());
	}
}
```

```
Class : class java.util.Date
```

2) public final void wait() throws InterruptedException

To wait current thread until another thread invokes the notify()

3) public final void wait(long timeout) throws InterruptedException

Causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).

4) public final void wait(long timeout,int nanos) throws InterruptedException

Causes the current thread to wait for the specified miliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).

5) public final void notify()

Wakes up single thread, waiting on this object's monitor.

6) public final void notifyAll()

Wakes up all the threads, waiting on this object's monitor

4. Overridable methods [5]

7. Public String toString()

We can use **toString()** method to get String Representation an Object.when ever we are tryingto print Object reference, internally toString() method will be called.

S.o.p(S1) == S.o.p(S1.toString())

```
public class Student {
	String rollno;
	String name;

	public Student(String rollno, String name) {
		this.rollno = rollno;
		this.name = name;
	}
	public static void main(String[] args) {
		Student s1 = new Student("101", "Satya");
		Student s2 = new Student("102", "Surya");
		System.out.println(s1);			// Student@15db9742
		System.out.println(s1.toString());	// Student@15db9742
	}
}
```

```
Student@15db9742
Student@15db9742
```

toString() is implemted as follows

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

Override toString() to display actual data in the Object

```
public class A {
	int sno;
	String name;
	String addr;
//Setters and Getters
	@Override
	public String toString() {
		return getSno() + " : " + getName() + " : " + getAddr();
	}

	public static void main(String[] args) {
		A a = new A();
		a.setSno(101);
		a.setName("Satya");
		a.setAddr("Hyd");
		System.out.println(a.toString());
	}
}
```

```
101 : Satya : Hyd
```

8. public boolean equals(Object obj)

Used to make equal comparison between two objects.it has 2 versions while comparing Objects

1.equals () at Object comparision level

- If we are comparing **non-String Objects** eqauls() method it **comapiars refrences of Objects**.
- **It is same as "== " Operator**

2. equals() at String Comparision Level

- If we are comparing **String data on** .eqauls() method **comapiars only content**.
- **References are doesn't matter**.
  But == always compairs **references**

See detailed at String Handling

9. public int hashcode()

Returns the hash code of the given object

- For every Object a **UNIQUE Number Generated by JVM** known as **Hashcode**
- Hashcode **won't represents Address of Object,**but it will use address to generate hashcode
- JVM will use hashcode while Saving Objects into Hashing related datastructres like **Hashtable, HashSet, HashMap etc because search Operation become Easy(**The most powerfull search alogorithm upto today is **Hashing)**

We can override **hashcode()** to generate our own hashcode, but hashcode must be UNIQUE

In Properway

```
public int hashCode() {
	return 1;
};
```

Example

```
public class HashDemo {
public static void main(String[] args) {
	HashDemo h1 = new HashDemo();
	HashDemo h2 = new HashDemo();
	System.out.println(h1.hashCode());
	System.out.println(h2.hashCode());
}
}
```

```
366712642
1829164700
```

10. Protected Object clone() throws CloneNotSupportedException

Creates and returns the exact copy (clone) of the object.

Rules

- *To clone an Object it must implement **java.lang.Clonable** Interface*
- *Otherwise it will return **CloneNotSupportedException.***

```
Public class Student implements Cloneable {
   int sno;
   String name;
   public Student(int sno, String name) {
          this.sno = sno;
          this.name = name;
   }

   public static void main(String[] args) throws CloneNotSupportedException {
          Student s1 = new Student(101, "Satya");
          Student s2 = (Student) s1.clone();
          System.out.println("S1 data ➔ "+s1.sno+":"+s1.name);
          System.out.println("S2 data ➔ "+s2.sno+":"+s2.name);
   }
}
```

```
S1 data ➔ 101:Satya
S2 data ➔ 101:Satya
```

We have two types of Cloning in java

1. Shallow Cloning
2. Deeply Cloning

1. Shallow copy Cloning

- The default version of **clone()** method creates the shallow copy of an object.
- The shallow **copy of an object will have exact copy of all the fields of original object**.
- If original object has **any references** to other objects as fields, then ***only references of those objects are copied*** into clone object, copy of those objects are not created
- Any **changes** made to those objects **through clone object will be reflected in original object or vice-versa**.
- Shallow copy is not 100% independent of original object.

Example

```
package clone;
class Address {
       String dno;
       String city;
       public Address(String dno, String city) {
              super();
              this.dno = dno;
              this.city = city;
       }
```

```
}

 class Student implements Cloneable {
       int sno;
       String name;
       Address address;
       public Student(int sno, String name, Address address) {
              super();
              this.sno = sno;
              this.name = name;
              this.address = address;
       }
       @Override
       protected Object clone() throws CloneNotSupportedException {
                     return super.clone();
       }
}

public class ShallowClone {
public static void main(String[] args) throws CloneNotSupportedException {

       Address addr = new Address("3-100", "HYDERABAD");
       Student s1 = new Student(101, "Satya", addr);
       Student s2 = (Student) s1.clone();
 System.out.println(s1.address.city+" : "+s2.address.city); //HYDERABAD : HYDERABAD

s1.address.city = "KANURU";  //Changing the Value

System.out.println(s1.address.city+" : "+s2.address.city);//KANURU : KANURU

       //S1,S2 are dependent to each other, sharing same Object reference

}
}
```

2. Deeply copy Cloning

- To create the deep copy of an object, you have to **override clone() method**
- Deep **copy of an object will have exact copy of all the fields of original object just like shallow copy**.
- But in additional, if original object has any references to other objects as fields, then copy of those objects are also created by calling clone() method on them.
- That means clone object and original object will be 100% disjoint. They will be 100% independent of each other
- Changes won't reflect each other.

Example

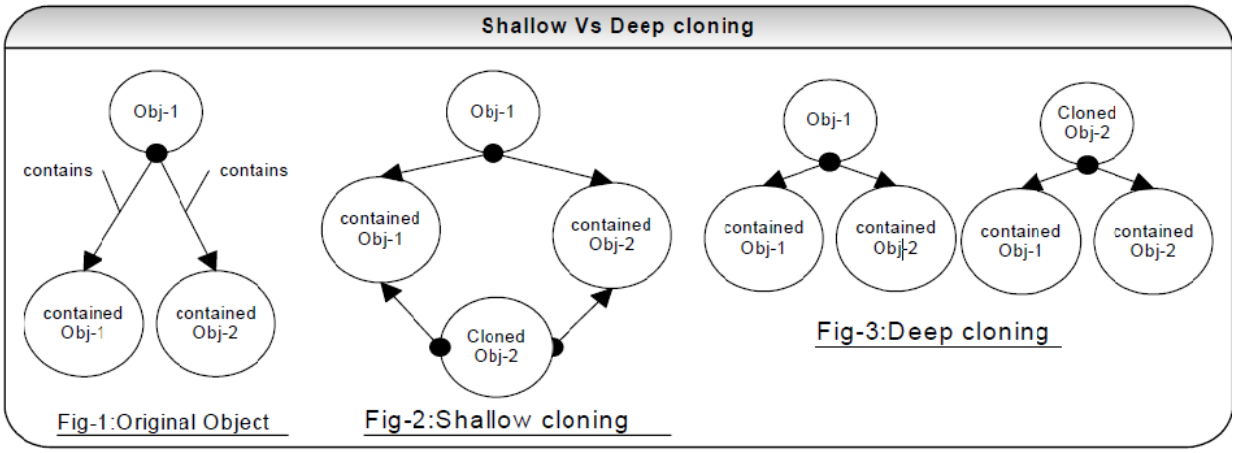
```
package clone.deep;
class Address implements Cloneable{
       String dno;
```

```
	String city;
	public Address(String dno, String city) {
		super();
		this.dno = dno;
		this.city = city;
	}
	@Override
	protected Object clone() throws CloneNotSupportedException {
		// TODO Auto-generated method stub
		return super.clone();
	}
}

 class Student implements Cloneable {
	int sno;
	String name;
	Address address;

	public Student(int sno, String name, Address address) {
		super();
		this.sno = sno;
		this.name = name;
		this.address = address;
	}

	@Override
	protected Object clone() throws CloneNotSupportedException {
			Student student = (Student) super.clone();
			student.address = (Address) address.clone();
			return student;
	}

}

public class DeeplyClone {
public static void main(String[] args) throws CloneNotSupportedException {

	Address addr = new Address("3-100", "HYDERABAD");
	Student s1 = new Student(101, "Satya", addr);
	Student s2 = (Student) s1.clone();

System.out.println(s1.address.city+" : "+s2.address.city); //HYDERABAD : HYDERABAD

	 s1.address.city = "KANURU";  //Changing the Value

	 System.out.println(s1.address.city+" : "+s2.address.city);//KANURU : KANURU

	//S1,S2 are dependent to each other , sharing same Object reference
}
}
```

```
HYDERABAD : HYDERABAD
KANURU : HYDERABAD
```

The shallow copying of this object will be pointing to the same memory reference as the original object. So a change in myData by either original or cloned object will be reflected in other also.

But in deep copying there will memory allocated and values assigned to the property will be same. Any change in object will not be reflected in other



11. protected void finalize() throws Throwable

It is invoked by the garbage collector before object is being garbage collected

2.18 Factory Method

A factory method is one whose return type is similar to name of the class where it presents.

RULES for factory method:

- The **return type** of the factory method **must be similar to name of the class** where it presents.
- Every **factory method must be static** (so that we can call with respect to name of the class).
- Every factory method must be **public**.

Factory methods are used for creating an object without using new operator. Every predefined abstract class contains at least one factory method for creating an object of abstract class.

Whenever we define a concrete class, that concrete class also can be made it as abstract and it is always further reusable or extendable by further classes.

```
interface Dog {
	void type();
}

//=========DOG Implementing classes ==========
class SmallDog implements Dog {
	public void type() {
		System.out.println("Iam SMALL DOG");
	}
```

```
}
class MediumDog implements Dog {
	public void type() {
		System.out.println("Iam MEDIUM DOG");
	}
}
class BigDog implements Dog {
	public void type() {
		System.out.println("Iam BIG DOG");
	}
}


//=========DOG FACTORY ==========
class DogFactory{
	public static Dog getDogFactory(String rule)
	{
		if(rule.contains("small"))
			return new SmallDog();
		else if (rule.contains("medium"))
			return new MediumDog();
		else if (rule.contains("big"))
			return new BigDog();
		else
			return null;
	}

}
//=========DOG FACTORY ==========
public class FactoryMain {
public static void main(String[] args) {
 DogFactory.getDogFactory("small").type();
 DogFactory.getDogFactory("medium").type();;
 DogFactory.getDogFactory("big").type();;

}
}
```

```
Iam SMALL DOG
Iam MEDIUM DOG
Iam BIG DOG
```

Steps

1. Create an Interface for which you want to create Factory of Object (Dog)
2. Create implementation classes (BigDog, SmallDog, MediumDog)
3. Create Factory Class which contains static method whose return type similar to class
4. Write Factory method which returns no.of Objects

2.19 Strictfp

Java strictfp keyword ensures that you will get the same result on every platform if you perform operations in the floating-point variable

The strictfp keyword can be applied on methods, classes and interfaces.

```
strictfp class A{}//strictfp applied on class
strictfp interface M{}//strictfp applied on interface
class A{
strictfp void m(){}//strictfp applied on method
}
```

The strictfp keyword cannot be applied on abstract methods, variables or constructors.

```
class B{
strictfp abstract void m();//Illegal combination of modifiers
}
class B{
strictfp int data=10;//modifier strictfp not allowed here
}
class B{
strictfp B(){}//modifier strictfp not allowed here
}
```

2.20 Java Reflection API (java.lang.Class)

Reflection is commonly used by programs which require the ability ***to examine or modify the runtime behavior of applications*** running in the Java virtual machine

Where it is used

- IDE (Integrated Development Environment) e.g. Eclipse, MyEclipse, NetBeans etc.
- Debugger
- Test Tools etc

1. Java.lang.Class

The java.lang.Class class performs mainly two tasks:

- Provides methods to get the **metadata** of a class at run time.
- Provides methods to **examine and change the run time behavior of a class**.

2. Methods

Method	Description
public String **getName**()	returns the class name
public static Class **forName**(String className)throws ClassNotFoundException	Loads the class and returns the reference of Class class.
public Object **newInstance()**throws InstantiationException,IllegalAccessException	Creates new instance.
public boolean **isInterface()**	Checks if it is interface.
public boolean **isArray()**	Checks if it is array.
public boolean **isPrimitive()**	Checks if it is primitive.
public Class **getSuperclass()**	Returns the superclass class reference.
public Field[] **getDeclaredFields()**	Returns the total number of fields of this class.
public Method[] **getDeclaredMethods()**	Returns the total number of methods of this class.
public Constructor[] **getDeclaredConstructors()**	Returns the total number of constructors of this class.
public Method **getDeclaredMethod**(String name,Class[] parameterTypes)throws NoSuchMethodException,SecurityException	Returns the method class instance.

III.java.lang Package (Java String Handling)

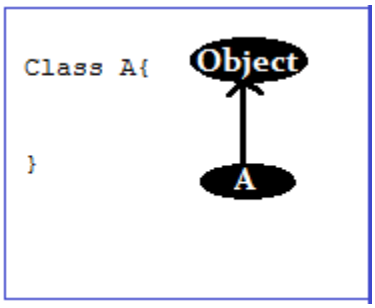
We have mainly five classes in java.lang. Which are most commonly used in any java program

1. **Object**
2. **String**
3. **StringBuffer**
4. **StringBuilder**
5. **Wrapper Classes (AutoBoxing / AutoUnboxing)**

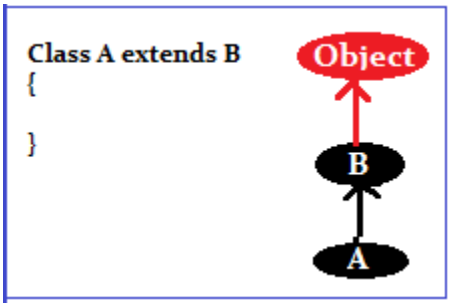
3.1 Java.lang.Object class

Java.lang.Object is parent class of all java classes because most commonly used methods are defined in Object class and provided all these methods to all java classes directly.

If our class doesn't extend any class then only our class is the **direct child class of Object**



If our class extend any other class then our class is the **in-direct child class of Object. So it is Multilevel inheritance, but not Multiple inheritance**



Java won't provide support for Multiple Inhetitance directly or indirectly

Total 11 methods are in Obejct Class (**Remember 11 methods ...11 number Important**)

1. **Public String toString()**
2. **Public native int hashCode()**
3. **Public Boolean equals()**
4. **protected native Object clone() thows CloneNotSupportedExcception**
5. **protected void finalize() throws Throwable**
6. **public final Class getClass()**
7. **public final void wait() throws IntrupptedException**
8. **public final void wait(long timeout) throws IntrupptedException**
9. **public final void wait(long timeout, int nanos) throws IntrupptedException**
10. **public final native void notify()**
11. **public final native void notifyAll()**

Private static void registerNatives() only required for Object class & not available to child classes

3.2 java.lang.String

String is basically an object that represents sequence of char values. Generally, string is a sequence of characters. But in java, string is an object that represents a sequence of characters. String class is used to create string object

String are classified into two types based on their characteristics. They are Immutable, Mutable.

Immutable → we can't change

- ***String***

Mutable → we can change

- ***StringBuffer***
- ***StringBuilder***

a. Creating String Object

There are two ways to create String object:

- Using **string literal**
- Using **new keyword**

1. Using String literal

Java String literal is created by using double quotes. For these storage location is **Sting Constast Pool**

```
String s1="Hyderabad";
```

2. Using new literal

Using new keyword we can create String Object. For these storage location is **Heap Memory**

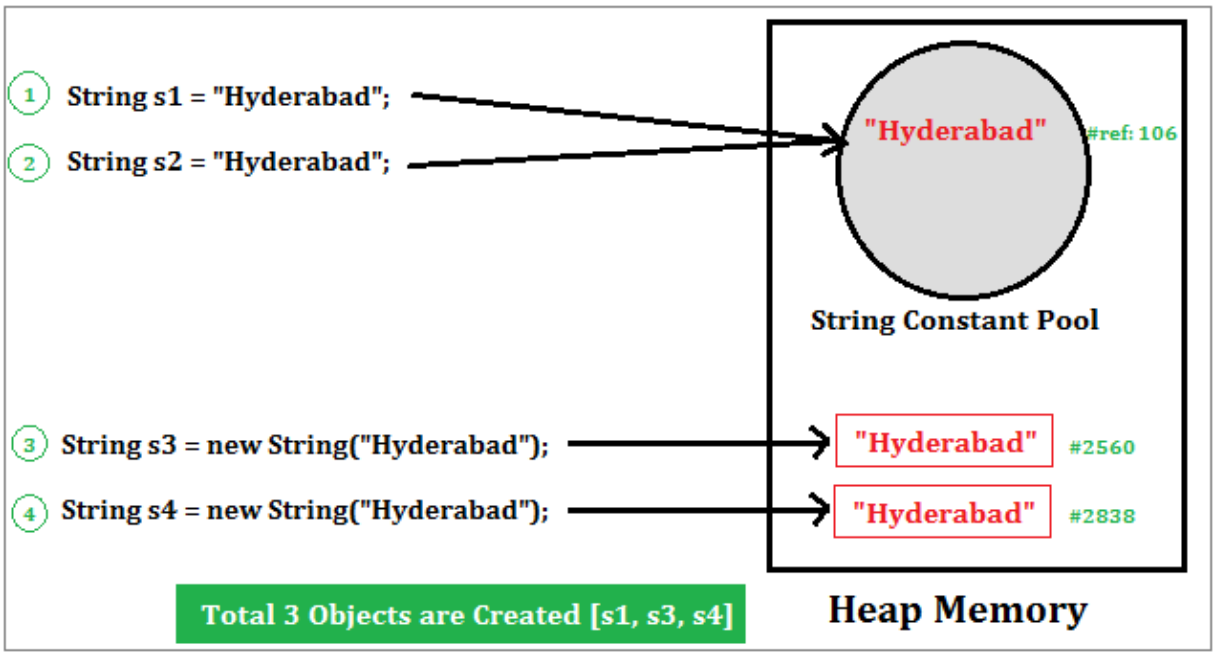
```
String s3 = new String("Hyderabad");
```

b. String Constant Pool

String Literals are stored in a special memory area known as String constant pool. Each time you create a string literal, the JVM checks the string constant pool first. If the string already exists in the pool, that String Literal reference is returnd. If string doesn't exist in the pool, a new string instance is created and placed in the pool.

```
String s1 = "Hyderabad";
String s2 = "Hyderabad";
```

In the above example only one object will be created.



1. Firstly JVM will not find any string object with the value "**Hyderabad**" in string constant pool, so it will **create a new object.**

2. After that it will find the string with the value "Hyderabad" in the pool, it will **not create new object** but will return the reference to the same instance**. (Ex. 106)**

3. String s3 = new String("Hyderbad") , here **new String Object** is created with new reference in **Heapmemory** & it wont comapiar String Literals eventhrogh content is same

4. String s4 = new String("Hyderbad") , here **new String Object** is created with new reference in **Heapmemory** & it wont comapiar String Literals eventhrogh content is same

```
public class hyd {
	public static void main(String[] args) {
		String s1 = "Hyderabad";
		String s2 = "Hyderabad";
		String s3 = new String("Hyderabad");
		String s4 = new String("Hyderabad");
		System.out.println(s1==s2);//true
		System.out.println(s1==s3);//false
		System.out.println(s3==s4);//false
		}
}
```

C.String comparsion

We can compare String using

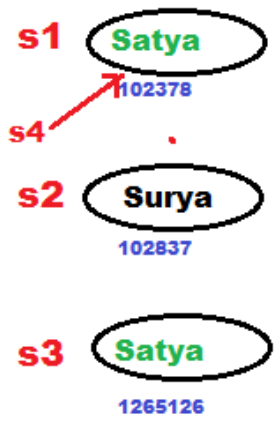
- **equals() method**
- **== operator**

.equals() method has two levels of comparison

1. equals () at Object comparision level

- If we are comparing **non-String Objects** eqauls() method it **comapiars refrences of Objects**.
- **It is same as "== " Operator**

```
public class Student {
	String name;

	public Student(String name) {
		this.name = name;
	}

	public static void main(String[] args) {
		Student s1 = new Student("Satya");
		Student s2 = new Student("Surya");
		Student s3 = new Student("Satya");
		Student s4 = s1;
		System.out.println(s1.equals(s2)); //F
		System.out.println(s1.equals(s3)); //F
		System.out.println(s1.equals(s4)); //T
		System.out.println("----------");
		System.out.println(s1==s2);
		System.out.println(s1==s3);
		System.out.println(s1==s4);
	}
 }
```




```
false
false
true
----------
false
false
true
```

2. equals () at String Comparision Level

- If we are comparing **String data on** .eqauls() method **comapiars only content**.
- **References are doesn't matter**.
- But == always compairs **references**

```
public class Studen {
	public static void main(String[] args) {
		String s1 =  "Satya" ;
		String s2 =  "Satya" ;
		String s3 = new String("Satya");
		String s4 = s1;

		System.out.println(s1.equals(s2));
		System.out.println(s1.equals(s3));
		System.out.println(s1.equals(s4));
		System.out.println("----------");
		System.out.println(s1==s2);
		System.out.println(s1==s3);
		System.out.println(s1==s4);
	}
}
```

OutPut

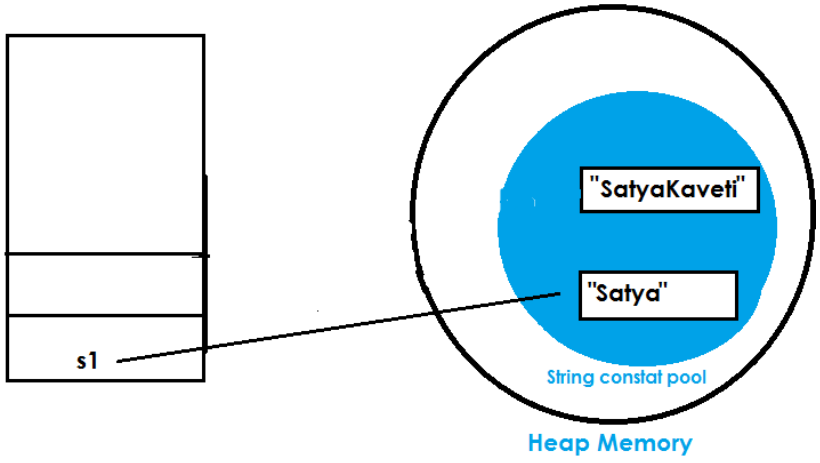
```
true
true
true
----------
true
false
true
```

B.Immutable String Objects

- In java, string objects are **immutable**.
- Immutable simply means **unmodifiable** or **unchangeable**.
- Once string object is created its **data or state can't be changed but a new string object is created**

```
public class ImmutableDemo {
public static void main(String[] args) {
	String s1 = "Satya";
	System.out.println("Before Concat: "+s1);
	System.out.println(s1.concat("Kaveti"));
	System.out.println("After Concat : "+s1)
}
}
```

```
Before Concat: Satya
SatyaKaveti
After Concat : Satya
New Object : SatyaKaveti
```

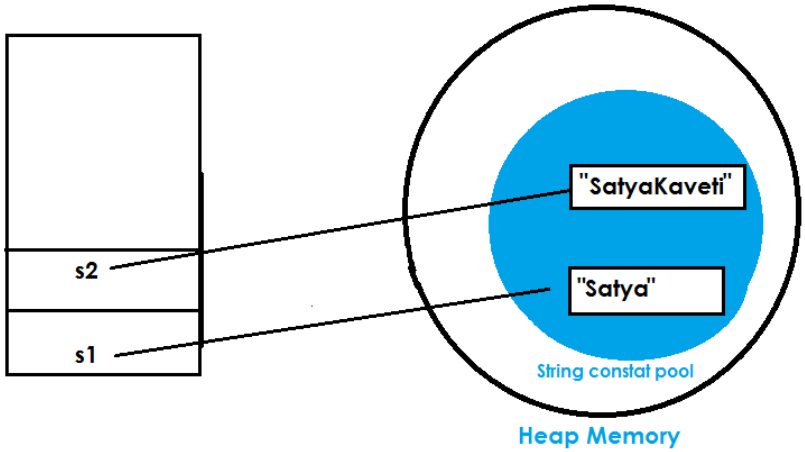


As you can see in the above figure that two objects are created but s reference variable still refers to "Satya" not to "SatyaKaveti"

Here we are Storing concated String into new String Object

```
public class ImmutableDemo {
public static void main(String[] args) {
	String s1 = "Satya";
	//====Storing in New Object============
	String s2 = s1.concat("Kaveti");
	System.out.println("New Object : "+s2);
}
}
```

```
New Object : SatyaKaveti
```



Why string objects are immutable in java?

Because java uses the concept of string literal.Suppose there are 5 reference variables,all referes to one object "Hyderabad".If one reference variable changes the value of the object, it will be affected to all the reference variables. That is why string objects are immutable in java

String Methods

	`String(String s)`	*create a string with the same value as* `s`
`int`	`length()`	*number of characters*
`char`	`charAt(int i)`	*the character at index* `i`
`String`	`substring(int i, int j)`	*characters at indices* `i` *through* `(j-1)`
`boolean`	`contains(String substring)`	*does this string contain* `substring`*?*
`boolean`	`startsWith(String pre)`	*does this string start with* `pre`*?*
`boolean`	`endsWith(String post)`	*does this string end with* `post`*?*
`int`	`indexOf(String pattern)`	*index of first occurrence of* `pattern`
`int`	`indexOf(String pattern, int i)`	*index of first occurrence of* `pattern` *after* `i`
`String`	`concat(String t)`	*this string with* `t` *appended*
`int`	`compareTo(String t)`	*string comparison*
`String`	`toLowerCase()`	*this string, with lowercase letters*
`String`	`toUpperCase()`	*this string, with uppercase letters*
`String`	`replaceAll(String a, String b)`	*this string, with* `a`*s replaced by* `b`*s*
`String[]`	`split(String delimiter)`	*strings between occurrences of* `delimiter`
`boolean`	`equals(Object t)`	*is this string's value the same as* `t`*'s?*

Mutable String Objects

A string that can be modified or changed is known as mutable string.We have two java classes which are categorize as Mutable.

- **StringBuffer**
- **StringBuilder**

Whose values can change!

3.3 StringBuffer

The StringBuffer class in java is same as String class except it is **mutable and synchronized**.

StringBuffer(): creates an empty string buffer with the **initial capacity of 16**.

StringBuffer(String str): creates a string buffer with the **specified string**.

StringBuffer(int capacity): creates an empty string buffer with the **specified capacity as length**.

Methods	
StringBuffer append(char c)	append c to the end of the StringBuffer
StringBuffer append(int i)	convert i to characters, then append them to the end of the StringBuffer
StringBuffer append(long L)	convert L to characters, then append them to the end of the StringBuffer
StringBuffer append(float f)	convert f to characters, then append them to the end of the StringBuffer
StringBuffer append(double d)	convert d to characters, then append them to the end of the StringBuffer
StringBuffer append(String s)	append the characters in s to the end of the StringBuffer
int capacity()	return the current capacity (capacity will grow as needed).
char charAt(int index)	get the character at index.
StringBuffer delete(int start, int end)	delete characters from start to end-1
StringBuffer deleteCharAt(int index)	delete the character at index
StringBuffer insert(int index, char c)	insert character c at index (old characters move over to make room).
StringBuffer insert(int index, String st)	insert characters from st starting at position i.
StringBuffer insert(int index, int i)	convert i to characters, then insert them starting at index.
StringBuffer insert(int index, long L)	convert L to characters, then insert them starting at index.
StringBuffer insert(int index, float f)	convert f to characters, then insert them starting at index.
StringBuffer insert(int index, double d)	convert d to characters, then insert them starting at index.
int length()	return the number of characters presently in the buffer.
StringBuffer reverse()	Reverse the order of the characters.
void setCharAt(int index, char c)	set the character at index to c.
String toString()	return a String object containing the characters in the StringBuffer.

Example

We must use StringBuffer() constructor to create mutable String Object

```
String s        = new String("Some Text")
```

Wrong

```
StringBuffer sf = new StringBuffer("Some Text");
```

Correct

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

		StringBuffer s = new StringBuffer("Satya Kaveti");
		System.out.println(s);

		s.append(" From Hyderabad");
		System.out.println("APPEND ==> " + s);

		s.insert(0, "i am ");
		System.out.println("INSERT ==> " + s);

		System.out.println("INDEX ==> " + s.indexOf("Hyderabad"));

		s.replace(23, 32, "VIJAYAWADA");
		System.out.println("REPLAC ==> " + s);

		s.delete(0, 10);
		System.out.println("DELETE ==> " + s);

		System.out.println("Length ==> " + s.length());
		System.out.println("CAPACITY==> " + s.capacity());

		System.out.println("REVERSE ==> " + s.reverse());

	}
}
```

```
Satya Kaveti
APPEND ==> Satya Kaveti From Hyderabad
INSERT ==> i am Satya Kaveti From Hyderabad
INDEX ==> 23
REPLAC ==> i am Satya Kaveti From VIJAYAWADA
DELETE ==>  Kaveti From VIJAYAWADA
Length ==> 23
CAPACITY==> 58
REVERSE ==> ADAWAYAJIV morF itevaK
```

Capacity() method : default capacity of the buffer is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.

3.4 StringBuilder

Java StringBuilder class is used to create mutable (modifiable) string. The Java StringBuilder class is same as StringBuffer class except that it is **non-synchronized**. It is available since JDK 1.5.

Methods are same as StringBuffer class

Example

```
public class StringBuilderEx {
	public static void main(String[] args) {
		StringBuilder s = new StringBuilder("SATYA");
		s.append(" KAVETI");
		System.out.println(s);
	}
}
```

3.5 String VS StringBuffer VS StringBuilder

Characteristic	String	StringBuffer	StringBuilder
Storage Area	Constant String Pool	Heap	Heap
Mutable	No	Yes	Yes
Thread Safe	Yes	Yes	No
Performance	Fast	Very slowly	Fast

Example: Checking Fastness

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

		long starttime = System.currentTimeMillis();
		StringBuffer sb = new StringBuffer("SATYA");
		for (int i = 0; i < 11500000; i++) {
			sb = sb.append("KEVTI ");
		}
System.out.println("StringBuffer TIME==>"+(System.currentTimeMillis()-starttime)+"  ms");
		starttime = System.currentTimeMillis();
		StringBuilder sl = new StringBuilder("SATYA");
		for (int i = 0; i < 11500000; i++) {
			sl = sl.append("KEVTI ");
		}
System.out.println("StringBuilder TIME==>"+(System.currentTimeMillis()-starttime)+"  ms");
	}
}
```

```
StringBuffer  TIME ==>   503  ms
StringBuilder TIME ==>   254  ms
```

IV. Java Exception Handling

Exceptional handling is a mechanism of converting system error messages into user friendly messages.

Errors are of two types. They are compile time errors and run time errors.

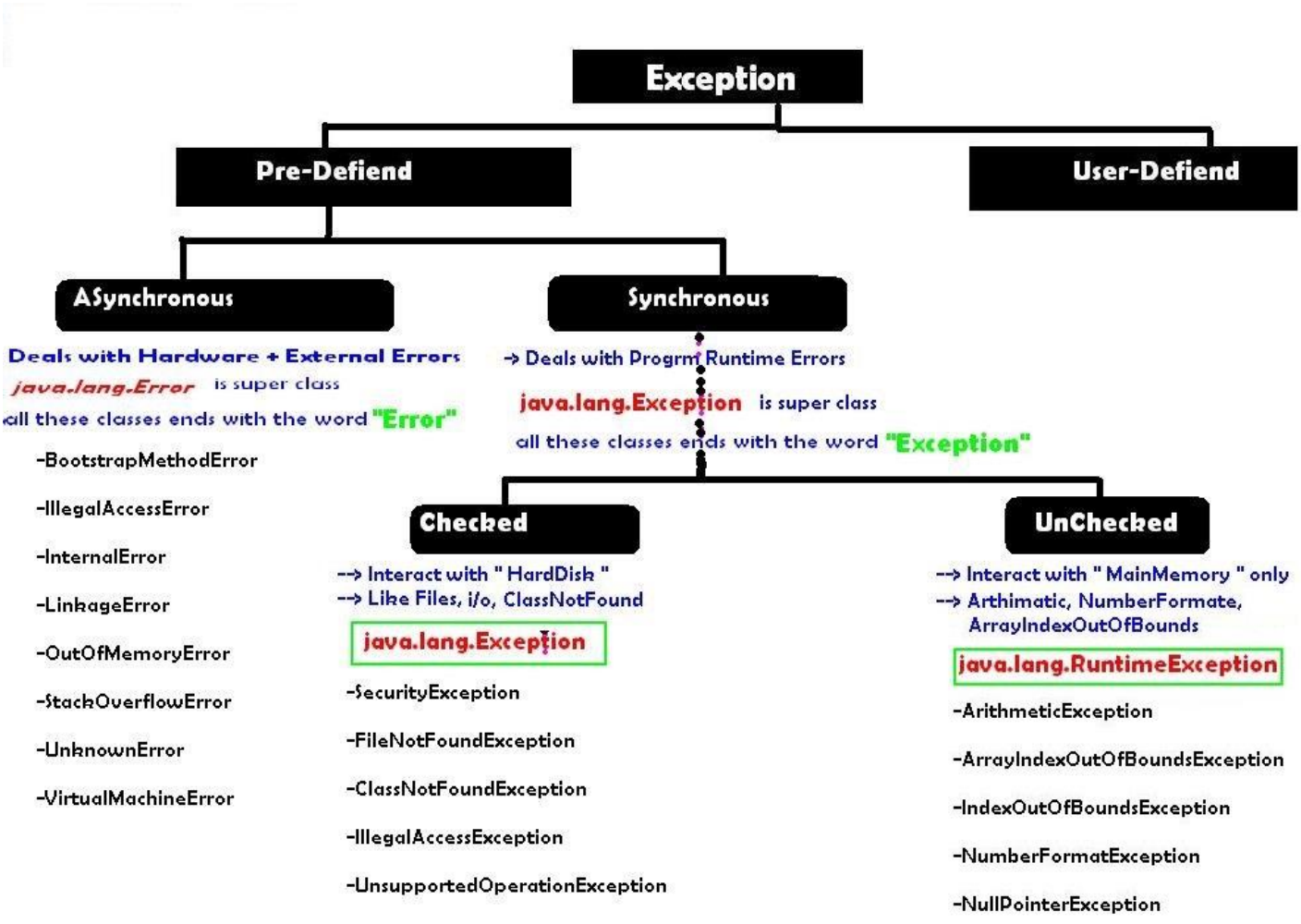
- **Compile time** errors are those which are occurring because of poor understanding of the language.
- **Run time** errors are those which are occurring in a program when the user inputs invalid data.

The run time errors must be always converted by the JAVA programmer into user friendly messages by using the concept of **exceptional handling**.

1. Types of Exceptions

We have 3 types of exceptions

1. Checked Exception
2. Unchecked Exception
3. Error



1. **Checked Exception**

   A checked exception is one which always deals with compile time errors. User must handle, because these are programmatic exceptions. **java.lang.Exception** is the super class.

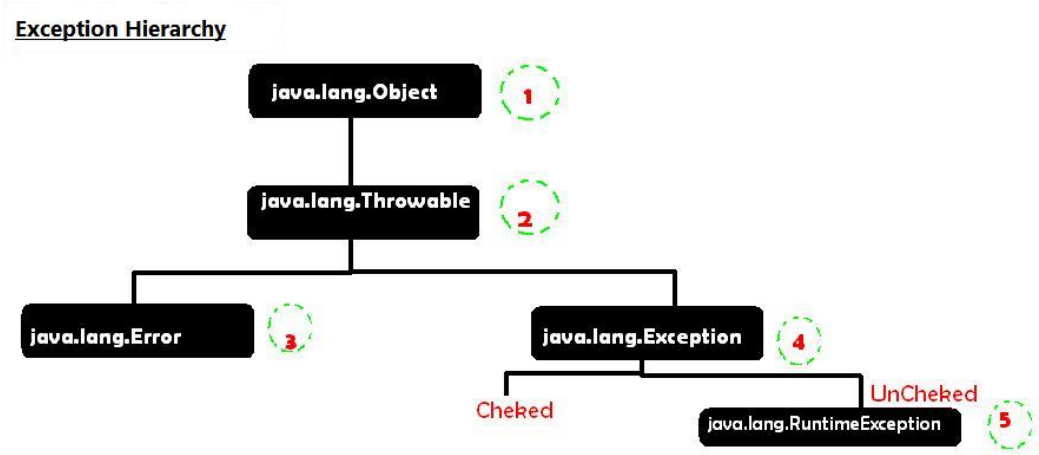
2. **Unchecked Exception**

   An unchecked exception is one which always deals with run time errors. User don't need handle, because these are not programmatic exceptions. . **java.lang.RunTimeException** is the super class.

3. **Error**

   Error is Hardware exceptions. **Java.lang.Error** is the Super class

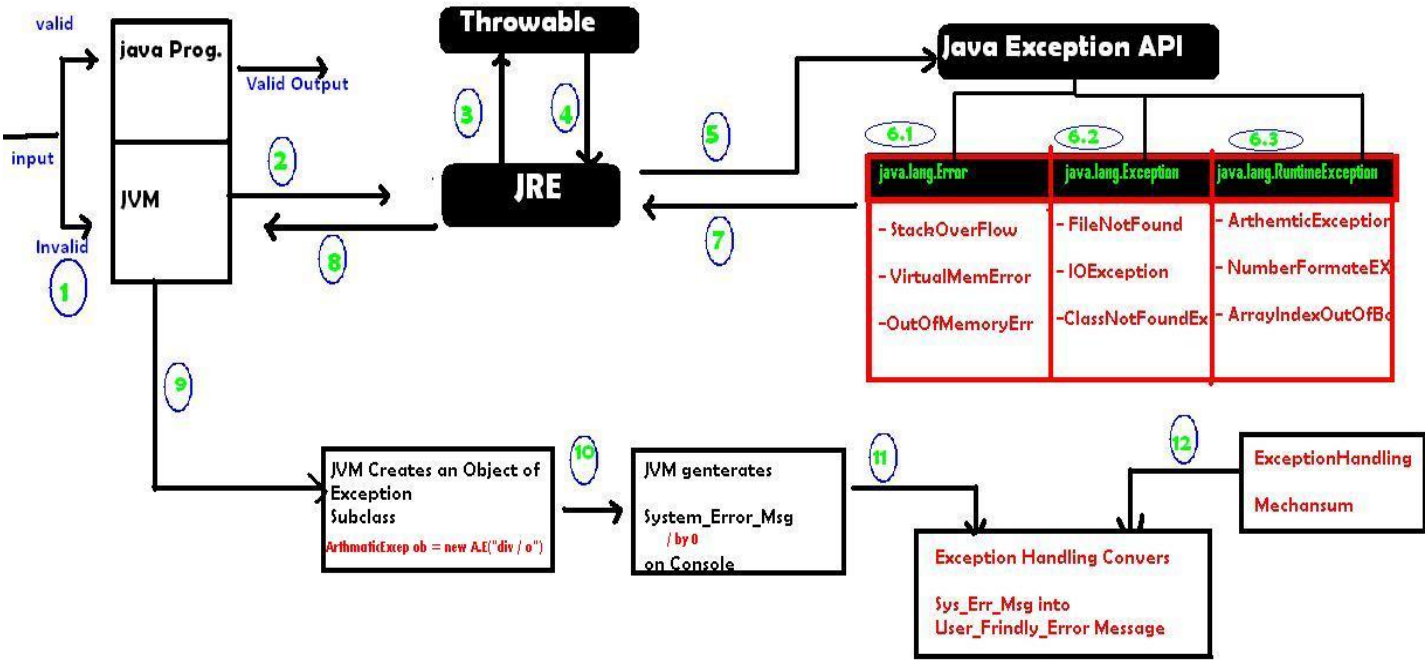
2. Exception Hierarchy



1. ***java.lang.object*** is the super class for all java prog's & it provides Garbage Collector facility
2. ***java.lang.Throwable*** is the super class for All Exception Classes in java.It Describes what type of Exception is Occured in our Java Programme
3. ***java.lang.Error*** is the Super Class for All "Asynchronos Exceptions ", Dealing with Hardware, External Errors
4. ***java.lang.Exception*** is the Super Class for All "Synchronos Exceptions ", Specially for Checked, Dealing with HardDisc
5. ***java.lang.RuntimeExeption*** is the Super Class for All "UnChecked Exceptions", Dealing with Primary Memory

3. Internal Flow of Exception Handling

Internal Flow of ExceptionHandling



1. JVM cannot process the irrelevant input.

2. Since JVM is unable to process by user input, hence it can contact to JRE for getting an appropriate exception class.

3. JRE contacts to **java.lang.Throwable** for finding type of exception

4. **java.lang.Throwable** decides what type of exception it is and pass the message to JRE.

5. JRE pass the type of exception to JAVA API.

6. [6.1 & 6.2] From the JAVA API either **java.lang.Error** class or **java.lang.Exception** class will found an appropriate sub class exception.

7. Exception API returns Exception Subclass to JRE

8. JRE will give an appropriate exception class to JVM.

9. JVM will create an object of appropriate exception class which is obtained from JRE

10. And it generates system error message.

11. In order to make the program very strong (robust), JAVA programmer must convert the system error messages into user friendly messages by using the concept of exceptional handling

4. Common scenarios where exceptions may occur

1. If we divide any number by zero, there occurs an **ArithmeticException**.

```
int a=50/0;//ArithmeticException
```

2. If we have null value in any variable, performing any operation occurs a **NullPointerException**.

```
String s=null;
System.out.println(s.length());//NullPointerException
```

3) The wrong formatting of any value, may occur **NumberFormatException**.

```
String s="abc";
int i=Integer.parseInt(s);//NumberFormatException
```

4) If you are inserting any value in the wrong index, it would result **ArrayIndexOutOfBoundsException**

```
int a[]=new int[5];
a[10]=50; //ArrayIndexOutOfBoundsException
```

5. Exception Handling

There are 5 keywords used in java exception handling.

1. **try**
2. **catch**
3. **finally**
4. **throws**
5. **throw**

Syntax

```
try
{
Block of statements which are to be monitored by JVM at run time (or problematic errors);
}
catch (Type_of_exception1 object1)
{
Block of statements which provides user friendly messages;
}
catch (Type_of_exception2 object2)
{
Block of statements which provides user friendly messages;
}
.
.
.
catch (Type_of_exception3 object3)
{
Block of statements which provides user friendly messages;
}
finally
{
Block of statements which releases the resources;
}
```

1. **try**

- Try block is used to enclose the code that might throw an exception.
- It must be used within the method
- try block must be followed by either catch or finally block
- If any exception is taking place the control will be jumped automatically to appropriate catch block.
- If any exception is taking place in try block, execution will be terminated and the rest of the statements in try block will not be executed at all and the control will go to catch block

2. **catch**

- Catch block is used to handle the Exception.
- It must be used after the try block only
- We can use multiple catch block with a single try.
- If we write 'n' number of catch's , then only one catch will be executing at any point
- After executing appropriate catch block, control never goes to try block even if we write return statement

3. Finally

- This is the block which is executing compulsory whether the exception is taking place or not.
- This block contains statements like releases the resources are opening files, opening databases, etc.
- Writing the finally block is optional.

Program without exception Handling

```
public class A {
public static void main(String[] args) {
	String s1=args[0];  //if we are not passing args
	String s2=args[1];
	int n1=Integer.parseInt (s1);
	int n2=Integer.parseInt (s2);
	int n3=n1/n2;
	System.out.println ("DIVISION VALUE = "+n3);
}
}
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
	at excep.A.main(A.java:5)
```

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

- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Prints the Line Number

with exception Handling

```
public class A {
	public static void main(String[] args) {
		try {
			String s1 = args[0];
			String s2 = args[1];
			int n1 = Integer.parseInt(s1);
			int n2 = Integer.parseInt(s2);
			int n3 = n1 / n2;
			System.out.println("DIVISION VALUE = " + n3);
		} catch (ArithmeticException Ae) {
			System.out.println("DONT ENTER ZERO FOR DENOMINATOR...");
		} catch (NumberFormatException Nfe) {
			System.out.println("PASS ONLY INTEGER VALUES...");
		} catch (ArrayIndexOutOfBoundsException Aioobe) {
			System.out.println("PASS DATA FROM COMMAND PROMPT...");
		} finally {
			System.out.println("I AM FROM FINALLY...always Exceute");
		}
	}
}
```

```
PASS DATA FROM COMMAND PROMPT...
I AM FROM FINALLY...always Exceute
```

- All catch **blocks must be ordered** from most specific to most general i.e. catch for **ArithmeticException** must **come before catch for Exception**
- At a time only one Exception is occurred and at a **time only one catch block is executed at a time**
- Java finally block must be followed **by try or catch block**
- If you don't handle exception, before terminating the program, JVM executes finally block if it exists.
- For each try block there can be **zero or more catch blocks, but only one finally block**.
- **The finally block will not be executed** if program exits(either by calling **System.exit()** or by causing a **fatal error** that causes the process to abort

```
public class A {
	public static void main(String[] args) {
		try {
			int a = 150 / 0;
		} finally {
			System.out.println("See What iam ");
		}
	}
}
```

```
Exception in thread "main" See What iam
java.lang.ArithmeticException: / by zero at excep.A.main(A.java:6)
```

4. Throws

This is the keyword which gives an indication to the calling function to keep the called function **under try and catch blocks**.

It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained

```
class Cal {
	public void div(String a, String b) throws ArithmeticException,NumberFormatException {
		int c = Integer.parseInt(a) / Integer.parseInt(b);
	}
}

public class A {
	public static void main(String[] args) {
		Cal ob = new Cal();
		try {
			ob.div("a", "b");
		} catch (ArithmeticException e) {
			System.out.println("Divide By Zero");
		} catch (NumberFormatException e) {
			System.out.println("Enter Only INT's");
		} catch (Exception e) {
			System.out.println(" Some Other " + e);
		}
	}
}
```

```
Enter Only INT's
```

In above **throws** `ArithmeticException,NumberFormatException` Indicates it may throws these exceptions so please put `ob.div(str,str)` method in try,catch block

Number of ways to find details of the exception

1.Using an object of java.lang.Exception

```
try
{
int x=Integer.parseInt ("10x");
}
catch (Exception e)
{
   System.out.println (e); // java.lang.NumberFormatException : for input string 10x
}                                name of the exception      ||  nature of the message
```

2.Using printStackTrace method

```
e.printStackTrace (); // java.lang.ArithmeticException : / by zero : at line no: 4
                         name of the exception         || nature of the message || line number
```

3.Using getMessage method:

```
System.out.println (e.getMessage ()); // / by zero
                                          nature of the message
```

5. Throw

Throw keyword is used to explicitly throw an exception.

In above we didn't create any Exception class Object in throws because JVM automatically creates Objects. If you want to create Exception class object manually and throw exception using **throw** keyword

Example

Suppose take a Marks class, if marks less than 35 we are throwing exception manually using throw keyword

```
public class Marks {

	public void pass(int marks) {
		if (marks < 35) {
			throw new ArithmeticException("You are Failed");
		} else {
			System.out.println(" You are Pass : " + marks);
		}
	}

	public static void main(String[] args) {
		Marks m = new Marks();
		m.pass(26);
	}
}
```

```
Exception in thread "main" java.lang.ArithmeticException: You are Failed
	at excep.Marks.pass(Marks.java:9)
	at excep.Marks.main(Marks.java:18)
```

6. User Defined Exceptions

User defined exceptions are those which are developed by JAVA programmer as a part of Application development for dealing with specific problems such as negative salaries, negative ages, etc

3 Steps to developing user defined exceptions

1. Choose the appropriate user defined class must extends either **java.lang.Exception** or **java.lang.RunTimeException** class.

2. That class must contain a **parameterized Constructor by taking string as a parameter**.

3. Above constructor must call super constructor with string Ex : **super(s)**

Example

For implementing example we must create 3 classes

1. ***User defined Exception class***
2. ***A class with a method which throws User defined Exception***
3. ***Main class which calls above method***

1. User Defined Exception class ➔ 1.Extends Exception || 2.Constructor(s) || 3.Super(s)

```
package excep;

public class NegativeNumberException extends Exception {
    public NegativeNumberException(String s) {
           super(s);
    }
}
```

2. A class with a method which throws User defined Exception ➔ throws & throw

```
public class Salary {
    public void show(int sal) throws NegativeNumberException {
           if (sal < 0) {
                  throw new NegativeNumberException("Salary Should be >1");
           } else {
                  System.out.println("Your Sal is :" + sal);
           }
    }
}
```

3. Main class which calls above method

```
public class UserMain {
    public static void main(String[] args) {
           Salary salary = new Salary();
           try {
                  salary.show(-100);
           } catch (NegativeNumberException e) {
                  e.printStackTrace();
           }
    }
}
```

```
excep.NegativeNumberException: Salary Should be >1
    at excep.Salary.show(Salary.java:8)
    at excep.Salary.main(Salary.java:18)
```

7. Automatic Resource Management and Catch block

Java 7 one of the feature was improved catch block where we can catch multiple exceptions in a single catch block. The catch block with this feature looks like below:

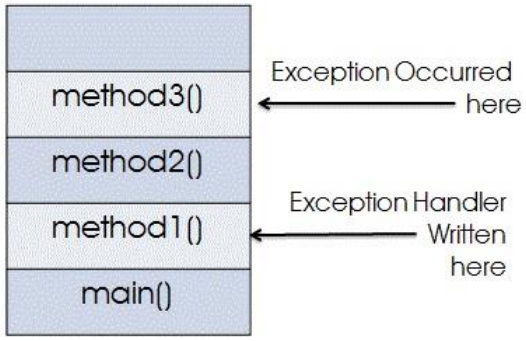
```
catch(IOException | SQLException | Exception ex){
    logger.error(ex);
    throw new MyException(ex.getMessage());
}
```

8. Java Exception propagation

If an exception is occurred in method 1 is not handled, it drops down to its previous methods and handled at method 4 or some other method is known as Exception propagation

When exception is occurred at the top of the stack and no exception handler is provided then exception is propagated.it is only applicable for **Unchecked Exceptions**

```
public class Propagation {

	void method3() {
		int result = 100 / 0; // Exception Gere
	}
	void method2() {
		method3();
	}
	void method1() {
		try {
			method2();
		} catch (Exception e) {
			System.out.println("Exception is handled here");
		}
	}
	public static void main(String args[]) {
		Propagation obj = new Propagation();
		obj.method1();
		System.out.println("Continue with Normal Flow...");
	}
}
```



You must remember one rule of thumb that – "**Checked Exceptions are not propagated in the chain**". Thus we will get compile error

9. Difference between throw and throws in Java

No.	throw	throws
1)	Java throw keyword is used to **explicitly throw** an exception.	Java throws keyword is used to **declare an exception.**
2)	Checked exception **cannot be propagated** using throw only.	Checked exception **can be propagated** with throws.
3)	Throw is **followed by an instance.**	Throws is **followed by class.**
4)	Throw is **used within the method.**	Throws is **used with the method signature.**
5)	You **cannot throw multiple exceptions**.	You **can declare multiple exceptions**

10. Difference between final, finally and finalize

No.	final	finally	finalize
1)	Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.	Finally is used to place important code, it will be executed whether exception is handled or not.	Finalize is used to perform clean up processing just before object is garbage collected.
2)	Final is a keyword.	Finally is a block.	Finalize is a method.

11. ExceptionHandling with MethodOverriding in Java

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

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

2. **If the superclass method declares an exception**

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

12. Wait and answer these

1. **Explain about Exception Handling with an example?**
2. **What is the use of finally block in Exception Handling?**
3. **In what kind of scenarios, a finally block is not executed?**
4. **Is a finally block executed even when there is a return statement in the try block?**
5. **Is a try block without corresponding catch block allowed?**
6. **Explain the hierarchy of Exception related classes in Java?**
7. **What is difference between an Error and an Exception?**
8. **What is the difference between a Checked Exception and an Un-Checked Exception?**
9. **How do you throw a Checked Exception from a Method?**
10. **How do you create a Custom Exception Classes?**
11. **How should the Exception catch blocks be ordered?**
12. **What are the new features related to Exception Handling introduced in Java7?**

13. Interview questions

1. **Can we keep the statements after finally block If the control is returning from the finally block itself?**

No, it gives unreachable code error. Because, control is returning from the finally block itself. Compiler will not see the statements after it. That's why it shows unreachable code error.

2. **What is OutOfMemoryError in Java?**

Ans : OutOfMemoryError in Java is a subclass of **java.lang.VirtualMachineError** and it's thrown by JVM when it ran out of heap memory

3. What is difference between ClassNotFoundException and NoClassDefFoundError?

1. **ClassNotFoundException :** ClassNotFoundException occurs when class loader could not find the required class in class path .
2. **NoClassDefFoundError :** This is thrown when at compile time the required classes are present , but at run time the classes are changed or removed or class's static initializes threw exceptions.

4.What happens when exception is thrown by main method?

Ans : When exception is thrown by main() method, Java Runtime **terminates** the program and **print** the exception message and stack trace in system console.

5.What are different scenarios causing "Exception in thread main"?

- *Exception in thread main* ***java.lang.NoClassDefFoundError***
- *Exception in thread main* ***java.lang.NoSuchMethodError: main***
- *Exception in thread "main"* ***java.lang.ArithmeticException***

V. Java UI (Applets/Swings)

In JAVA we write two types of programs or applications. They are **standalone applications (Local/Desktop)** and **distributed applications (web/Network)**

Initially, before Servlets come into picture above 2 types of applications are implemented using

1. **Swings** → Developing Standalone Applications
2. **Applets**→ Developing Distributed Applications

1. Applet Basics

"**An applet is a JAVA program which runs in the context of browser or World Wide Web**".

- To deal with applets we must import a package called **java.applet.*.** This package
- It only one class Applet whose fully qualified name is **java.applet.Applet.**

In **java.applet.Applet** we have four life cycle methods. They are public **void init (),** public **void start (),** public void **stop ()**, public void **destroy ()**, **void paint ()** it not a life cycle method

1. **Public void init ():**

This is the method which is called by the browser **only one time after loading the applet**. In this method we write some block of statements which will perform one time operations, such as, obtaining the resources like opening the files, obtaining the database connection, initializing the parameters, etc.

2. **Public void start ():**

   Start method will be called each and every time. In this method we write the block of statement which provides business logic.

3. **Public void stop ():**

   Stop method is called by the browser when we minimize the window. In this method we write the block of statements which will temporarily releases the resources which are obtained in init method.

4. **Public void destroy ():**

   This is the method which will be called by the browser when we close the window button or when we terminate the applet application. In this method we write same block of statements which will releases the resources permanently which are obtained in init method.

5. **Public void paint ().**

This is the method which will be called by the browser **after completion of start method**. This method is used for displaying the data on to the browser. Paint method is internally call **drawstring method**

STEPS for developing APPLET Program

1. **Import java.applet.Applet** package.
2. Choose the user defined **public class that must extends java.applet.Applet class**
3. **Overwrite the life cycle methods** of the applet if require.
4. Save the program and **compile.**
5. **Run the applet**: To run the applet we have two ways.
   - Using HTML program
   - Using applet viewer tool.

```
public class AppletDemo extends Applet {
	String s = "";

	public void init() {
		s = s + "  -INIT";
	}
	public void start() {
		s = s + "  -START";
	}
	public void paint(Graphics g) {
		g.drawString(s, 50, 50);
	}

	public void stop() {
		s = s + "  -STOP";
	}

	public void destroy() {
		s = s + "  -DESTROY";
	}
}
```

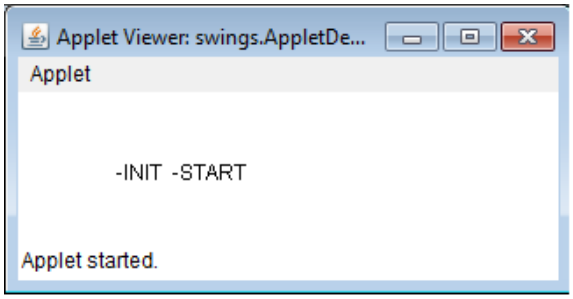
Compile the above Program, Run using any of below methods

1) Using HTML program

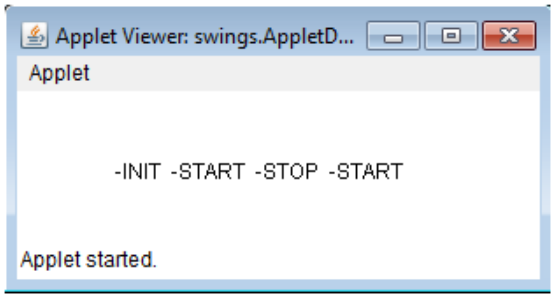
```
<APPLET code="AppletDemo " height=100 width=150>
```

2) Using applet viewer tool.

```
appletviewer AppletDemo.java
```



1. 1st Time Running



2. Minimized & Opened

2. Swing Basics

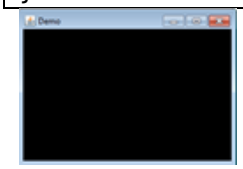
We can develop standalone applications by using AWT (old) & Swing concepts

For developing any Swing based application we need to extend either **java.awt.Frame** or **javax.swing.JFrame**

Difference between AWT and Swing

No.	Java AWT	Java Swing
1)	AWT components are **platform-dependent**.	Java swing components are **platform-independent**.
2)	AWT components are **heavyweight**.	Swing components are **lightweight**.
3)	AWT **doesn't support pluggable look and feel**.	Swing **supports pluggable look and feel**.
4)	AWT provides **less components** than Swing.	Swing provides **more powerful components** such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
5)	AWT **doesn't follows MVC**	Swing **follows MVC**.

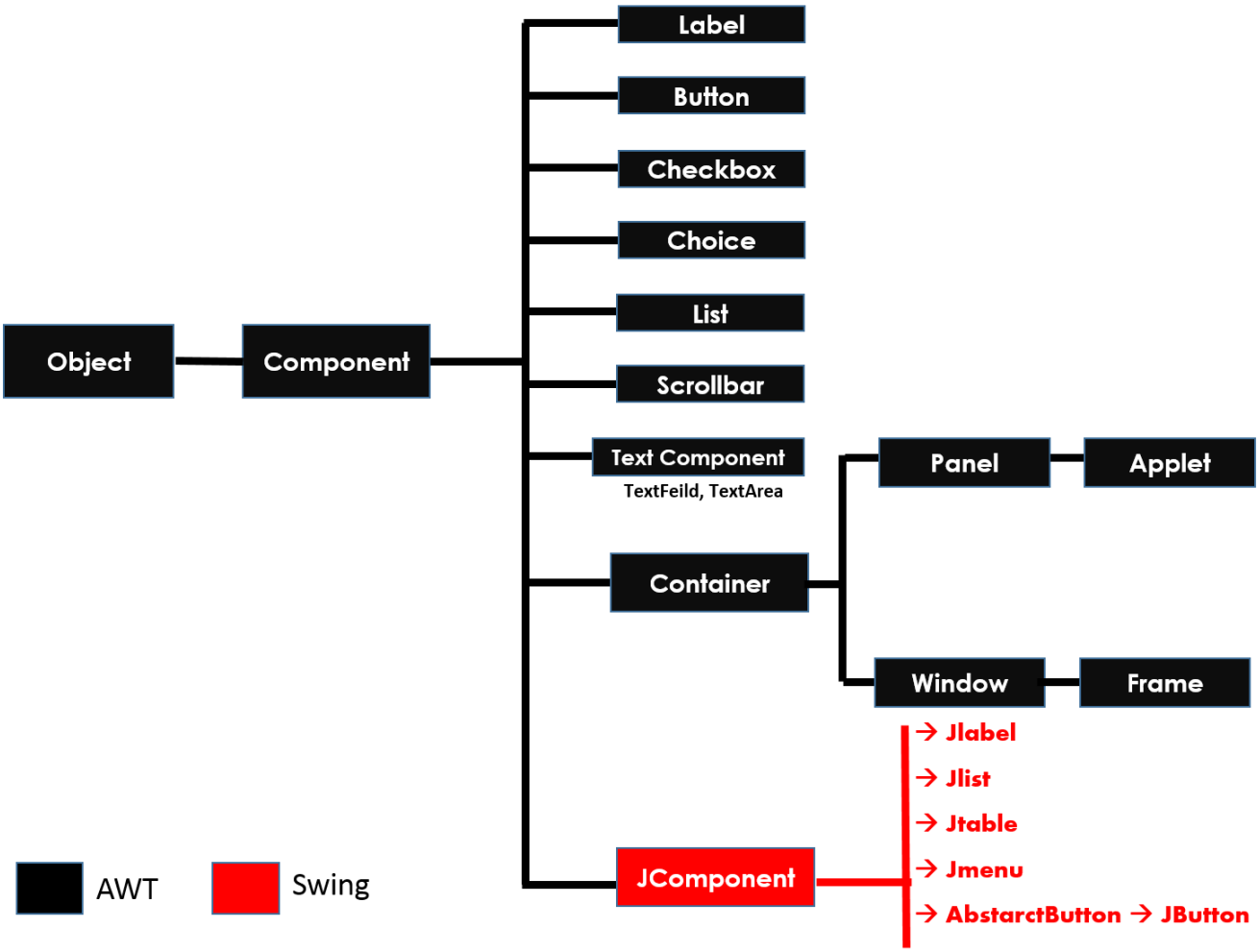
```java
public class FrameDemo extends Frame {
	public FrameDemo() {
		setTitle("Demo");
		setSize(100, 100);
		setBackground(Color.black);
		setForeground(Color.red);
		setVisible(true);
	}

	public static void main(String[] args) {
		new FrameDemo();
	}
}
```



This is very basic program. We will explain the in detail in upcoming topics☺

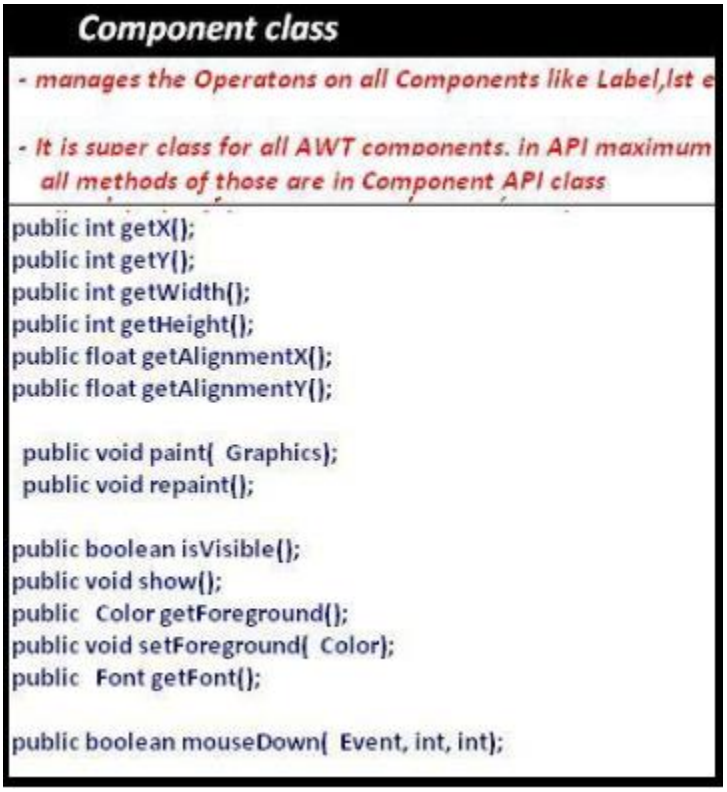
3. AWT (abstract windowing toolkit)

See, **java.awt.* & javax.swing.*** both packages hierarchy almost same. Only difference is letter **'J'**



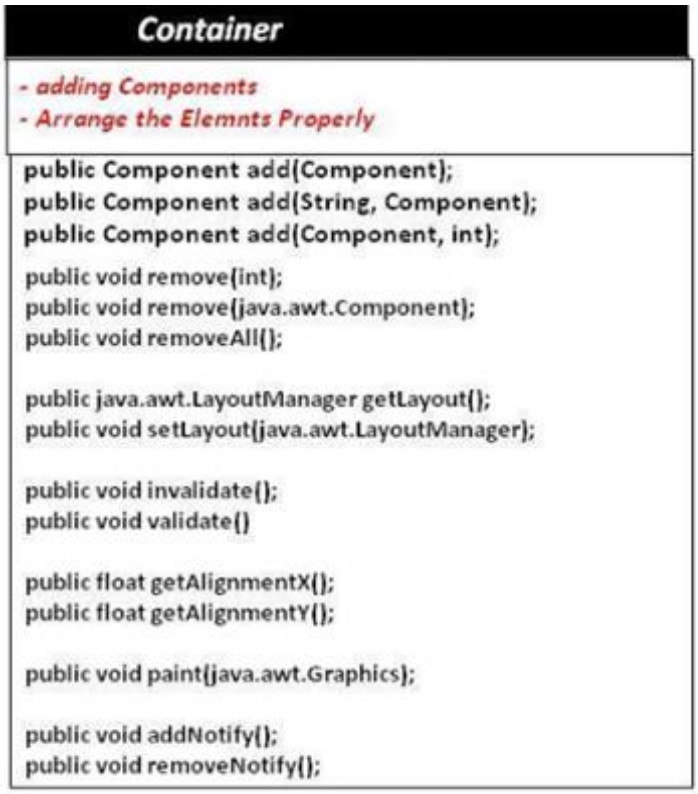
1. Component:

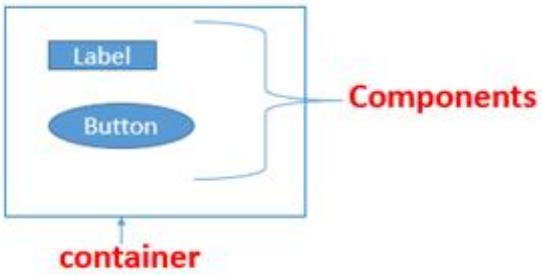
Component is any GUI Component like Label, Button, Etc



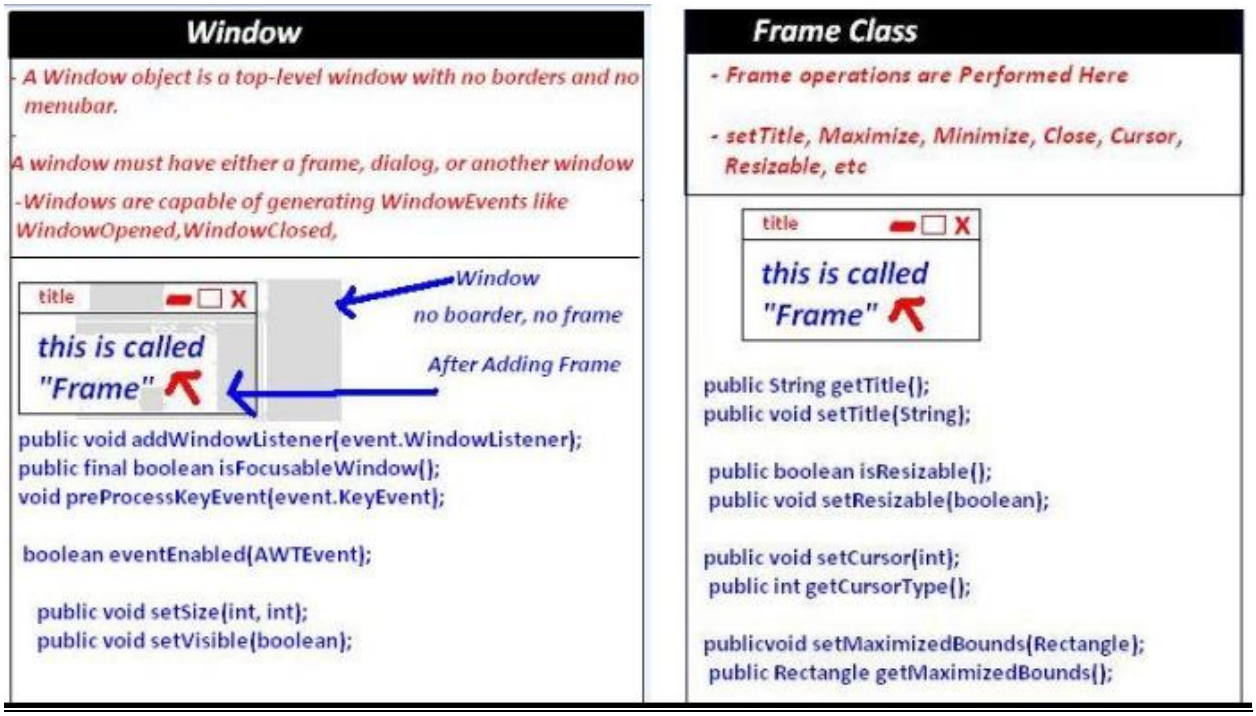
2. Container

Container is an empty space, we have to place components

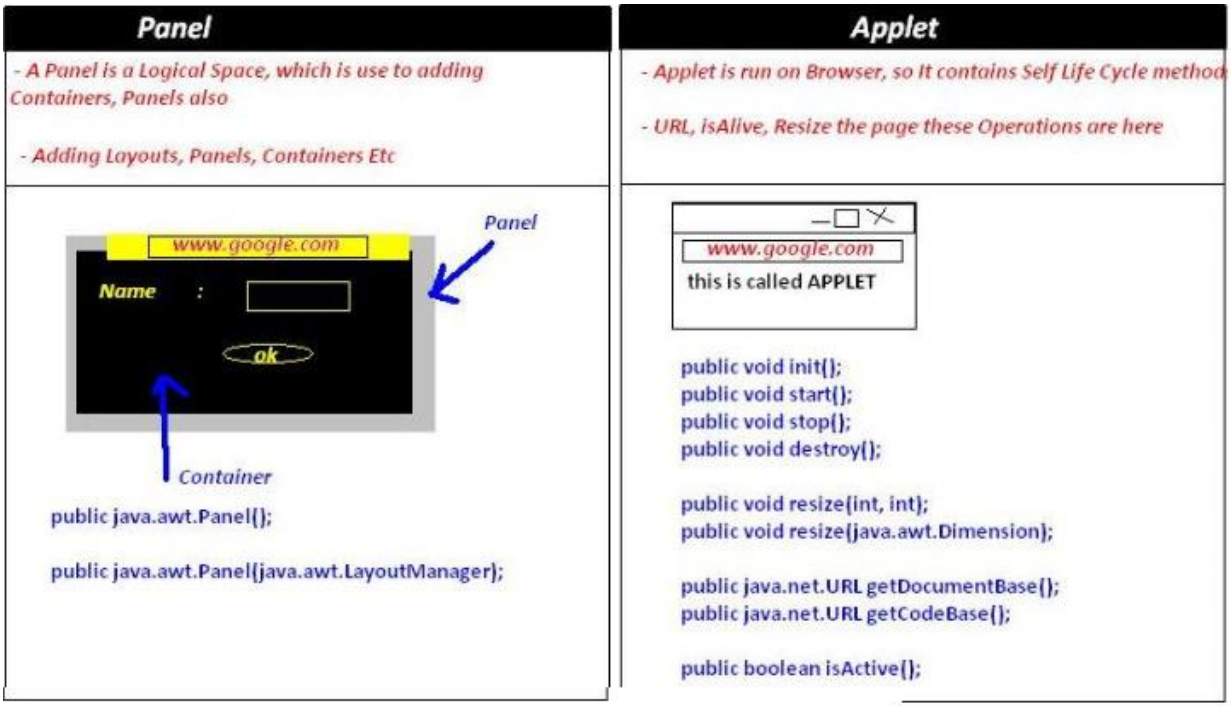




3. Window & Frame



4. Panel & Applet



4. Events Handling

As a part of GUI applications we use to create two types of components. They are **passive components** and **active components**

- **Passive component**, no interaction from the user. For example **Label**.
- **Active component** there is an interaction from the user. For example **button, check box, etc**

For developing Event handling, a class must have below steps

1. **Class which implement Listener Interface**
2. **Component must register with Listener**
3. **Get the object of Event class**
4. **Implement event method**

1. Class which implement Listener Interface

Every interactive component must have a predefined listener whose general notation is xxx listener.

Button	**java.awt.event.ActionListener**
Choice	**java.awt.event.ItemListener**
TextField	**java.awt.event.TextListener**
TextArea	**java.awt.event.TextListener**
Scrollbar	**java.awt.event.AdjustmentListener**

2 .Component must register with Listener

Each and every interactive component must be registered and unregistered with particular event and Listener. The general form of registration and un-registration methods is as follows:

```
public void addxxxListener (xxxListener);
public void removexxxListener (xxxListener);
```

Component name	Registration method	Un-registration method
Button	public void addActionListener (ActionListener);	public void removeActionListener (ActionListener);
Choice	public void addItemListener (ItemListener);	public void removeItemListener (ItemListener);
TextField	public void addTextListener (TextListener);	public void removeTextListener (TextListener);
TextArea	public void addTextListener (TextListener);	public void removeTextListener (TextListener);

3. Get the object of Event class

Whenever we interact any active component, the corresponding active component Event class object will be created. That object contains two details:

1. ***Name of the component.***
2. ***Reference of the component.***

The general form of every Event class is **xxxEvent**.

Component name	Event name
Button	**java.awt.event.ActionEvent**
choice	**java.awt.event.ItemEvent**
textField	**java.awt.event.TextEvent**
textArea	**java.awt.event.TextEvent**
scrollbar	**java.awt.event.AdjustmentEvent**

4. Implement Event method

All these methods are present in **xxxLisnter** classes. We have to implement appropriate method

Component name	Undefined method name
Button	`public void actionPerformed (java.awt.event.ActionEvent)`
Choice	`public void actionPerformed (java.awt.event.ItemEvent)`
TextField	`public void actionPerformed (java.awt.event.TextEvent)`
TextArea	`public void actionPerformed (java.awt.event.TextEvent)`

5. Components

1. Label

constructors	
`public Label()`	Constructs a Label displaying *aString* with the *alignment*, which can be one of CENTER, LEFT or RIGHT, as specified.
`public Label( String aString)`	
`public Label( String aString, int alignment)`	

instance methods	
`public void setText( String newString)`	Sets or obtains the text displayed by the Label.
`public String getText()`	
`public void setAlignment( int newAlignment)`	Sets or obtains the alignment of the Label to *newAlignment*. Will throw an IllegalArgumentException if *newAlignment* is not one of the three manifest values.
`public int getAlignment()`	

2. Button

constructors	
public Button()	Constructs a Button without a label, or with *aString* as its label.
public Button(String *aString*)	

instance methods	
public void setLabel(String *newLabel*)	Sets and obtains the label displayed by the Button.
public String getLabel()	
public void addActionListener(ActionListener *listener*)	Registers or removes a listener for the ActionEvents generated by the button. More than one listener can be registered, but there is no guarantee on the sequence they will be called.
public void removeActionListener(ActionListener *listener*)	
public void setActionCommand(String *command*)	Associates a command string with the button which will be sent with the ActionEvent.
public String getActionCommand()	

3.TextComponet

TextComponent instance methods	
public void setText(String setTo)	Sets, or obtains, the text shown in the component.
public String getText()	
public String getSelectedText()	Obtains the selected text, or obtains or sets its location within the TextComponent. The extent of the selection may also be set by the users' interactions with the peer.
public void SelectAll()	
public void Select(**int** *fromhere* **int** *tohere*)	
public void setSelectionStart(**int** *startHere*)	
public int getSelectionStart()	
public void setSelectionEnd(**int** *endHere*)	
public int getSelectionEnd()	
public void setCaretPosition(**int** *toHere*)	Sets, or obtains, the location of the insertion point. (Conventionally shown as a caret glyph.)
public int getCaretPosition()	
public void setEditable(**boolean** *editable*)	Allows the contents of the component to be changed (**true**) or not (**false**) by the user. Or obtains the value of the attribute.
public boolean isEditable()	
public void addTextListener(ItemListener *listener*)	Registers or removes a listener for the TextEvents generated by the TextComponent. More than one listener can be registered, but there is no guarantee of the sequence they are called.
public void removeTextListener(ItemListener *listener*)	

4. TextField

TextField constructors	
`public TextField()`	Creates a new empty TextField, or one with contents and number of columns, or default number of columns, as specified
`public TextField( String contents)`	
`public TextField( String contents int columns)`	

instance methods	
`public void setColumns( int thisMany)`	Sets, or obtains the number of columns visible.
`public int getColumns()`	
`public void setEchoChar( char setTo)`	Sets, or obtains, or determines, if an echo character is set. If an echo character is set then all input by the user will the be confirmed using this character.
`public boolean echoCharIsSet()`	
`public int getEchoChar()`	
`public void addActionListener( ItemListener listener)`	Registers or removes a listener for the ActionEvents generated by the TextField. More than one listener can be registered, but there is no guarantee of the sequence they will be called.
`public void removeActionListener( ItemListener listener)`	

5. Layout Managers

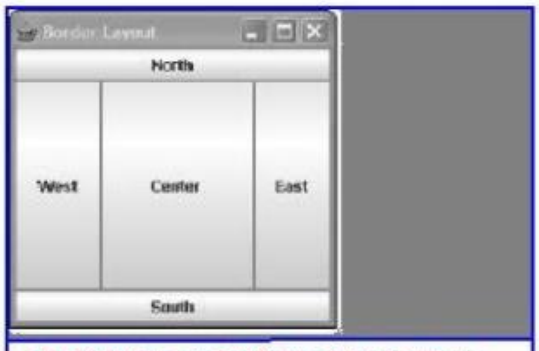
2 steps for using layout managers

1. *Create Object for Appropriate Layout to place components in the Container*

```
BoarderLayout ob = new BoarderLayout()
```

2. *Set the layout for our programe using 'setLayout()' of java.awt.Component class*

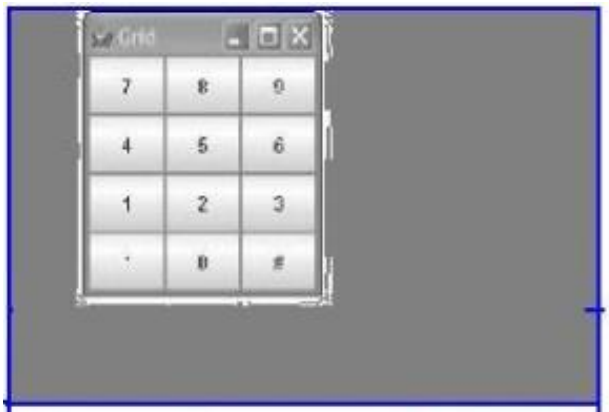
```
setLayout(ob);
```



1.Default Layout for 'Frame' Applications

1.Default Layout for 'Applet' Applications

2.Arranges the Data in 'ROW WISE'



1.Arranges the componets in 'ROW X Col'

2.Components must be of 'SameSize'

3.'No Space' btwn the components

1.Arranges the componets in 'ROW X Col

2.Components are 'SameSize' or 'Diff Size

3.'Space' btwn the components

Steps for developing awt program:

1. Import the appropriate packages.

2. Choose the appropriate class and it must **extend java.awt.Frame** and **implements** appropriate **Listener** if required.

3. Identify & **declare components** as data members in the top of the class.

4. **Set the title** for the window.

5. **Set the size** of the window.

6. **Create Objects of the components in the Constructor** which are identified in step 3.

7. **Add** the created **components to container**.

8. **Register** the events of the appropriate interactive component **with appropriate Listener**.

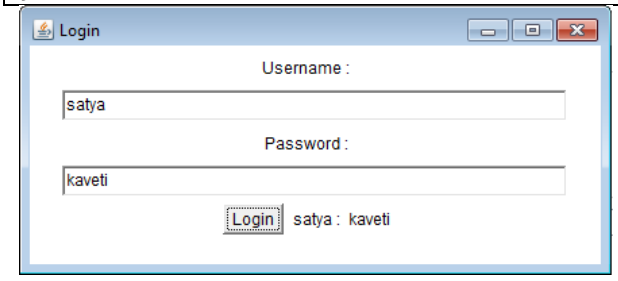
9. Make the components to be visible (**setvisible(true)**).

10. **Define the undefined methods** in the current class which is coming from appropriate **Listener**.

11. Write functionality to GUI component in that method

Example

```
public class LoginDemo extends Frame implements ActionListener {
	// 1. Decalring components
	Label l1, l2, status;
	TextField t1, t2;
	Button login;

	public LoginDemo() {
		// 4,5 Setting title & Size
		setSize(200, 200);
		setTitle("Login");

		// 6.creating Component Objects
		l1 = new Label("Username : ");
		l2 = new Label("Password : ");
		status = new Label("Status");
		t1 = new TextField(50);
		t2 = new TextField(50);
		login = new Button("Login");

		// 7.adding componets to container
		add(l1);
		add(t1);
		add(l2);
		add(t2);
		add(login);
		add(status);
		setLayout(new FlowLayout());

		// 8.reister with Listener
		login.addActionListener(this);
		setVisible(true);// 9.setvisble
	}

	@Override // 10
	public void actionPerformed(ActionEvent e) {
		if (e.getSource() == login) {
			// 11.implemeting Logic
			status.setText(t1.getText() + " :  " + t2.getText());
		}
	}

	public static void main(String[] args) {
		new LoginDemo();
	}
}
```



Similarly we have no.of components but the process of each one is similar.

Combined steps to develop FRAME and APPLET applications

1. Import appropriate packages for GUI components (java.awt.*) providing functionality to GUI components (java.awt.event.*) and for applet development (java.applet.Applet).

2. Every user defined class must extend either Frame or Applet and it must implement appropriate Listener if required.

3. Identify which components are required to develop a GUI application.

4. Use life cycle methods (init, start, destroy) in the case of applet, use default Constructor in the case of Frame for creating the components, adding the components, registering the components, etc.

5. Set the title of the window.

6. Set the size of the window.

7. Set the layout if required.

8. Create those components which are identified.

9. Add the created components to container.

10. Every interactive component must be registered with appropriate Listener.

11. Make the components to be visible in the case of Frame only.

12. Implement or define the abstract method which is coming from appropriate Listener.

VI. Java Inner Classes

If a class declared inside a class is known as Inner Class. We get below advantages if we use inner classes

- Nested classes represent a special type of relationship that is **it can access all the members (data members and methods) of outer class including private**.

- Nested classes are used to **develop more readable and maintainable code because** it logically group **classes and interfaces in one place only**

- **Code Optimization** because we write less code

We have two types of Inner classes

1) Inner Classes **(Non-Static Inner classes)**
- Member inner class
- Local inner class
- Anonymous inner class

2) Nested Classes **(Static Inner classes)**

Type	Description
Member Inner Class	A class created **within class and outside method.**
Local Inner Class	A class created **within method**.
Anonymous Inner Class	A class created for implementing interface or extending class. Its name is decided by the java compiler.
Static Nested Class	A static class **created within class.**
Nested Interface	An **interface created within class or interface**.

1. Member Inner Classes

If a non-static class is created in the class & outside the method is known as “Member Inner class”. Because it is just a member of that class

Example :

```
public class Outer{
   int a =100;
   String msg="Iam Outer Class";
   class Inner{
      int b=200;
      String inmsg="Inner class variable";
      public void show(){
         System.out.println(b+"\n"+inmsg+"\n"+msg);
      }
   }
  public static void main(String []args){
       Outer o = new Outer();
      Outer.Inner i = o.new Inner();
      i.show();
    }
}
```

Internal Working

1. instance of inner class is created inside the instance of outer class.

The java compiler creates two class files in case of inner class. The class file name of inner class is **"Outer$Inner".For** Outer.java it will create 2 .class files

- Outer$Inner.class
- Outer.class

For creating normal class object we do `OuterClass ob = new OuterClass();`

- For creating inner class object we have to add OuterClass class & Object as below
  `OuterClass.InnerClass i = o.new InnerClass();`

```
Outer o      =    new Outer();
Inner i      =    new Inner();      ☒ Wrong
Outer.Inner i =   o.new Inner();    ☑ Correct
```

Outer$Inner.Java Generated code

```
import java.io.PrintStream;

class Outer.Inner {
 int b;
 String inmsg;

 Outer.Inner() {
  this.b = 200;
  this.inmsg = "Inner class variable";
 }

 public void show() {
  System.out.println("" + this.b + "\n" + this.inmsg + "\n" + Outer.this.msg);
 }
}
```

2. Local Inner Classes

If a class is created inside the method is known as "Local Inner Class"

- Local class variable should **not Private, Public and Protected**
- Local inner **class cannot be invoked from outside of the method**.
- Local Inner class only access Final variables from outside class(until 1.7 , from 1.8 they can access non-final also)

Example : Local.java

```
public class Local {
 public void get() {
  System.out.println("Get Method");
  int a = 100;
  class Inner {
   public void show() {
    System.out.println(a);
   }
  }
  Inner ob = new Inner();
  ob.show();
 }
```

```
 public static void main(String ar[]) {
  Local ob = new Local();
  ob.get();
 }
}
```

```
Get Method
100
```

3. Anonymous Inner Classes

If a class doesn't have any Name, such type of classes are noted as Anonymous Inner classes.in real time two types of Anonymous inner classes we may implement

Class : If method of one class may return Instance we can directly implement and will get the Object

Interface: at same way a method of interface return object we directly implement to get the object

Example

```
interface A {
	public void aShow();
}

abstract class B {
	abstract void bShow();
}

public class AnnonymousDemo {

	A a = new A() {
		@Override
		public void aShow() {
			System.out.println("A show()");
		}
	};

	B b = new B() {
		@Override
		void bShow() {
			System.out.println("B show()");
		}
	};

	public static void main(String[] args) {
		AnnonymousDemo demo = new AnnonymousDemo();
		demo.a.aShow();
		demo.b.bShow();
	}
}
```

```
A show()
B show()
```

Internal Working

1.If we use anonymous inner class in our main class, internally it creates the new inner class with name **MainClass$X(x is a number)** which is

- **Extends** in case of **Class**
- **Implements** in case of **Interface**

In above class the complile generates Anonymous inner class as below

Class A : Inner class

```
class AnnonymousDemo$1  implements A
{
  AnnonymousDemo$1(AnnonymousDemo paramAnnonymousDemo) {}

  public void aShow()
  {
    System.out.println("A show()");
  }
}
```

Class B : Inner Class

```
class AnnonymousDemo$2  extends B
{
  AnnonymousDemo$2(AnnonymousDemo paramAnnonymousDemo) {}

  void bShow()
  {
    System.out.println("B show()");
  }
}
```

2.If we want to create the Object for inner class we must use outer class object . because inner classes are generated inside of outer class

```
AnnonymousDemo demo = new AnnonymousDemo();
             demo.a.aShow();
             demo.b.bShow();
```

4. Static Nested Classes (Nested Classes)

If a Static class is created inside Outer class is know as Static Nested class

- **Non Static Data Members/Methods** : it **Cannot** assess directly
- **Static Data Members** : it **Can** access

Example

```
public class StaticNestedDemo {
	int a = 100;
	static int b = 200;
	static class Inner {
		static void get() {
			System.out.println("B " + b);
		// a -Cannot make a static reference to the non-static field a
		}
	}
	public static void main(String[] args) {
		StaticNestedDemo.Inner ob = new StaticNestedDemo.Inner();
		ob.get();
		// ditectly
		StaticNestedDemo.Inner.get();
	}
}
```

```
B 200
B 200
```

VII.Java I/O

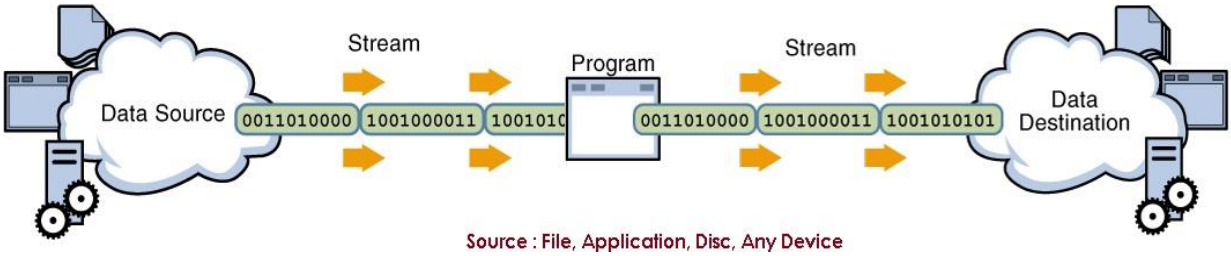
For dealing with input & Output Operations in java we have **java.io.*** package

In java we will write two types of programs

1. volatile programs : whose result are stored in main Memory (RAM),Temporally (Ex. Console Applications)

2. Non-Volatile programs : whose results are saved permanently in secondary memory like Drives,Harddisks, Databases & files.

Stream : flow of data/bites/bytes from source to destination



We have following types of streams to handle IO operations.

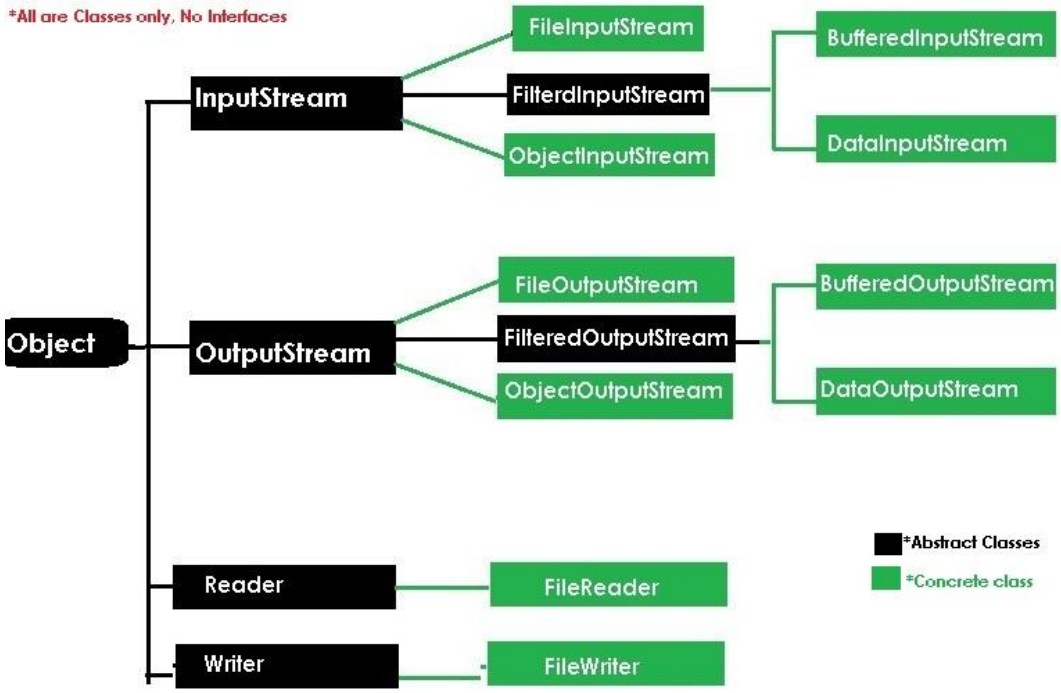
1. **Byte Streams** : perform input and output of 8-bit bytes. (**FileInputStream** & **FileOutputStream**)

2. **Character Streams** : I/O of character data, automatically handling translation to and from the local character set (**FileReader and FileWriter)**

3. Buffered Streams : Above are **unbuffered I/O**. This means each read or write request is handled **directly by the underlying OS.** Buffered input streams read data from a memory area known as a buffer; the native input API is called only when the buffer is empty. Similarly, buffered output streams write data to a buffer, and the native output API is called only when the buffer is full **(BufferedInputStream and BufferedOutputStream)**

4. Data Streams : handle I/O of primitive data type and String values. **(DataInputStream & DataOutputStream.)**

5. Object Streams : handle binary I/O of objects**. (ObjectInputStream and ObjectOutputStream)**



InputStream, OutputStream methods can be used by all their child classes for performing IO operations

1. InputStream : read Data from File/Source

i. *public int read ();*
ii. *public int length (); // total size of the file*
iii. *public int available (); // available number of bytes only*
iv. *public void close ();*

2.OutputStream : Write data to file/Destination

i. *public void write (int);*
ii. *public int length ();*
iii. *public void available ();*
iv. *public void close ();*

In java End of file (EOF) is indicated by -1

7.1 Byte Streams

- Data transfer is **one byte at a time** from source to destination
- used for reading streams of raw bytes such as **image data**
- used to read byte-oriented data for example **to read image, audio, video etc**

1. FileInputStream is meant for reading streams of raw bytes such **as image data**. For reading streams of characters, consider using **FileReader**. Below are the constructor's to use FileInputStream

Constructors	Methods
FileInputStream(File file) **FileInputStream(String FilePath)** **FileInputStream(FileDescriptor fdObj)**	Int read(byte b[]) Int read(byte[] b, int off, int len)

2. FileOutputStream

Constructors	Methods
FileOutputStream(File file) **FileOutputStream(String filepath)** **FileOutputStream(FileDescriptor fdObj)** **FileOutputStream(File file, boolean append)** //true ➔append, false➔ overrite **FileOutputStream(String name, boolean append) name.**	void write(byte b[]) void write(byte[] b, int off, int len)

used for reading/writing data from/to **Binary files like image,videos,xls,docs**.

Example

```
public class ByteStreams {
	public static void main(String[] args) throws IOException {
		String filepath = "E:\\users\\Kaveti_s\\Desktop\\Books\\tmp.txt";
		FileOutputStream outputStream = new FileOutputStream(filepath);
		for (int i = 0; i < 10; i++) {
			outputStream.write(i);
		}
		FileInputStream inputStream = new FileInputStream(filepath);
		int i;
		while ((i = inputStream.read()) != -1) {
			System.out.println("I : " + i);
		}
	}
}
```

```
I : 0,I : 1,I : 2,I : 3,I : 4,I : 5,I : 6,I : 7,I : 8,I : 9
```

If we open **tmp.txt** , the data in the form of bytes. That means we can't read that data. For above example the file data is

7.2 Character Streams

Character stream I/O automatically translates this internal format to and from the local character set.here the data is read by character by character

1. **FileReader** is meant for reading streams of characters
2. **FileWriter** is meant for writing streams of characters

Here Methods & Constructors are Similar to Byte Stream, but **instead of byte they will char data.** used for reading/writing data from/to **Files by character encoding**.

Example

```
public class CharacterStreams {
	public static void main(String[] args) throws IOException {
		String filepath = "E:\\users\\Kaveti_s\\Desktop\\Books\\tmp.txt";
		char[] ch ={ 'a', 'b', 'c', 'd', 'e' };
		FileWriter w = new FileWriter(filepath);
			w.write(ch);
			w.close();

		FileReader r= new FileReader(filepath);
		int i;
		while ((i = r.read()) != -1) {
			System.out.println(i+":"+(char)i);
		}
	}
}
```

```
97:a    98:b    99:c    100:d   101:e
```

Here we can read file data. Data stored in the file is abcde

If we pass int data to FileWriter the program will execute without Compilation Error but it doesn't display any Output / Empty Output

7.3 Buffered Streams

Buffering can speed up IO quite a bit. Rather than read one byte at a time from the network or disk, the BufferedInputStream reads a larger block at a time into an internal buffer.

When you read a byte from the BufferedInputStream you are therefore reading it from its internal buffer. When the buffer is fully read, the BufferedInputStream reads another larger block of data into the buffer. This is typically much faster than reading a single byte at a time from an InputStream, especially for disk access and larger data amounts.

To convert an unbuffered stream into a buffered stream, we need to pass the unbuffered stream object to the constructor for a buffered stream class

Example

inputStream = new BufferedReader(new FileReader("xanadu.txt"));

outputStream = new BufferedWriter(new FileWriter("characteroutput.txt"));

1. BufferedInputStream:

BufferedInputStream class is used for reducing number of physical read operation. When we create an object of BufferedInputStream, we get a temporary peace of memory space **whose default size is 1024 bytes** and it can be increased by multiples of 2.

2.BufferedOutputStream:

BufferedOutputStream class is used for reducing number of physical write operation when we create an object of BufferedOutputStream, we get a temporary peace of memory space whose **default size is 1024** bytes and it can be increased by multiple of 2.

Constructors	Methods
BufferedInputStream(InputStream is) **BufferedInputStream(InputStream is, int bufsize)** **BufferedOutputStream(OutputStream os)** **BufferedOutputStream(OutputStream os, int bufsize)**	Int read(byte b[]) Int read(byte[] b, int off, int len) void write(byte b[]) void write(byte[] b, int off, int len)

used for reading/writing data from/to **Files**.

Example

```
public class BufferedStreams {
	public static void main(String[] args) throws IOException {
		String filepath = "E:\\users\\Kaveti_s\\Desktop\\Books\\sl.txt";
		// 1.Create Stream Object
		FileOutputStream fos = new FileOutputStream(filepath);
		// 2.pass Stream object to BufferStream constructor
		BufferedOutputStream bos = new BufferedOutputStream(fos);
		String s = "SmlCodes.com -Programmimg Simplified";
		byte[] b = s.getBytes();
		bos.write(b);
		bos.flush();

		// 1.Create Stream Object
		FileInputStream fis = new FileInputStream(filepath);
		// 2.pass Stream object to BufferStream constructor
		BufferedInputStream bis = new BufferedInputStream(fis);
		int i;
	    while((i=bis.read())!=-1){
	     System.out.println((char)i);
	    }
	}
}
```

- Byte streams will transfer 1 byte of data at a time
- Character streams will transfer 2 bytes of data at a time
- Buffered Streams will transfer 1024 bytes of data at a time

7.4 Data Streams

Data streams support binary I/O of **primitive data type values** (***boolean, char, byte, short, int, long, float, and double***) and **String** values.All data streams implement either the **DataInput** interface or the **DataOutput** interface

1. **DataInputStream :**Used for read primitive Java data types from input stream.(**readXXX() method)**
2. **DataOutputStram** : Used for write primitive Java data types to Output stream.(**writeXXX() method)** here XXX = primitive data types

Constructors	Methods
DataInputStream (InputStream is)	Int read(byte b[]) Int read(byte[] b, int off, int len) Byte readByte() Int readInt() Char readchar()
DataOutputStream (OutputStream os)	void write(byte b[]) void write(byte[] b, int off, int len) void writeByte(byte b) void writeInt(int i)

Example

```
public class DataStream {
	public static void main(String[] args) throws Exception {
		DataOutputStream dos = new DataOutputStream(new
FileOutputStream("sml.bin"));
		dos.writeInt(10);
		dos.writeUTF("Satya");

		DataInputStream dis = new DataInputStream(new
FileInputStream("sml.bin"));
		System.out.println("Int : " + dis.readInt());
		System.out.println("String : " + dis.readUTF());
	}
}
```

```
Int : 10
String : Satya
```

7.5 Object Streams

Just as data streams support I/O of primitive data types, **object streams support I/O of objects.** Here we have to know about **Serialization.**

ObjectOutputStream(OutputStream out)	ObjectInputStream(InputStream in)
void writeObject(Object obj)	Object readObject()

7.5.1 Serialization

Serialization is the **process of saving the state of the object permanently** in the form of a file/byte stream. To develop serialization program follow below steps

Steps to implement Serialization

1. **Choose** the appropriate **class** name whose object is participating in serialization.

2. This class must **implement java.io.Serializable** (this interface does **not contain any abstract methods and** such type of interface is known as **marker or tagged interface**)

3. Choose **data members** , writer **setters** & **getters**

4. Choose **Serializable subclass**

5. **Choose the file** name and **open it into write mode** with the help of **FileOutputStream** class

6. Pass OutputStream object to **ObjectOutputStream(out)** constructor to write object data at a time

7.use oos.writeObject(student) method to write Student Object data

Example

```
class Student implements Serializable {
	// Exception in thread "main" java.io.NotSerializableException:
io.Student
	private int sno;
	private String name;
	private String addr;
	public int getSno() {
		return sno;
	}
	public void setSno(int sno) {
		this.sno = sno;
	}
	public String getName() {
		return name;
	}
	//same way setName,SetAddr methods…
}
public class Serialization {
public static void main(String[] args) throws  Exception {
	Student student = new Student();
	student.setSno(101);
	student.setName("Satya Kaveti");
	student.setAddr("VIJAYAWADA");

	FileOutputStream fos = new FileOutputStream("student.txt");
	ObjectOutputStream oos = new ObjectOutputStream(fos);
	oos.writeObject(student);
}
}
```

```
¬í sr
io.StudentÓÞ®(¦°¦ ————————————————————I ————————————————————snoL
addrt Ljava/lang/String;L
nameq ~ xp   et
VIJAYAWADAt  //data saved in student.txt
```

7.5.2 Deserialization

De-serialization is a **process of retrieve the data from the file in the form of object**.

Steps

1. Choose the file name and open it into read mode with the help of FileInputStream class

2. Pass InputStream object to ObjectInputStream(in) constructor to read object data at a time

3.use ois.readObject() method to get Student Object

Example

```
public class Deserialization {
	public static void main(String[] args) throws Exception{
		FileInputStream fis = new FileInputStream("student.txt");
		ObjectInputStream ois = new ObjectInputStream(fis);
		Student st = (Student)ois.readObject();
		System.out.println(st.getSno());
		System.out.println(st.getName());
		System.out.println(st.getAddr());
	}
}
```

```
101
Satya Kaveti
VIJAYAWADA
```

If we use above process to implement serialization, all the data members will participate in Sterilization process. If you want to use selected data members for serialization use **Transient** keyword

7.5.3 Transient Keyword

In order to avoid the variable from the serialization process, make that variable declaration as transient i.e., **transient variables never participate in serialization process**.

Example

```
class Student implements Serializable {
	private transient int sno;
	private transient String name;
	private String addr;
```

```
}

public class TransientExample {
	public static void main(String[] args) throws  Exception {
		Student student = new Student();
		student.setSno(101);
		student.setName("Satya Kaveti");
		student.setAddr("VIJAYAWADA");

		FileOutputStream fos = new FileOutputStream("student.txt");
		ObjectOutputStream oos = new ObjectOutputStream(fos);
		oos.writeObject(student);

		FileInputStream fis = new FileInputStream("student.txt");
		ObjectInputStream ois = new ObjectInputStream(fis);
		Student st = (Student)ois.readObject();
		System.out.println(st.getSno());
		System.out.println(st.getName());
		System.out.println(st.getAddr());
	}
}
```

```
0
null
VIJAYAWADA
```

Printing of sno,name returns 0,null because values of sno,name was not serialized.

7.6 Summarize Java I/O Streams

Streams

- byte oriented stream (8 bit)
- good for binary data such as a Java .class file
- good for "machine-oriented" data

Readers/Writers

1. char (utf-16) oriented stream (16 bit)
2. good for text such as a Java source
3. good for "human-oriented" data

Buffered : always useful unless proven otherwise

Type	Byte Based		Character Based	
Basic	InputStream	OutputStream	Reader InputStreamReader	Writer OutputStreamWriter
1.Arrays	ByteArrayInputStream	ByteArrayOutputStream	CharArrayReader	CharArrayWriter
2.Files	FileInputStream RandomAccessFile	FileOutputStream RandomAccessFile	FileReader	FileWriter

3.Pipes	PipedInputStream	PipedOutputStream	PipedReader	PipedWriter
4.Buffering	BufferedInputStream	BufferedOutputStream	BufferedReader	BufferedWriter
5.Filtering	FilterInputStream	FilterOutputStream	FilterReader	FilterWriter
6.Parsing	PushbackInputStream StreamTokenizer		PushbackReader LineNumberReader	
7.Strings			StringReader	StringWriter
8.Data	DataInputStream	DataOutputStream		
9.Formatted		PrintStream		PrintWriter
10.Objects	ObjectInputStream	ObjectOutputStream		
11.Utilities	SequenceInputStream			

- Byte Streams → xxxInputStreams, xxxOutputStreams
- Char Streams → xxxReader , xxxWriter

1.Arrays: These are used to read/write **same data** to/from **multiple files at same time**

- **ByteArrayInputStream**
- **ByteArrayOutputStream**

- **CharArrayReader**
- **CharArrayWriter**

2.Files

These are used to read/write data to/from one file at a time. **See above examples for more**

- **FileInputStream**
- **FileOutputStream**
- **RandomAccessFile**
- **RandomAccessFile**

- **FileReader**
- **FileWriter**

3.Pipes

- **PipedInputStream**
- **PipedOutputStream**

- **PipedReader**
- **PipedWriter**

4.Buffering

- **BufferedInputStream**
- **BufferedOutputStream**

- **BufferedReader**
- **BufferedWriter**

5.Filtering

The FilterInputStream/FilterOutputStream is a base class for **implementing your own filtering input/output streams**. They does not have any special behavior. Frankly there is no sensible purpose for this class

- **FilterInputStream**
- **FilterOutputStream**

- **FilterReader**
- **FilterWriter**

6. Parsing

PushbackInputStream/PushbackReader

Sometimes you need to read ahead a few bytes to see what is coming, before you can determine how to interpret the current byte.

The PushbackInputStream allows you to do that. it allows you to push the read bytes back into the stream. These bytes will then be read again the next time you call read().

```
PushbackInputStream input = new PushbackInputStream(
                                new FileInputStream("c:\\data\\input.txt"));

int data = input.read();

input.unread(data);
```

StreamTokenizer : StreamTokenizer class (java.io.StreamTokenizer) can **tokenize the characters read from a Reader into tokens**. For instance, in the string "Mary had a little lamb" each word is a separate token

```
StreamTokenizer streamTokenizer = new StreamTokenizer(
        new StringReader("Mary had 1 little lamb..."));

while(streamTokenizer.nextToken() != StreamTokenizer.TT_EOF){

    if(streamTokenizer.ttype == StreamTokenizer.TT_WORD) {
        System.out.println(streamTokenizer.sval);
    } else if(streamTokenizer.ttype == StreamTokenizer.TT_NUMBER) {
        System.out.println(streamTokenizer.nval);
    } else if(streamTokenizer.ttype == StreamTokenizer.TT_EOL) {
        System.out.println();
    }
}
streamTokenizer.close();
```

7. Strings

- **StringReader :** takes input string and changes it into character stream
- **StringWriter :** uses BufferedReader to read file data & written in character array as buffer

8.Data

read Java primitives (int, float, long etc.) from an InputStream instread of bytes

- **DataInputStream**
- **DataOutputStream**

9.Formatted

PrintStream : PrintStream class can **format primitive types like int, long etc**. formatted as text, rather than as their byte values.it is comes under **ByteStream**

```
PrintStream printStream = new PrintStream(outputStream);

printStream.print(true);
printStream.print((int) 123);
printStream.print((float) 123.456);

printStream.close();
```

System.out and System.err are PrintStreams

PrintWriter :this class (java.io.PrintWriter) enables you to write formatted data to an underlying Writer. For instance, writing int, long and other primitive data formatted as text, rather than as their byte values.it is comes under **Character Stream**

```
FileWriter  writer      = new FileWriter("d:\\data\\report.txt");
PrintWriter printWriter = new PrintWriter(writer);

printWriter.print(true);
printWriter.print((int) 123);
printWriter.print((float) 123.456);

printWriter.printf(Locale.UK, "Text + data: %1$", 123);

printWriter.close();
```

10.Objects :read Java objects from an InputStream instead of just raw bytes

- **ObjectInputStream**
- **ObjectOutputStream**

11.Utilities : SequenceInputStream combines two or more other InputStream's into one. First the SequenceInputStream **will read all bytes from the first InputStream**, **then all bytes from the second InputStream**. That is the reason it is called a SequenceInputStream, since the InputStream instances are read in sequence

```
InputStream input1 = new FileInputStream("c:\\data\\file1.txt");
InputStream input2 = new FileInputStream("c:\\data\\file2.txt");

SequenceInputStream si = new SequenceInputStream(input1, input2);

int data = si.read();
while(data != -1){
    System.out.println(data);
    data = si.read();
}
```

VIII. Java Threads

8.1 Introduction to Multi-threading

If a program contains **multiple flow of controls for achieving concurrent execution** then that program is known as **multi-threaded** program

The languages like **C, C++ comes under single threaded modeling languages**, since there exist single flow of controls whereas the languages **like JAVA, DOT NET are treated as multi-threaded modeling languages**, since there is a possibility of creating multiple flow of controls

When we write any JAVA program there exist two threads they are

1. fore ground thread
2. Back ground thread.

1. Fore ground threads are those which are **executing user defined sub-programs**. There is a possibility of creating **'n' number of fore ground threads**

2. Back ground threads are those which are **monitoring the status of fore ground thread.** And always there exist **single back ground thread.**

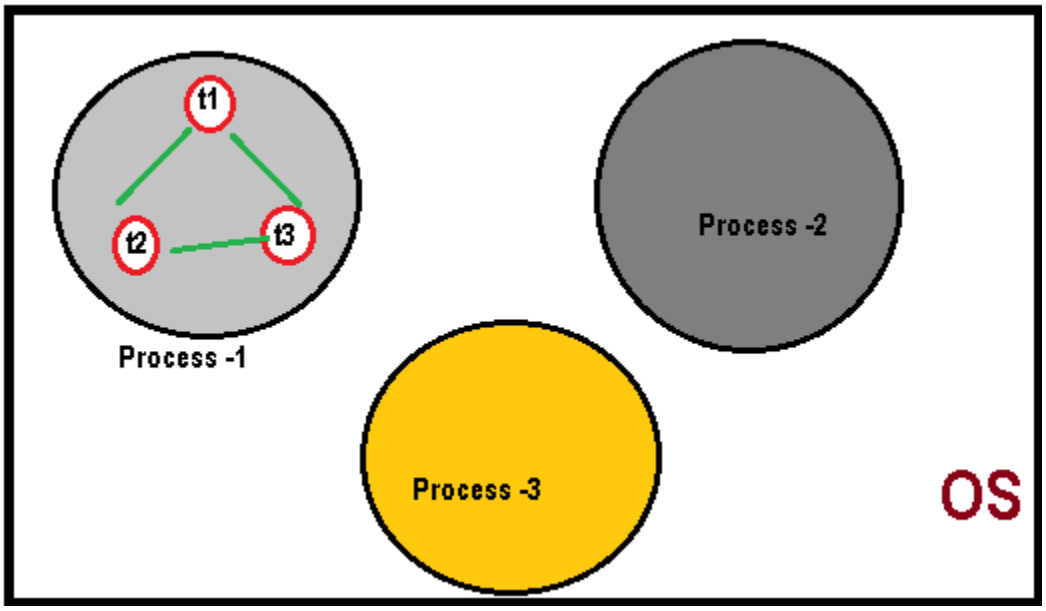
In information technology we can develop two types of applications. They **are process based applications** and **thread based applications.**

Context switch is the concept of operating system and it says switching the control from one address page to another address page

Process Based Applications	Thread Based Applications
1. Exist **single flow of control**.	1. Exist **Multiple flow of controls**.
2. All **C, C++** applications comes under it.	2. All **JAVA, DOT NET** applications comes under it.
3. **Context switch is more**.	3. **Context switch is very less**.
4. Each process have **its own address in memory** i.e. each process allocates separate memory area.	4. Threads share the **same address space**
5. These are treated as **heavy weight** components.	5. These are treated as **light weight** components.
6. In this we can achieve only sequential execution and they **are not recommending for developing internet applications**.	6. In thread based applications we can achieve both sequential and concurrent execution and they are always **recommended for developing internet applications**.

8.2 What is Thread

A thread is a lightweight sub process, a smallest unit of processing. It is a separate path of execution. Threads are independent, if there occurs exception in one thread, it doesn't affect other threads. It shares a common memory area.

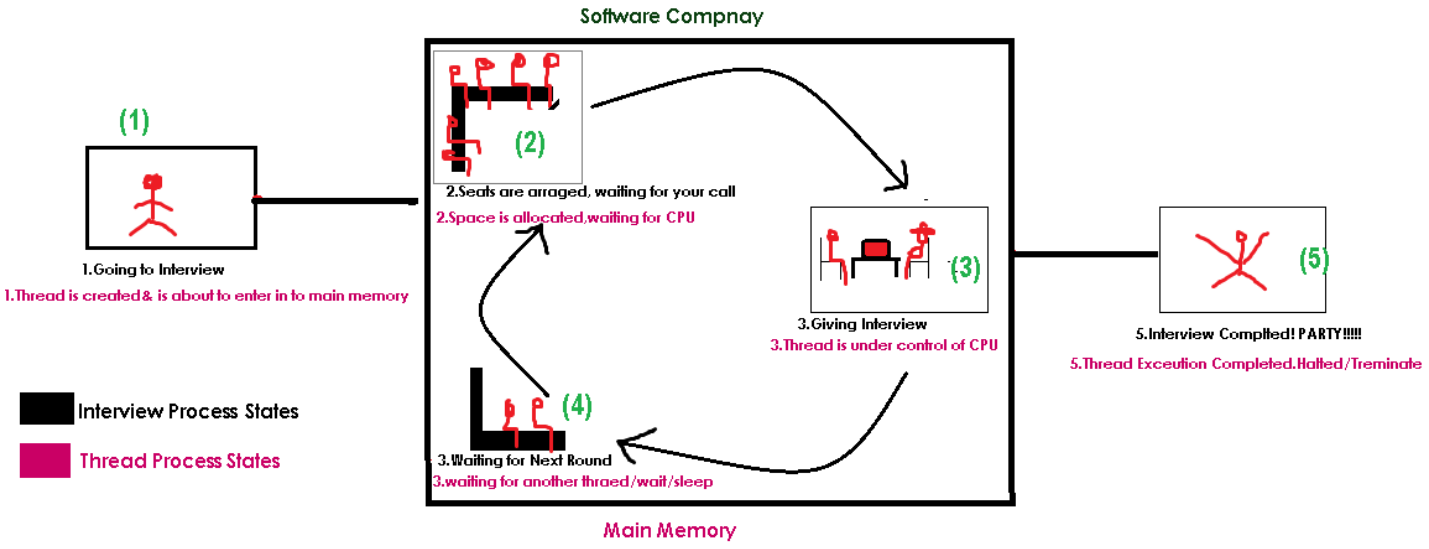


As shown in the above figure, thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS and one process can have multiple threads

At a time one thread is executed only.

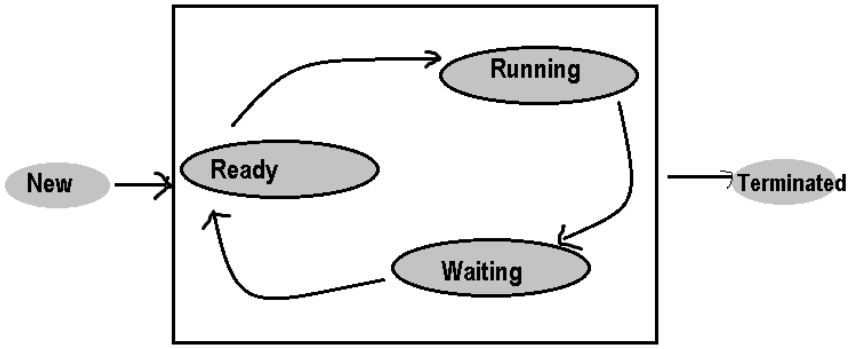
8.3 Thread Life Cycles (Thread States)

See below picture which compares Interview process & Thread Execution process.



Interview Process	Thread execution process
1.going to interview	1.Thread is created & is about to enter into main memory
2.Seats are arranged, waiting for your call	2.Space is allocated, waiting for CPU
3.giving interview	3.Tharead is under control of CPU
4.wating for next round	4.waiting for another thread/wait/sleep
5.interview completed	5.interview completed

Based on the process of execution of thread, people said there are 5 states of a thread



1. New: Thread is created and about to enter into main memory. **i.e.**, New Thread Object is created but **before the invocation of start() method.**

2. Ready/Runnable: thread memory space allocated and it is waiting for CPU for executing. i.e., **after invocation of start() method**, but the thread scheduler has not selected

3. Running: thread is under the control of CPU. i.e., **thread scheduler has selected (run() executing)**.

4. Waiting: This is the state when the thread is still alive, but is currently not eligible to run.Thread is waiting because of the following factors:

- For the repeating CPU burst time of the thread
- Make the thread to sleep for sleep for some specified amount of time.
- Make the thread to suspend.
- Make the thread to wait for a period of time.
- Make the thread to wait without specifying waiting time.

5. Terminated: thread has completed its total execution. i.e., Exit form run () method

We have two ways of creating Thread,

- **by extending java.lang.Thread class**
- **By implementing java.lang.Runnable interface.**

In multi-threading we get only one exception known as java.lang.InterruptedException.

8.4 java.lang.Thread class

Creating a flow of control in JAVA is nothing but creating an object of java.lang.Thread class.

An object of Thread class can be created in three ways. They are:

- Directly **Thread t=new Thread ();**
- Using factory method **Thread t1=Thread.currentThread ();**
- Using sub-class that **extends Thread class**

Public class **Thread** *extends* **Object** *implements* **Runnable**

Constructors	Usage
Thread()	Creates new Thread, whose default thread name is Thread-0
Thread(String name)	Creates new Thread, with user defined thread name
Thread(Runnable r)	Used for converting Runnable Object to Thread Object for accessing start() method with default thread name
Thread(Runnable r, String name)	Used for converting Runnable Object to Thread Object with user-defined thread name

Instance Thread State Methods

1. Void start () : Used for making the **Thread to start to execute the thread logic**. The method start is **internally calling** the method **run ().**

2. Void run () : The thread that logic must be defined only in run () method. When the thread is started, the JVM looks for the appropriate run () method for executing the logic of the thread. **Thread class is a concrete class and it contains all defined methods and all these methods are being to final except run () method. run () method is by default contains a definition with null body**. Since we are providing the logic for the thread in run () method. Hence it must be overridden by extending Thread class into our own class.

3. void suspend() - This method is used for **suspending the thread from current execution** of thread. When the thread is suspended**, it sends to waiting state by keeping the temporary results in process control block (PCB)** or job control block (JCB). (deprecat**ed)**

4. void resume() -resumes suspend() Thread. Resumed to start executing from where it left out previously by **retrieving the previous result from PCB** (deprecated)

5. void interrupt() -Interrupts this thread

6. void join() -Waits for this thread to die.

7. void join(long mil) -Waits at most milliseconds for this **thread to die**

8. void stop() -is used to stop the thread(deprecated).

Static Methods

Static void sleep(long millis)	- sleeps/temporary block the thread for specified amount of time
Static void yield ()	- pause current thread and allow other threads to execute
Static Thread currentThread()	-Get currently running thread Object. mainly used in **run()**
Static int activeCount()	-Counts the no.of active threads in current thread group& subgroups.

Object class methods used in Multithreading

void wait()	- waits the current thread until another thread invokes the notify()
void wait (long ms)	**-** waits the current thread until another thread invokes the notify()/specified amount of time
void notify()	-Wakes up a single thread that is waiting on this object's monitor.
Void notifyAll()	-Wakes up all threads that are waiting on this object's monitor.

Other useful instance methods

- **void setName(String name)** -set thread's name
- **String getName()** -Returns this thread's name.
- **long getId()** -Returns the identifier of this Thread.
- **void setDaemon(boolean on)** -Marks this thread as either a daemon thread
- **int getPriority()** -Returns this thread's priority.
- **Boolean isAlive()** -Tests if this thread is alive
- **Boolean isDaemon()** -Tests if this thread is a daemon thread.
- **Thread.State getState()** -Returns Current Thread State
- **ThreadGroup getThreadGroup()** -Returns the thread group to which this thread belongs.

- ☑ **public static final int MIN_PRIORITY (1);**
- ☑ **public static final int NORM_PRIORITY (5);**
- ☑ **public static final int MAX_PRIORITY (10);**

The above data members are used for setting the priority to threads are created. By default, whenever a thread is created whose default priority **NORM_PRIORITY**

Thread class contains all defined &final methods except run () method.run() is null body method

8.5 java.lang.Runnable Interface

Runnable Interface has only one abstract method run(). Thread class implemented Runnable interface as run() method as null body method

Public void run(): is used to perform action for a thread

As said we can use either of Thread class /Runnable interface to implement threads.

By Extending Thread Class

```
public class ThreadDemo extends Thread {
	@Override
	public void run() {
		System.out.println("Iam Running");
	}
	public static void main(String[] args) {
		ThreadDemo ob = new ThreadDemo();
		ob.start();
	}
}
```

By Implementing Runnable Interface

```
public class RunnableDemo implements Runnable {
	@Override
	public void run() {
		System.out.println("Iam Running");
	}
	public static void main(String[] args) {
		RunnableDemo r = new RunnableDemo();
		Thread ob = new Thread(r);
		ob.start();
	}
}
```

By Extending Thread Class	By Implementing Runnable Interface

1.write a class **extending Thread class**
 	
2.write **execution logic in run()** method
 	
3.Create Object of thread
`ThreadDemo ob = new ThreadDemo();`
 	
4.**call start()** method, it internally calls run() method
`ob.start();`
	1.write a class **implements Runnable Interface**
	
	2.write **execution logic in run()** method
	
	3.Create Object of implemented thread class & create Thread Object by passing it
	`RunnableDemo r = new RunnableDemo();`
	`Thread ob = new Thread(r);`
	
	4.call start() method, it internally calls run() method
	`ob.start();`

Example 1:

```
public class ThreadExample extends Thread {
	@Override
	public void run() {
		System.out.println("----- \n Im Run() Running....\n -----");
	}
	public static void main(String[] args) throws InterruptedException {
		ThreadExample th = new ThreadExample();
		System.out.println(th.getState().name());
		th.start();
		System.out.println(th.getState().name());
		System.out.println("getId : " + th.getId());
		System.out.println("getName : " + th.getName());
		System.out.println("getPriority : " + th.getPriority());
		System.out.println("isAlive : " + th.isAlive());
		System.out.println("isDaemon : " + th.isDaemon());
		System.out.println("getThreadGroup : " + th.getThreadGroup().getName());
		th.setName("SmlCodes-Thread");
		System.out.println("getName : " + th.getName());
		Thread.sleep(2500);//
		System.out.println(th.getState().name());
	}
}
```

```
NEW
RUNNABLE
getId : 9
getName : Thread-0
getPriority : 5
isAlive : true
isDaemon : false
getThreadGroup : main
getName : SmlCodes-Thread
-----
 Im Run() Running....
 -----
TERMINATED
```

Example 2: Thread program which displays 1 to 10 numbers after each and every 1 second

```
public class SleepDemo extends Thread {
	public void run() {
		for (int i = 1; i <= 10; i++) {
			System.out.println(i);
			try {
				Thread.sleep(1000);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
	}
	public static void main(String[] args) {
		SleepDemo ob1 = new SleepDemo();
		SleepDemo ob2 = new SleepDemo();
		ob1.start();
		ob2.start(); }
}
```

Output

```
1
1
2
2
3
3
4
4
5
5
6
6
7
7
8
8
9
9
10
10
```

Example 3: What happens if we starts same Thread(ob) Twice?

```
public class ThreadDemo extends Thread {
	@Override
	public void run() {
		System.out.println("Iam Running");
	}
	public static void main(String[] args) {
		ThreadDemo ob = new ThreadDemo();
		ob.start();
		ob.start();
	}
}
```

```
Exception in thread "main" java.lang.IllegalThreadStateException
	at java.lang.Thread.start(Thread.java:705)
	at threads.ThreadDemo.main(ThreadDemo.java:11)
Iam Running
```

What happens if we call run() method instead of start()

If we start **run()** method directly JVM treats it as a normal method & it does have characteristics like concurrent execution. In Example 2 if you see both threads are executing parallel. Here below example we are calling **run()** method directly. See the output

```
public class SleepDemo extends Thread {
	public void run() {
		for (int i = 1; i <= 10; i++) {
			System.out.println(i);
			try {
				Thread.sleep(1000);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
	}
	public static void main(String[] args) {
		SleepDemo ob1 = new SleepDemo();
		SleepDemo ob2 = new SleepDemo();
		ob1.run();
		ob2.run();
	}
}
```

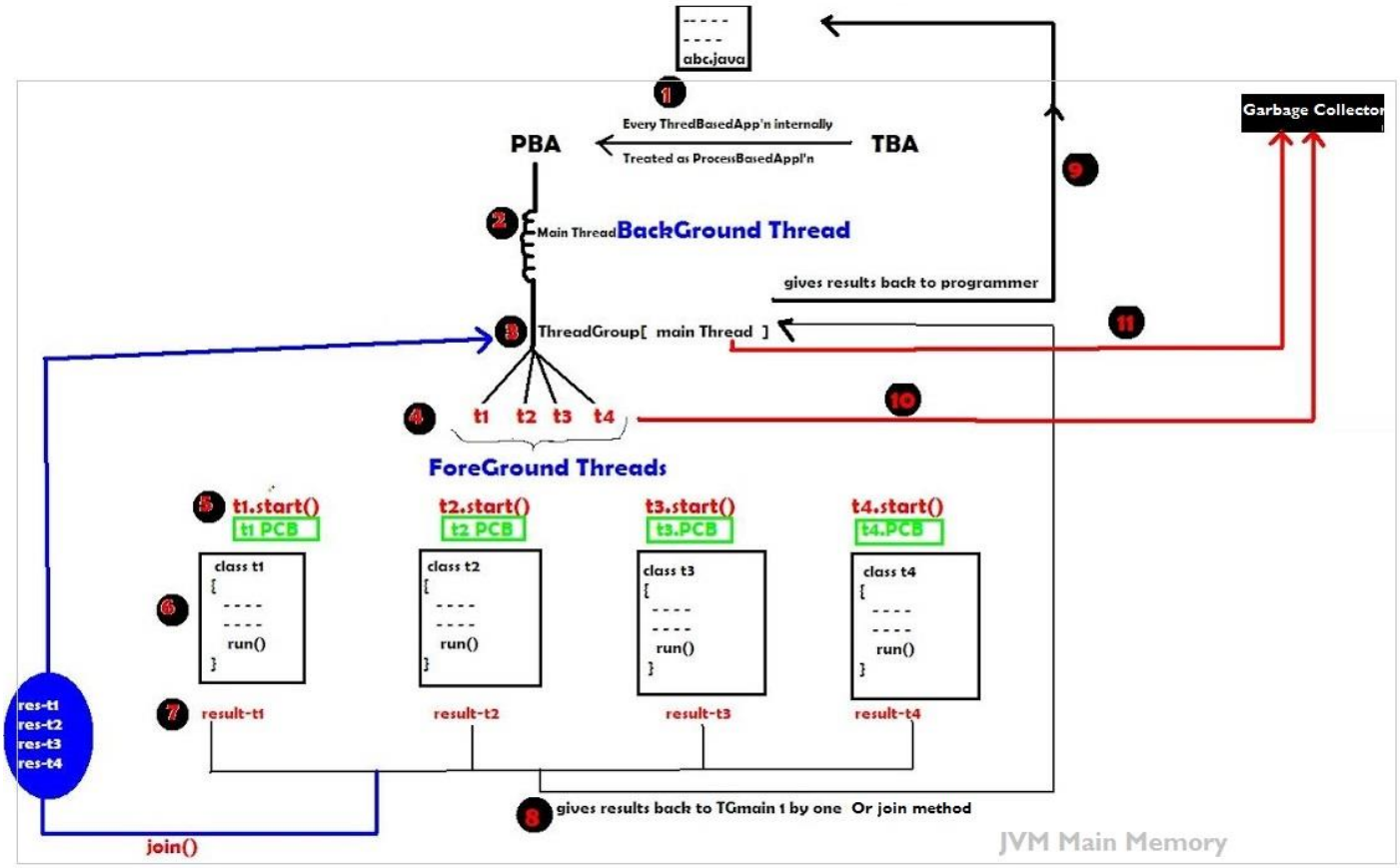
Output

```
1
2
3
4
5
6
7
8
9
10
1
2
3
4
5
6
7
8
9
10
```

Remember: only one exception InterruptedException

8.6 Joining a Thread (join () method)

Thread internal working



1. Thread Program goes to under Execution.

2. Internally every ThredBasesApplc'n is Treated as ProcessBasedAppl'n

3. JVM creates one "ThreadGroupName[main]", which is always resides in main memory

4. TGName Creates no. of ForeGroundThreads (t1,t2,t3,t4)

5. start excecuting of ForeGround threds by calling 'Start()'.
then JVM allocates "ProcessControlBlock" to each FGThred to maintain the Status of the Thread

6. 'Start()' method internally calls 'run()' automatically

7. After excuting run(), results are come out

8. Those RESULTS are tranfer to TGName 1 by 1 (or) All at Once by using "join()"

9. TGName gives results to PROGRAMMER

10. TGName Collects the FGThreds and HandOver to "Garbage Collector"

11. JVM collects TGName, and HandOver to "Garbage Collector"

In above after completion of thread execution result are given to one by one / All at once using **join() method to Thread Group name**

This method is used for making the **fore ground threads to join together**, so that JVM can call the garbage collector only one time for collecting all of them instead of collecting individually.

We have two join() methods

1. **public void join()throws InterruptedException :*Waits for this thread to die***.
   It will wait until Thread logic completion & after that it will joins the Thread & Gives to Garbage collector

Join() Example

```
package threads;

public class JoinExample extends Thread {
	@Override
	public void run() {
		for (int i = 1; i <= 10; i++) {
			try {
				Thread.sleep(500);
			} catch (InterruptedException e) {
				// TODO Auto-generated catch
block
				e.printStackTrace();
			}
			System.out.println(i);
		}
	}

	public static void main(String[] args) {
		JoinExample t1 = new JoinExample();
		JoinExample t2 = new JoinExample();
		JoinExample t3 = new JoinExample();

		t1.start();
		try {
			t1.join();
		} catch (InterruptedException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
		t2.start();
		t3.start();
	}
}
```

Output

```
1
2
3
4
5
6
7
8
9
10
1
1
2
2
3
3
4
4
5
5
6
6
7
7
8
8
9
9
10
10
```

In above t2, t3 threads waits for t1 thread to die. After completion of t1 thread execution t12, t3 are started.

2. **public void join(long milliseconds)throws InterruptedException :** ***Waits at most milliseconds for this thread to die.*** That means it waits for thread to die in give milliseconds. If it won't die in give time treated as normal thread & executes parallel with other threads if any.

```
package threads;

public class JoinExample extends Thread {
	@Override
	public void run() {
		for (int i = 1; i <= 10; i++) {
			try {
				Thread.sleep(500);
			} catch (InterruptedException e) {
			// TODO Auto-generated catch block
				e.printStackTrace();
			}
			System.out.println(i);
		}
	}

	public static void main(String[] args) {
		JoinExample t1 = new JoinExample();
		JoinExample t2 = new JoinExample();
		JoinExample t3 = new JoinExample();

		t1.start();
		try {
			t1.join(2500);
		} catch (InterruptedException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
		t2.start();
		t3.start();
	}
}
```

```
1
2
3
4
5
1
1
6
7
2
2
3
3
8
9
4
4
10
5
5
6
6
7
7
8
8
9
9
10
10
```

If you see in above example t2, t3 threads are waiting for t1 thread to die in 2500 milliseconds. But in given time t1 did not die. So, t2, t3 threads start their execution parallel with t1 thread

8.7 Thread Priority

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, **thread scheduler schedules the threads according to their priority** (known as preemptive scheduling). **But it is not guaranteed because it depends on JVM specification** that which scheduling it chooses.

1. **public static int MIN_PRIORITY //1**
2. **public static int NORM_PRIORITY //5 (default)**
3. **public static int MAX_PRIORITY //10**

Example

```
public class ThreadPriority extends Thread{
	@Override
	public void run() {
	Thread th= Thread.currentThread();
	System.out.println("Name :"+th.getName() +"\t Priortity:"+th.getPriority());
	}
	public static void main(String[] args) {
		ThreadPriority t1 = new ThreadPriority();
		ThreadPriority t2 = new ThreadPriority();
		ThreadPriority t3 = new ThreadPriority();

		t1.setPriority(MIN_PRIORITY);
		t2.setPriority(NORM_PRIORITY);
		t3.setPriority(MAX_PRIORITY);

		t1.start();
		t2.start();
		t3.start();
	}
}
```

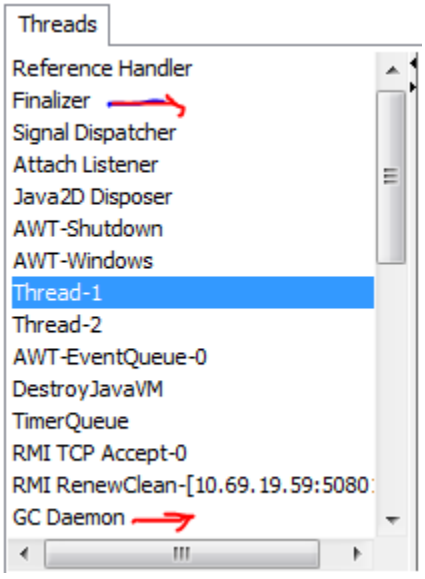
```
Name :Thread-2	Priortity:10
Name :Thread-1	Priortity:5
Name :Thread-0	Priortity:1
```

Even though t1 starts first, it has ***MIN_PRIORITY*** so, it executes last that to depends on JVM Specification

8.8 Daemon Thread

Daemon thread is a thread that provides services to the user thread. There are many java daemon threads running automatically **e.g. gc, finalizer etc.** JVM terminates these thread automatically**.**

We can see all daemon threads using **JConsole** (C:\Program Files\Java\jdk1.8.0_45\bin\jconsole.exe)



- It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.
- Its life depends on user threads.
- It is a low priority thread.

We have two methods to dealing the Demon Threads

1.public void setDaemon(boolean status) : set'scurrent thread as daemon thread or user thread.

2. public boolean isDaemon() : is used to check that current is daemon.

Example

```
package threads;

public class DemonEx extends Thread {
	@Override
	public void run() {
		Thread th = Thread.currentThread();
		if (th.isDaemon()) {
			System.out.println("DEMON THREAD " + th.getName());
		} else {
			System.out.println("NORMAL THREAD " + th.getName());
		}
	}

	public static void main(String[] args) {
		DemonEx t1 = new DemonEx();
		DemonEx t2 = new DemonEx();

		t1.setDaemon(true);
		t1.start();
		t2.start();
	}
}
```

```
DEMON THREAD Thread-0
NORMAL THREAD Thread-1
```

If you want to make a user thread as Daemon, it must not be started otherwise it will throw IllegalThreadStateException

```
public static void main(String[] args) {
		DemonEx t1 = new DemonEx();
		DemonEx t2 = new DemonEx();
		t1.start();
		t1.setDaemon(true);
		t2.start();
	}
```

```
Exception in thread "main" java.lang.IllegalThreadStateException
	at java.lang.Thread.setDaemon(Thread.java:1352)
	at threads.DemonEx.main(DemonEx.java:18)
```

8.9 Thread Group

Thread Group is a process of grouping multiple threads in to a single object. We can suspend, interrupt & resume in a single method call.

Constrctors	
ThreadGroup(String name)	creates a thread group with given name.
ThreadGroup(ThreadGroup parent, Str name)	creates a thread group with given parent group & name.

Methods	
int activeCount()	returns no. of threads running in current group.
int activeGroupCount()	returns a no. of active group in this thread group.
void destroy()	destroys this thread group and all its sub groups.
String getName()	returns the name of this group.
ThreadGroup getParent()	returns the parent of this group.
void interrupt()	interrupts all threads of this group.
void list()	prints information of this group to standard console.

Example

```
class ThreadEx extends Thread {
	@Override
	public void run() {
	Thread th = Thread.currentThread();
System.out.println("Thread Name:"+th.getName()+"Name:"+ th.getThreadGroup());
	}
}
public class ThreadGroupDemo {
	public static void main(String[] args) throws InterruptedException {
		ThreadGroup tg = new ThreadGroup("SmlCodes Group");
		ThreadEx thread = new ThreadEx();
		// adding thraeds to Theard Group
		Thread t1 = new Thread(tg, thread, "Thread-1");
		t1.start();
		Thread t2 = new Thread(tg, thread, "Thread-2");
		t2.start();
		Thread t3 = new Thread(tg, thread, "Thread-3");
		t3.start();
		tg.list();
	}
}
```

Output

```
java.lang.ThreadGroup[name=SmlCodes Group,maxpri=10]
Thread Name: Thread-3    Thread Group Name: java.lang.ThreadGroup[name=SmlCodes Group,maxpri=10]
Thread Name: Thread-1    Thread Group Name: java.lang.ThreadGroup[name=SmlCodes Group,maxpri=10]
Thread Name: Thread-2    Thread Group Name: java.lang.ThreadGroup[name=SmlCodes Group,maxpri=10]
    Thread[Thread-1,5,SmlCodes Group]
    Thread[Thread-2,5,SmlCodes Group]
    Thread[Thread-3,5,SmlCodes Group]
```

8.10 Synchronization

Synchronization is a process of allowing only one thread at a time

Lock: Synchronization is built around an internal entity known as the lock or monitor. Every object has a lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

Problem without Synchronization

```
class Counter implements Runnable {
	private int count;
	@Override
	public void run() {
		for (int i = 1; i <= 5; i++) {
			waitCounter (i);
			count++;
		}
	}

	public int getCount() {
		return this.count;
	}
	public void waitCounter(int i) {
	try {
			Thread.sleep(1000);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
	}
}
public class ThreadSafety {
	public static void main(String[] args) throws InterruptedException {
		Counter c = new Counter();
		Thread t1 = new Thread(c);
		t1.start();
		Thread t2 = new Thread(c);
		t2.start();
		// wait for threads to finish processing
		t1.join();
		t2.join();
		System.out.println("Processing count=" + c.getCount());
	}
}
```

```
Processing count=8
```

In above example for a single thread **Counter** we created two child threads t1,t2 and **count** is a variable common for those two threads. **After completion of thread execution the counter must be 10**. But **here it is displaying output as 8** because two threads are executing parallel on same method **waitCounter(),** the result may is overlapped two threads are executing same method at same time.

To resolve these types of problems we use synchronization. We can implement synchronization is **3 ways**

1. Synchronized Instance Methods

2. Synchronized Static Methods.

3. Synchronized Blocks

1. Synchronized Instance methods:

If the ordinary instance method is made it as **synchronized** then the object of the corresponding class will be locked

```
	synchronized void waitCounter(int i) {
		try {
			Thread.sleep(1000);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
	}
```

2. Synchronized static method

If an ordinary static method is made it as synchronized then the corresponding class will be locked.

```
synchronized static void waitCounter(int i) {
		try {
			Thread.sleep(1000);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
	}
```

3. Synchronized block:

When we inherit non-synchronized methods from either base class or interface into the derived class, we cannot make the inherited method as synchronized. Hence, we must use synchronized blocks

```
	public void waitCounter(int i) {
		synchronized (this) {
			try {
				Thread.sleep(1000);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
	}
```

If you use any of above methods the output should be `Processing count=10`

8.11 Inter Thread Communication

If a Thread is synchronized only one thread should be access at a time. To access multiple threads on synchronized resource their should be some communication between them. Inter-thread communication or Co-operation is all about allowing synchronized threads to communicate with each other.

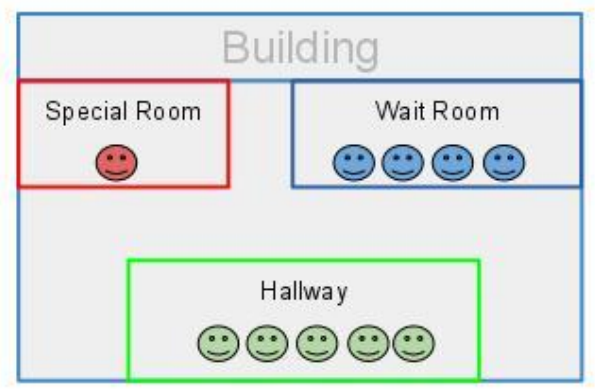
Inter-thread communication is a mechanism in which **a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed**. It is implemented by following **methods of Object class**:

- **void wait()** - waits the current thread until another thread invokes the notify()
- **void wait (long ms)-** waits the current thread until another thread invokes the notify()/specified amount of time

- **void notify()** -Wakes up a single thread that is waiting on this object's monitor.
- **Void notifyAll()** -Wakes up all threads that are waiting on this object's monitor.

8.11.1. What is a Monitor?

In general terms monitor can be considered as a building which contains a special room. The special room can be occupied by only one customer(thread) at a time. The room usually contains some data and code.



A monitor is mechanism to control concurrent access to an object.

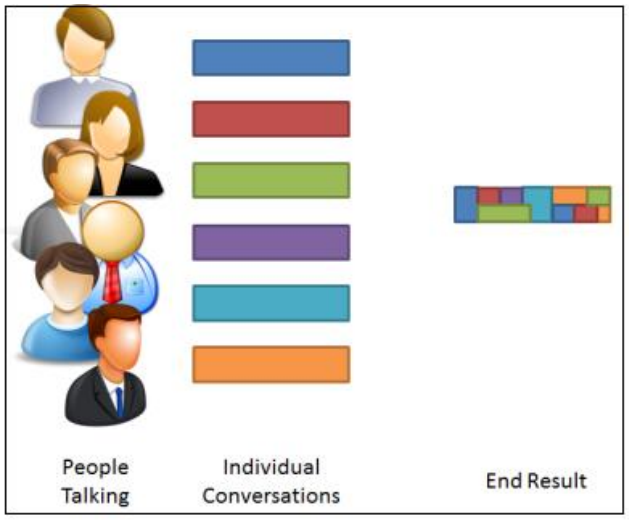
Thread 1:

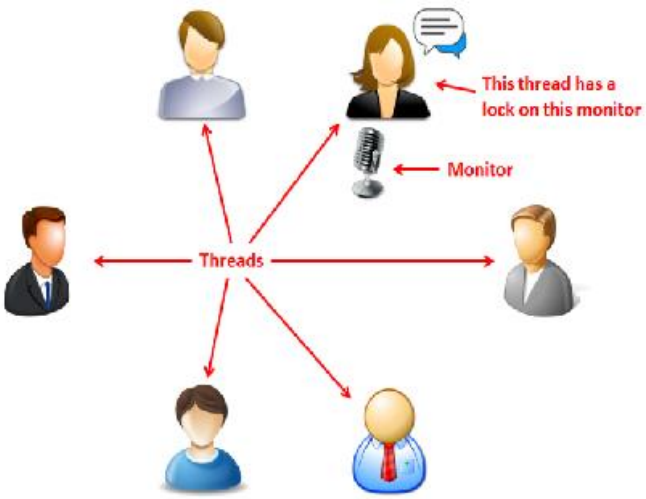
```
public void a()
{
    synchronized(someObject) {
        // do something (1)
    }
}
```

Thread 2:

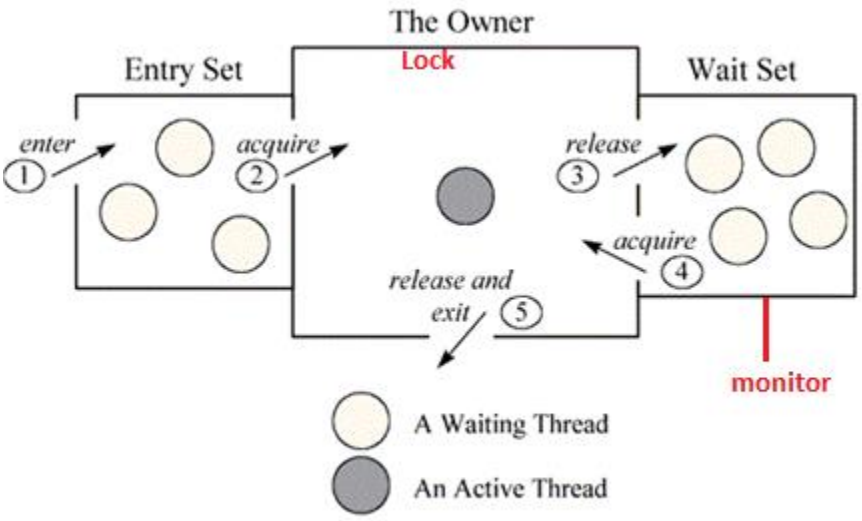
```
public void b()
{
    synchronized(someObject) {
        // do something else (2)
    }
}
```

This prevents Threads 1 and 2 accessing the monitored (synchronized) section at the same time. One will start, and monitor will prevent the other from accessing the region before the first one finishes.





8.11.2 Difference between lock and monitor



Above figure shows the monitor as three rectangles. In the center, a large rectangle contains a single thread, the monitor's owner. On the left, a small rectangle contains the entry set. On the right, another small rectangle contains the wait set.

Locks help threads to work independently on shared data without interfering with one another, wait-sets help threads to cooperate with one another to work together towards a common goal e.g. all waiting threads will be moved to this wait-set and all will be notified once lock is released. This wait-set helps in building monitors with additional help of lock (mutex).

Example Inter thread Communication

```
class Customer {
	int amount = 10000;

	synchronized void withdraw(int amount) {
		System.out.println("WITHDRAWING \n***************");
		if (this.amount < amount) {
		System.out.println(" LESS BALANCE !!!!");
		try {
	System.out.println("withdraw() is on wait untill deposit() notify ");
				wait();
			} catch (Exception e) {
			}
		}
		this.amount -= amount;
		System.out.println(" ******* WITHDRAW COMPLETED **********");
	}

	synchronized void deposit(int amount) {
		System.out.println("\n\n DEPOSITING \n ***************");
		this.amount += amount;
		System.out.println("DEPOSIT COMPLETED ");
		System.out.println("calling nofity on withdraw()");
		notify();
	}
}

public class InterThreadCom {
	public static void main(String args[]) {
		final Customer c = new Customer();
		new Thread() {
			public void run() {
				c.withdraw(15000);
			}
		}.start();
		new Thread() {
			public void run() {
				c.deposit(10000);
			}
		}.start();
	}
}
```

```
WITHDRAWING
***************
 LESS BALANCE !!!!
withdraw() is on wait untill deposit() notify

 DEPOSITING
 ***************
DEPOSIT COMPLETED
calling nofity on withdraw()
 ******* WITHDRAW COMPLETED **********
```

8.11.3 Difference between wait and sleep

Let's see the important differences between wait and sleep methods.

wait()	sleep()
wait() method releases the lock	sleep() method doesn't release the lock.
is the method of Object class	is the method of Thread class
is the non-static method	is the static method
is the non-static method	is the static method
should be notified by notify() or notifyAll() methods	after the specified amount of time, sleep is completed.

wait(), notify() and notifyAll() methods are defined in Object class s because they are related to lock and object has a lock

8.12 Interrupting a Thread

If any thread is in sleeping or waiting state (i.e. sleep() or wait()), calling the interrupt() method on the thread, breaks out the sleeping or waiting state throwing InterruptedException.

- **public void interrupt()** - Inturpting a Thread
- **public static boolean interrupted()**
  It is a Static method, tests whether the CURRENTLY running thread is interrupted or not
- **public boolean isInterrupted()**
  It is a instance method and tests whether the thread instance on which the method is invoked is interrupted or not

If the thread is not in the sleeping or waiting state, calling the interrupt() method performs normal behaviour and doesn't interrupt the thread but sets the interrupt flag to true

Thread is Interrupted & Stops working

```
public class InterruptNormal extends Thread {
	public void run() {
		try {
			Thread.sleep(1000);
			System.out.println("task");
		} catch (InterruptedException e) {
			throw new RuntimeException("Thread interrupted..." + e);
		}
		System.out.println("Thread is Running ...");
	}
	public static void main(String args[]) {
		InterruptNormal t1 = new InterruptNormal();
		t1.start();
		try {
			t1.interrupt();
		} catch (Exception e) {
			System.out.println("Exception handled " + e);
		}
	}
}
```

```
Exception in thread "Thread-0" java.lang.RuntimeException: Thread
interrupted...java.lang.InterruptedException: sleep interrupted
	at threads.InterruptNormal.run(InterruptNormal.java:9)
```

above Example your are re throwing IntereuptException. So thraed is Inturrpeed & also stops its execution

Thread is Interrupted & doesn't Stops working

```
public class InterruptHandled extends Thread {
	public void run() {
		try {
			Thread.sleep(3000);
			System.out.println(" *** Sleep is Still Running ****");
		} catch (InterruptedException e) {
			System.out.println("Thread interrupted..." + e);
		}
		System.out.println("Thread is Running ...");
	}
	public static void main(String args[]) {
		InterruptHandled t1 = new InterruptHandled();
		t1.start();
		try {
			t1.interrupt();
		} catch (Exception e) {
			System.out.println("Exception handled " + e);
		}
	}
}
```

```
Thread interrupted...java.lang.InterruptedException: sleep interrupted
Thread is Running ...
```

above Example Exception is handled. It is only interruped sleeping thraed.reaming are excuting as normal

If thread is not in sleeping or waiting state, calling the interrupt() method sets the interrupted flag to true that can be used to stop the thread by the java programmer later.

```
public class InterruptHandled extends Thread {
	public void run() {
		System.out.println(" *** No Sleep is Here ****");
		System.out.println("Thread is Running ...");
	}

	public static void main(String args[]) {
		InterruptHandled t1 = new InterruptHandled();
		t1.start();
		try {
			t1.interrupt();
		} catch (Exception e) {
			System.out.println("Exception handled " + e);
		}
	}
}
```

```
*** No Sleep is Here ****
Thread is Running ...
```

8.13 Thread Pool

Java 5 added a new Java package to the Java platform, the java.util.concurrent package. This package contains a set of classes that makes it easier to develop concurrent (multithreaded) applications in Java

JDK provides a set of ready-to-use **data structures and functionality for developing advance multi-threaded applications**. All these classes reside within the package java.util.concurrent.

8.13.1 Lock Interface (`java.util.concurrent.locks.Lock`)

Lock Object is similar to implecit lock aquired by a thraed to execute synchronized method or synchrized block.Lock implemetions provide more extencive operation then traditional implicit locks.

Methods

1.void lock() – To aquire Lock.if lock is already availabale current thread will get that lock.if lock is not available it waits untill get the lock.it is similar to sysnchronized keyword

2.boolean tryLock() – To quire lock waitout waiting.if it quires lock returns true, if not false & continues it execution without waiting.in this case thread never goes into waiting state

```
if(l.tryLock())
{//perform safe operations
}else{
	//perform alternative operations
}
```

3.boolean tryLock(long time, TimeUnit unit) – Same as above, but specifying time.TimeUnit is Enum having values as NANOSECONDS,SECONDS,MINITUES,HOURS,DAYS

```
if(l.tryLock(1000,TimeUnit.MINITUES)) //waiting for 1000 minitues
{//perform safe operations
}else{
      //perform alternative operations
}
```

4. Void lockInterruptibly() – Aquires lock if available & returns immdefiatly. Not available it will wait.while waiting if thread is interruped then thread wont get the lock.

5.void unlock() – Releases the lock.if we call on thread which is not having lock it will thorws runtime exception IllegalMonitorStateException

Implemeting Classes

8.13.2 ReentrantLock

it is the **implementation** class of **Lock interface** & direct child class of Object.Reentrant means A thraed can aquire same lock multiple times without any issue.

Internally ReentrantLock increments threads personel count when ever we call lock() & decrements count() value when ever thraed calls unlock(). Lock will realesed when ever count reaches 0

8.13.3 Thread Pool

Normally we will create 10 threads for completing 10 jobs. This is reduce the performance.

Thread pool is a pool of already created threads, ready to do our job. Java 1.5 introduced **Executor framework** to implement thread pool.

1. We create thread pool of 3 threads
ExcecutorService service = Executors.newFixedThreadPool(3)

2. we can submit a runnable job by using sumbit() method
service.submit(job)

3. we can shutdown executor service by using shutdown() method
Service.shutdown()

```
class PrintJob implements Runnable {
	String name;
	public PrintJob(String name) {
		this.name = name;
	}

	@Override
	public void run() {
System.out.println(name + "\t ..Job STARTED by Thread:\t" +
Thread.currentThread().getName());
		try {
			Thread.sleep(5000);
		} catch (InterruptedException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
		System.out.println(name + "\t ..Job COMPLETED by Thread:\t" +
Thread.currentThread().getName());
	}
}

public class ExcecutorDemo {
	public static void main(String[] args) {
		PrintJob[] jobs = { new PrintJob("1.Satya"), new
PrintJob("2.Ravi"), new PrintJob("3.Rakes"),
				new PrintJob("4.Surya"), new PrintJob("5.Madhu"), };
		ExecutorService service = Executors.newFixedThreadPool(5);

		for (PrintJob printJob : jobs) {
			service.submit(printJob);
		}
		service.shutdown();
	}
}
```

```
1.Satya  ..Job STARTED by Thread:       pool-1-thread-1
4.Surya  ..Job STARTED by Thread:       pool-1-thread-4
2.Ravi   ..Job STARTED by Thread:       pool-1-thread-2
5.Madhu  ..Job STARTED by Thread:       pool-1-thread-5
3.Rakes  ..Job STARTED by Thread:       pool-1-thread-3
5.Madhu  ..Job COMPLETED by Thread:     pool-1-thread-5
3.Rakes  ..Job COMPLETED by Thread:     pool-1-thread-3
2.Ravi   ..Job COMPLETED by Thread:     pool-1-thread-2
1.Satya  ..Job COMPLETED by Thread:     pool-1-thread-1
4.Surya  ..Job COMPLETED by Thread:     pool-1-thread-4
```

8.13.4 Callable interface

Public void run() method won't return any thing after completing job & also we must handle the Runnable Exception.

If a thread is required to return some result after execution the we go for Callable interface.it contains only one method **call();**

```
Public Object call() throws Exceptions
```

If call service.submit(c) it returns **Feature**. Feature Object can be used to retrive the result from callble job by using **f.get()** .

```
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
class CallableJob implements Callable {
	int num;

	public CallableJob(int num) {
		this.num = num;
	}

	@Override
	public Object call() throws Exception {
		System.out.println(num + "\t ..SUM STARTED by Thread:\t" +
Thread.currentThread().getName());
		int sum = 0;
		for (int i = 0; i < num; i++) {
			sum = sum + i;
		}
System.out.println(num + "\t ..SUM COMPLETED by Thread:\t" +
Thread.currentThread().getName());
		return sum;
	}
}

public class CallableDemo {
	public static void main(String[] args) throws InterruptedException,
ExecutionException {
		CallableJob[] jobs = { new CallableJob(10), new CallableJob(20),
new CallableJob(30), new CallableJob(40),
				new CallableJob(50) };
		ExecutorService service = Executors.newFixedThreadPool(2);

		for (CallableJob CallableJob : jobs) {
			Future f = service.submit(CallableJob);
			System.out.println("Sum is : " + f.get() + "\n");
		}
		service.shutdown();
	}
}
```

```
10	..SUM STARTED by Thread:	pool-1-thread-1
10	..SUM COMPLETED by Thread:	pool-1-thread-1
Sum is : 45

20	..SUM STARTED by Thread:	pool-1-thread-2
20	..SUM COMPLETED by Thread:	pool-1-thread-2
Sum is : 190
30	..SUM STARTED by Thread:	pool-1-thread-1
30	..SUM COMPLETED by Thread:	pool-1-thread-1
Sum is : 435
40	..SUM STARTED by Thread:	pool-1-thread-2
40	..SUM COMPLETED by Thread:	pool-1-thread-2
Sum is : 780
50	..SUM STARTED by Thread:	pool-1-thread-1
50	..SUM COMPLETED by Thread:	pool-1-thread-1
Sum is : 1225
```

IX. Java Collections

If you want to use 10 **variables** in your program you declare like

```
int a1=10;
int a2=20;
int a3=30;
 .......
int a10=100;
```

This makes code complex & Performance will decreases. So **Arrays** came into picture to avoid those.

Arrays:

```
int a[] = new int[10];
      [or]
int a1[] = new int[]{10,20,30,40,50,60,70,80};
```

But **Arrays size is fixed** &, they don't dynamically increase size. And **don't allows different data types.**

Collection:

If you want to represent a group of individual objects as a Single Entity then we should go for **Collection**.

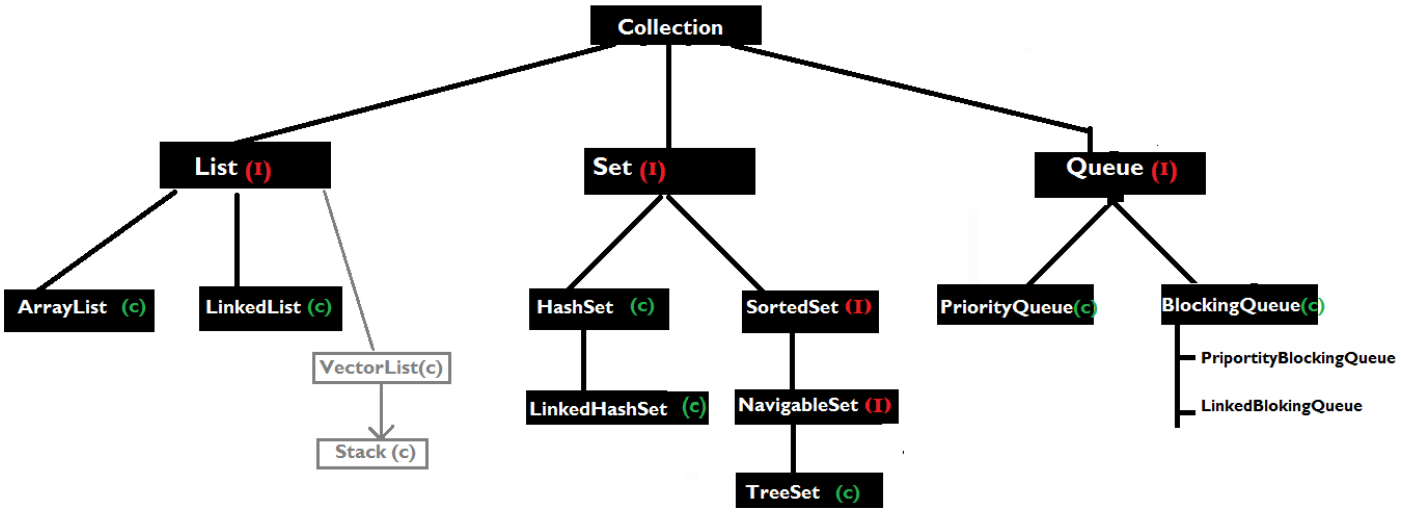
Arrays	Collections
Arrays are of **fixed size.**	Collections are **dynamic in nature**
Arrays **consume more memory**. When the size of an integer array is declared as 100, then 400 bytes of memory (4 bytes * 100 elements) are reserved irrespective of number of elements actually placed in the array.	Collection **consumes less memory**. Since they are dynamic in nature, memory space will be reserved only the actual number of elements in the collection.
Arrays can only **hold elements of the same type**.	Collections can **hold elements of same type as well as elements of different types**.
Both primitive data types and objects can be stored in arrays.	Collections can **only store objects**.
No readily available methods because it is not D.s	Have readily available methods.

9.1 Collection Framework

Collection framework defines several Classes & interfaces to represent group of Objects into single unit.

Java.util.collection: root **interface** in the collection hierarchy. Used to represent group of objects into single entity

Java.util.collection: is a **class** define utility methods to perform useful operations on collection



9.2 Java.util.collection

Collection interface defines most common methods which can be applied for any collection Object

1.	**Boolean add(Object o)**	- add a single object to collection
2.	**Boolean addAll(Collection c)**	- add a collection Object to collection
3.	**Boolean remove(Object o)**	- removes a single object to collection
4.	**Boolean removeAll(Collection c)**	- removes a collection Object to collection
5.	**Boolean retainAll(Collection c)**	- Except give collection Object remove remaining object
6.	**Boolean contains(Object o)**	- Check give object is there or not
7.	**Boolean containsAll(Collection c)**	- Check given collection objects are there or not
8.	**Void isEmpty()**	- check is empty or not
9.	**Int size()**	- Check size of a collection
10.	**Object[] toArray()**	- Convert collection to Array
11.	**Void clear()**	
12.	**Iterator iterator()**	

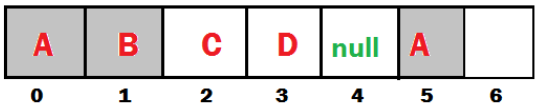
The functionality of Enumeration and the Iterator are same. You can get **remove() from Iterator to remove an element**, **while Enumeration does not have remove() method**. Using Enumeration you can only traverse and fetch the objects, where as using Iterator we can also add and remove the objects.

```
Enumeration                                  Iterator
----------------                             ----------------
Boolean hasMoreElement()                     Boolean hasNext()
E       nextElement()                        E       next()
N/A                                          void    remove()
```

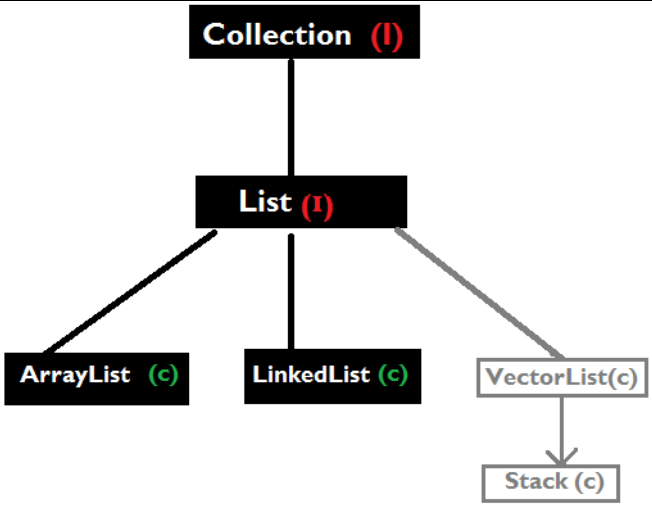
Every collection is implemented by using underlying data structure

9.3 java.util.List (Interface)

- List is child interface of collection
- If we want to represent group of individual objects as a single entity where **duplicates are allowed & insertion order must be preserved** then we should go for List
- We can **preserve insertion order via index & differentiate duplicate objects using index**
- **Index** will play very important role in List



Add / Remove	Find	Special
Boolean add(int index, Object o) **Boolean addAll(int index,Collection c)** **Boolean remove(int index)**	**Object get(int index)** **Object set(int Object new)** **Int indexOf(Object c)** **Int lastIndexOf(Object c)**	**ListIterator listIterator()**



9.3.1 ArrayList

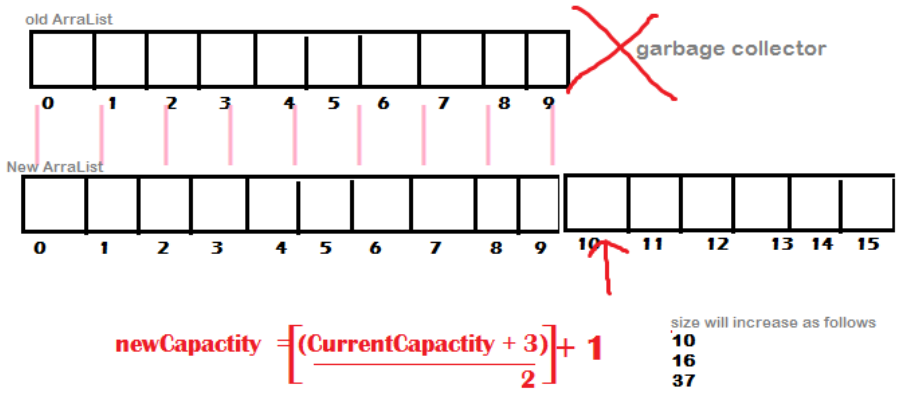
- The underlying data structure is **ResizableArray** or **Growable** Array
- Duplicates are allowed
- Insertion order is preserved
- Heterogeneous(different datatypes) Objects are allowed
- Null is insertion is allowed
- ArrayList implements Serializable, Clonable & RandomAccess

Except TreeSet & TreeMap Everywhere Heterogeneous Objects are allowed

Constructors

1. ArrayList l = new ArrayList()

Creates an Empty ArrayList Object with default initial capacity 10.Once ArrayList reaches its max capacity the new Arraylist object is created with newcapacity = (currentcapacity + 3/2)+1 & old one will give to Garbage collector.



2. ArrayList l = new ArrayList(int intialcapacity)

Creates an Empty ArrayList Object with specified initial capacity

3. ArrayList l = new ArrayList(Collection c)

Creates ArrayList Object for the given collection

```
public class ArrayListDemo {
	public static void main(String[] args) {
		ArrayList l = new ArrayList<>();
		l.add("A");
		l.add(10);
		l.add("A");
		l.add(null);
		System.out.println(l); // [A, 10, A, null]
		l.remove(2);
		System.out.println(l); // [A, 10, null]
		l.add(2, "M");
		System.out.println(l); // [A, 10, M, null]

		l.add("N");
		System.out.println(l); // [A, 10, M, null, N]
	}
}
```

From 1.5 onwards generics are introduced for type safety. So if we use ArrayList without generics it will display below notification

```
D:\Workspace\smlcodes\src\collections>javac ArrayListDemo.java
Note: ArrayListDemo.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

D:\Workspace\smlcodes\src\collections>
```

Important Points

- Usually we use collections to **hold & transfer objects from one location to another location**. To provide support for this requirement every collection class implements **Serializable & Clonable interfaces**
- **ArrayList and Vector classes** implements **RandomAccess interface**, so that any random element we can access with same speed. **RandomAccess is a marker interface** & doesn't have any methods
- **Insertion/Deletion** is middle ArrayList is the **Worst choice**. For **retrieval Best Choice**
- In every collection class **toString()** is overridden to print data readable format **Ex. [ob1, ob2, ob3]**

ArrayList	Vector
• Methods are **Non-Synchronized** • **Not thread Safe**, multiple threads may access at time • **Performance is High**, because no wait threads • Introduced in **1.2(New)**	• Methods are **Synchronized** • **Thread Safe**, multiple threads may access at time. • **Performance is Low**, because threads may wait. • Introduced in **1.0 version(Legacy)**

How to make ArrayList as Synchronized

To make ArrayList as Synchronized, we have a static method in java.util.Collections Interface

public static List sychronizedList(List l)

Example

```
ArrayList l = new ArrayList();
List l2 = Collections.synchronizedList(l);
```

Similarly we can get Synchronized version of **Set & Map** Object

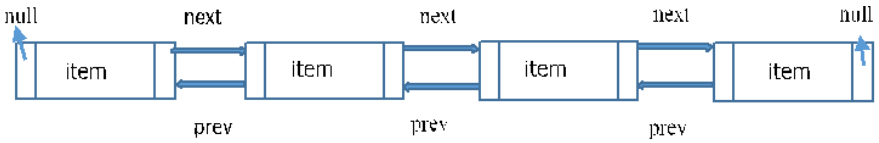
- **public static Set sychronizedSet(List l)**
- **public static Map sychronizedSet(List l)**

Best for Retrieval Operations, Worst for Insert/Deletion

9.3.2 LinkedList

- Underlying data structure is **DoubleLinkedList**
- **Insertion** order is preserved

- **Duplicates** are **allowed**
- **Heterogeneous** objects are **allowed**
- **Null** insertion is **allowed**
- LinkedList **implements Serializable & Clonable interfaces but not RandomAccess**
- **Best Choice for Insertion/Deletion, Worst for Retrieval operation**



1. LinkedList l = new LinkedList ()

Creates an Empty LinkedList Object with unlimited initial capacity

2. LinkedList l = new LinkedList (Collection c)

Creates ArrayList LinkedList for the given collection

3. LinkedList class Specific methods

Usually we can use LinkedList to develop Stacks & Queues.to provide support for these requirement

LinkedList class defines the following methods

- **Void addFirst(Object o)**
- **Void addLast(Object o)**
- **Object getFirst()**
- **Object getLast()**
- **Object removeFirst()**
- **Object removeLast()**

```
public class LinkedListDemo {
	public static void main(String[] args) {
		LinkedList l = new LinkedList<>();
		l.add("A");
		l.add(10);
		l.add("A");
		l.add(null);
		System.out.println(l); // [A, 10, A, null]
		l.set(0, "Satya"); // replaces
		System.out.println(l); // [Satya, 10, A, null]
		l.add(0, "Johnny"); // just add
		System.out.println(l); // [Johnny, Satya, 10, A, null]
		l.removeFirst();
		System.out.println(l); // [Satya, 10, A, null]
		System.out.println(l.getFirst());// Satya
	}
}
```

```
[A, 10, A, null]
[Satya, 10, A, null]
[Johnny, Satya, 10, A, null]
[Satya, 10, A, null]
Satya
```

9.3.3 Vector

- The underlying data structure is **ResizableArray** or **Growable** Array
- Duplicates are allowed
- Insertion order is preserved
- Heterogeneous(different datatypes) Objects are allowed
- Null is insertion is allowed
- Vector implements Serializable, Clonable & RandomAccess
- Vector is Synchronized

Constructors

1. Vector v = new Vector()

Creates an Empty Vector Object with default initial capacity 10.Once Vector reaches its max capacity the new Vector object is created with newcapacity = (currentcapacity*2)& old one will give to Garbz collector.

2. Vector v = new Vector(int intial capacity)

3. Vector v = new Vector(int intial capacity, int incrementalCapacity) //currcapacity+100

4. Vector v = new Vector(Collection c)

Methods

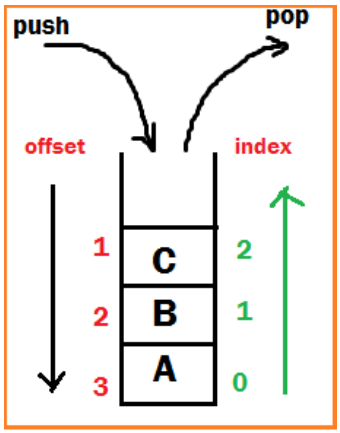
Add / Remove	Find	Special
addElement(Obejct o)	**Object elementAt(int index)**	**Int size()**
removeElement(Object o)	**Object firstElement()**	**Int capacity()**
removeElementAt(int index)	**Object lastElemet()**	//to know default/incremental capacity
removeAllElements()		**Enumeration elements()**

```
public class VectorDemo {
public static void main(String[] args) {
	Vector v = new Vector();
	for (int i = 1; i <=10; i++) {
		v.addElement(i);
	}
System.out.println("Before adding 11th element -Capacity:"+v.capacity()); //
	v.addElement("Satya");
	System.out.println(v);
System.out.println("After adding 11th element -Capacity:"+v.capacity());
	System.out.println("size : "+v.size());
}
}
```

```
Before adding 11th element -Capacity:10
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, Satya]
After adding 11th element -Capacity:20
size : 11
```

9.3.4 Stack

The Stack class represents a **last-in-first-out (LIFO)** stack of objects. It extends class Vector with **five operations** (below 5 methods) that allow a vector to be treated as a stack



1. Object push(Object o) -Insert an object into top of the stack

2. Object pop() -Removes & returns from top of the stack

3. Object peak() -Just returns Object from top of the stack

4. boolean empty() -returns TRUE if stack is empty

5. int search(Object o) - returns offset if available otherwise -1

Stack s = new Stack() //only one constrcutor

```
public class StackDemo {
	public static void main(String[] args) {
		Stack s = new Stack();
		s.push("A");
		s.push("B");
		s.push("C");
		System.out.println(s);// [A, B, C]

		System.out.println(s.search("A"));// 3
		System.out.println(s.search("X"));// -1

		s.pop();
		System.out.println(s);//[A, B]
	}
}
```

```
[A, B, C]
3
-1
[A, B]
```

9.3.5 Cursors - Enumeration VS Iterator VS ListIterator

ListIterator is subclass of List.so all the methods are available in ListIterator.

We have to follow 3 steps to use Cursors in our application

1. Get the all elements in a collection in Cursor Object
2. Check is next/previous element is exist or not
3. Get the element

Enumeration (vector/stack)	**Iterator(**ArrayList**)**	**ListIterator (**LinkedList**)**
Can iterate over an Collection	Can iterate over an Collection	Can iterate over an Collection
Remove operation not allowed	Remove operation allowed	Remove operation allowed
Add operation not allowed	**Add operation not allowed**	Add operation allowed
Backward direction not allowed	**Backward direction not allowed**	Backward direction allowed
1.Public Enumeration elements() Ex. Enumeration = v.elemetnts() **2.Boolean hasMoreElements()** **3.E nextElement()**	**1.Public Iterator iterator()** Ex. Iterator = l.iterator() **2.Boolean hasNext()** **3.E next()** **void remove()**	**1.Public ListIterator listIterator()** Ex. ListIterator = l.listIterator () **add(E e)** **hasPrevious()** **nextIndex()** **previous()** **previousIndex()** **remove()** **hasNext()** **next()** **set(E e)**

Enumeration Example using Vector

```
public class VectorEnumation {
public static void main(String[] args) {
 Vector v = new Vector();
 for (int i = 1; i <= 10; i++) {
  v.addElement(i);
      }
Enumeration e = v.elements();
while (e.hasMoreElements()) {
 Object s = (Object) e.nextElement();
 System.out.print(s + ",");
  }
 }
}
```

```
1,2,3,4,5,6,7,8,9,10,
```

Iterator Example using ArrayList

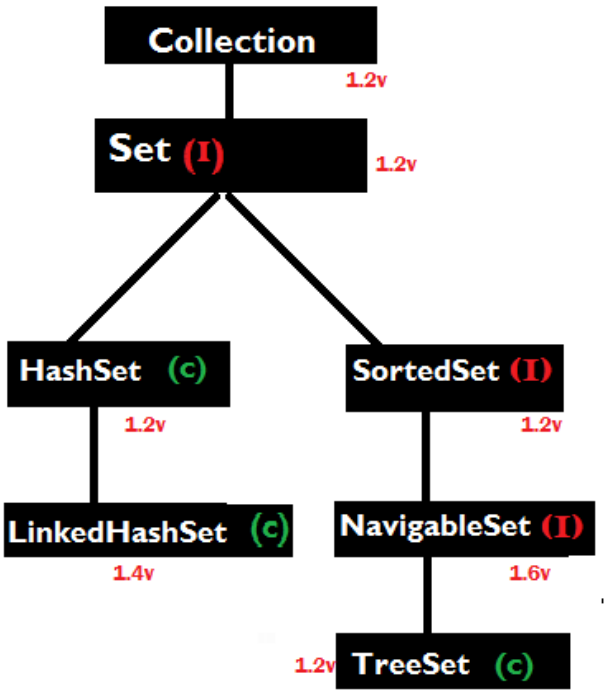
```
public class ArrayListIterator {
public static void main(String[] a) {
      ArrayList l = new ArrayList<>();
      l.add("A");
      l.add(10);
      l.add("A");
      l.add(null);
Iterator i= l.iterator();
while (i.hasNext()) {
      Object s = (Object) i.next();
      System.out.print(s+",");   }
 }
}
```

```
A,10,A,null,
```

9.4 java.util.Set (Interface)

- Set is child interface of collection
- If we want to represent group of individual objects as a single entity where **duplicates are Not allowed & insertion order Not be preserved** then we should go for List
- Set Interface doesn't contain any new method & we have to use only collection interface methods

In All Hash related collections insertion is based on Hashcode.so no insertion order preserved.



9.4.1 HashSet

- The underlying datastructure is Hashtable
- Duplicate Objects are Not Allowed
- Insertion Order is Not preserved & it is based hash code of Objects
- Null Insertion is possible(Only once)
- Heterogeneous Objects are allowed
- Implements Serializable & Clonable but not RandomAccess Interface
- HashSet is the Best Choice for Search Operation

In HashSet Duplicates are not allowed. If we are trying to insert duplicates then it won't get any Compile time or Runtime Error and add() method simply returns FALSE

1. HashSet h = new HashSet () **//16 capacity, Def. fill ratio = 0.75**

Creates an empty Object with def. initial capacity 16 & def. fill ratio 0.75

2. HashSet h = new HashSet (int intialcapacity) **// Def. fill ratio = 0.75**

3. HashSet h = new HashSet (int intialcapacity, float fillRatio)

4. HashSet h = new HashSet (Collection c)

Fill ratio (Load Factor): After filling how much ratio a new Object will be created, this ratio is called fill ratio/ load Factor

```
public class HashSetDemo {
public static void main(String[] args) {
	HashSet h = new HashSet();
	h.add("A");
	h.add("B");
	h.add("C");
	h.add(10);
	h.add(null);
	System.out.println(h.add("A"));//False
	System.out.println(h);
}
}
```

```
false
[null, A, B, C, 10]
```

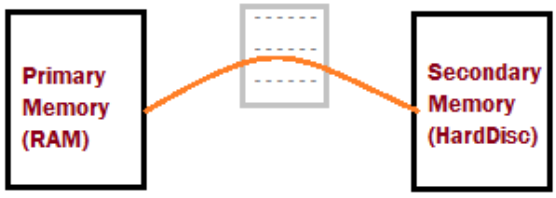
9.4.2 LinkedHashSet

LinkedHashSet is similar to HashSet (including constructors & methods) only difference is it preserves Insertion Order. Below are differences between them

HashSet (1.2v)	LinkedHashSet(1.4v)
• Underlying D.S is Hashtable • Insertion order is Not Preserved	• Underlying D.S is **Hashtable +LinkedList** • Insertion order is Preserved

```
public class HashSetDemo {
public static void main(String[] args) {
	LinkedHashSet h = new LinkedHashSet();
	h.add("A");
	h.add("B");
	h.add("C");
	h.add(10);
	h.add(null);
	System.out.println(h.add("A"));
	System.out.println(h);
}
}
```

```
false
[A, B, C, 10, null]
```

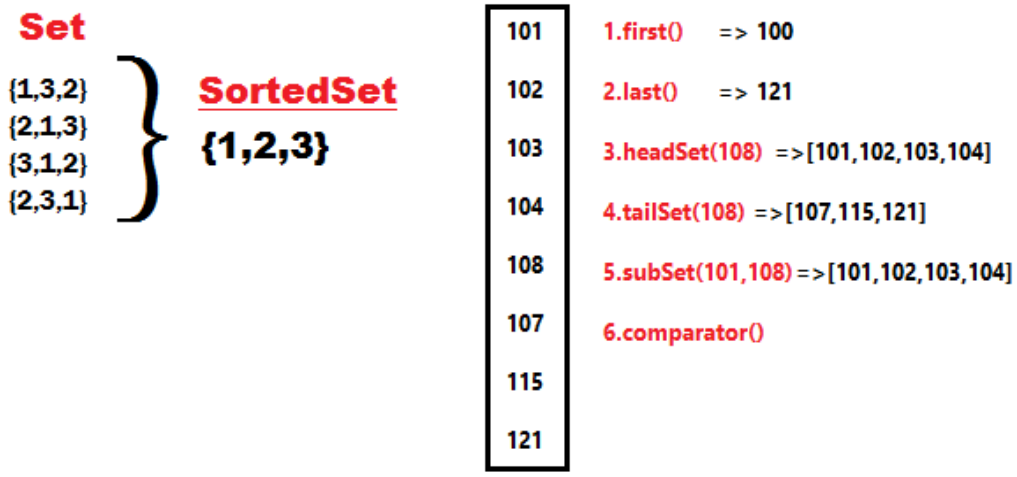


In general we can use **LinkedHashSet** to develop cache based applications where duplicates are not allowed & Insertion order must be preserved

9.5 java.util.SortedSet (Interface)

- Is the child interface of Set
- If we want to represent a group of individual objects according to some sorting order without duplicates then we should go for SortedSet

SortedSet Interface defines following 6 methods.



1. **Object first( )**
2. **Object last( )**
3. **SortedSet headSet(Object obj)**
4. **SortedSet tailSet(Object obj)**
5. **SortedSet subSet(Object start, Object end)**

6. **Comparator comparator()**

Used to get Default Natural sorting order

- **Numbers** ➔ Ascending order [1, 2, 3, 4, 5....]
- **Strings** ➔ Alphabetical Order [A, B, C, D, E...a,b,c,d ...] (Unicode values)

9.5.1 TreeSet

- Underlying D.S is BalancedTree
- Duplicate Objects are Not Allowed
- Insertion order Not Preserved but we can sort elements
- Heterogeneous Objects are Not Allowed , if try it throws `ClassCastException` at Runtime
- Null Insertion allowed(Only once)
- TreeSet implements Serializable & Clonable but not RandomAccess
- All objects are inserted based on some sorting order either default or customized sorting order

1. TreeSet h = new TreeSet () **//Default. SortingOrder**

Creates an Empty TreeSet Object, all the elements inserted according to Default Natural SortingOrder

2. TreeSet h = new TreeSet (Comparator c) **//Customized. SortingOrder**

Creates an Empty TreeSet Object, all the elements inserted according to Customized Natural SortingOrder

3. TreeSet h = new TreeSet (Collection c)

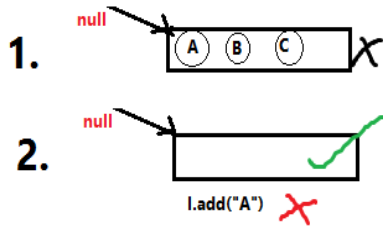
4. TreeSet h = new TreeSet (SortedSet s)

```
public class TreeSetDemo {
public static void main(String[] args) {
	TreeSet t = new TreeSet();
	t.add("A");
	t.add("N");
	t.add("Z");
	t.add("h");
	t.add("X");
	t.add("i");
	//t.add(10);
	//Exception in thread "main" java.lang.ClassCastException:
	//java.lang.String cannot be cast to java.lang.Integer

	//t.add(null); // java.lang.NullPointerException
	System.out.println(t);
}
}
```

```
[A, N, X, Z, h, i]
```

1. For Non-Empty TreeSet if we are trying to insert null we will get NullPointerException

2. For Empty TreeSet, as the 1st element we can Insert Null but after that if try to insert any data it will again throws NullPointerException



From 1.7 Version onwards it won't allow null even as the 1st element

```
public class TreeSetStringBuffer {
public static void main(String[] args) {
	TreeSet t = new TreeSet();
	t.add(new StringBuffer("A"));
	t.add(new StringBuffer("X"));
	t.add(new StringBuffer("O"));
	t.add(new StringBuffer("L"));
	System.out.println(t);
}
}
```

```
Exception in thread "main" java.lang.ClassCastException:
java.lang.StringBuffer cannot be cast to java.lang.Comparable
```

If we are depending on Def. Natural SortingOrder objects should be homogeneous & Comparable. Otherwise we will get Runtime Exception `java.lang.ClassCastException.`

An object is said to be comparable if and only if corresponding class implements Comparable interface

Java.lang.String & all wrapper classes (Int, Float, Byte) already implements Comparable interface

```
public final class java.lang.String implements java.io.Serializable, java.lang.Comparable
```

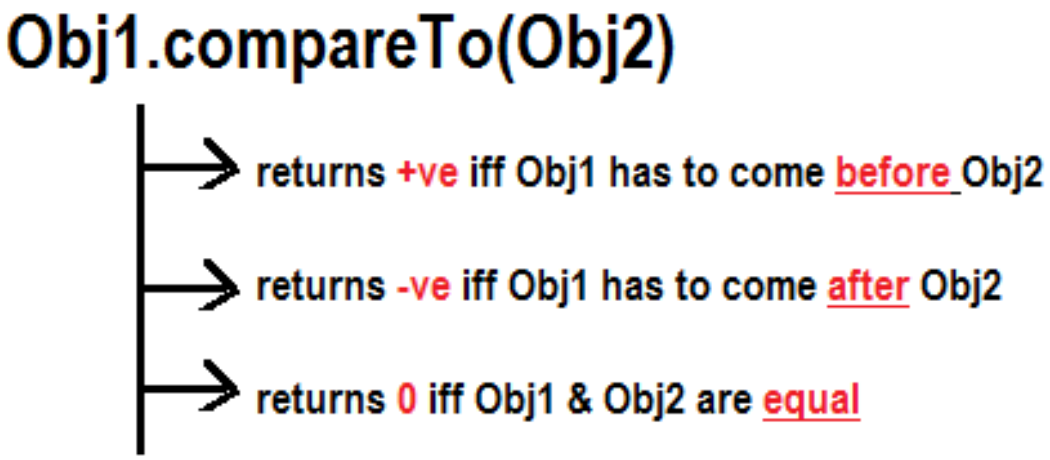
Java.lang.StringBuffer doesn't implements comparable interface

```
public final class java.lang.StringBuffer extends java.lang.AbstractStringBuilder implements
java.io.Serializable,java.lang.CharSequence
```

i. Comparable Interface

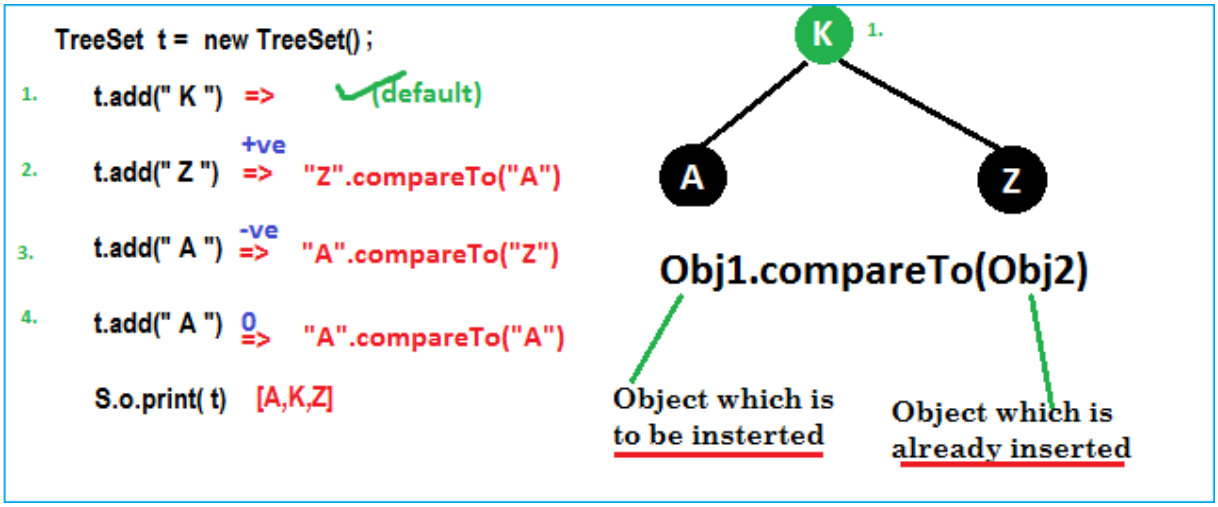
It is present in **java.lang**. Package & it contains only one method **compareTo()**

```
public  int  compareTo(Object obj)
```




```
public class ComparableDemo {
	public static void main(String[] args) {
		System.out.println("A".compareTo("Z")); // 1-26 = -25
		System.out.println("Z".compareTo("C")); // 26-3 = 23
		System.out.println("A".compareTo("A")); // 1-1 = 0
		// System.out.println("A".compareTo(null)); //R.E NPE
	}
}
```

While adding Objects into TreeSet JVM will call compareTo() method

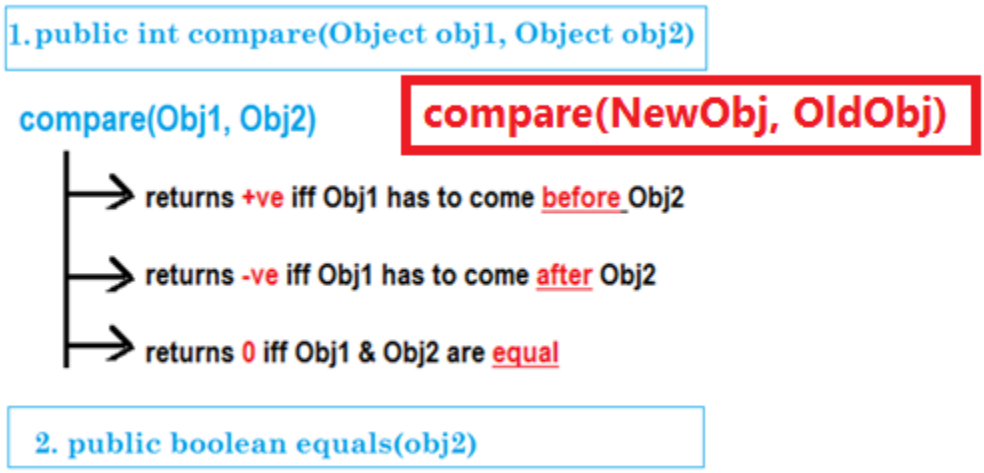


If default Natural sorting order not available or if we are not satisfied with default natural sorting order then we can go for customized sorting by using **comparator**

- Comparable ➔ for Default Natural Sorting Order
- Comparator ➔ for Customized sorting Order

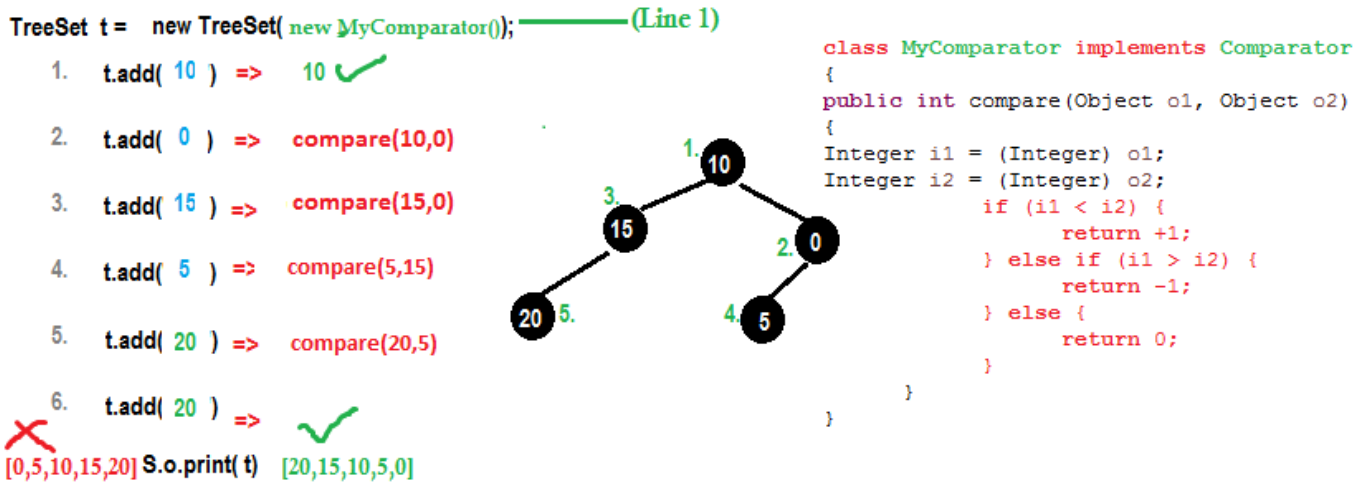
ii. Comparator Interface

Comparator present in java.util package & it defines two methods **compare (ob1, ob2) & equals (ob1)**



Whenever we are implementing comparator interface we should provide implementation only for **compare ()** method & we are not required implementation for **equals()** method, because it is already available to our class from Object class through inheritance

Example: To insert integer Objects into TreeSet where Sorting Order is Descending order




```
public class TreesetComp {
	public static void main(String[] args) {
		TreeSet t = new TreeSet(new MyComparator());
		t.add(10);
		t.add(0);
		t.add(15);
		t.add(5);
		t.add(20);
		t.add(20);
		System.out.println(t);
	}
}

class MyComparator implements Comparator {
	@Override
	public int compare(Object newObj, Object oldObj) {
		Integer i1 = (Integer) newObj;
		Integer i2 = (Integer) oldObj;
		if (i1 < i2) {
			return +1;
		} else if (i1 > i2) {
			return -1;
		} else {
			return 0;
		}
	}
}
```

```
[20, 15, 10, 5, 0]
```

At **Line1,** if we not passing MyComparator object then internally JVM will call **compareTo()** method which is for default Natural Sorting order.in this case output is [0,5,10,15,20].

At **Line1,** if we passing MyComparator object then JVM will call **compare()** method which is for customize Sorting order.in this case output is [20,15,10,5,0].

```
class MyComparator implements Comparator
{
public int compare(Object o1, Object o2)
{
Integer i1 = (Integer) o1;
Integer i2 = (Integer) o2;
```

1. return i1.compareTo(i2) //[0,5,10...]
2. -return i1.compareTo(i2) //[20,15,10..]
3. return i2.compareTo(i1) //[20,15,10..]
4. -return i2.compareTo(i1) //[0,5,10]
5. return +1 //[Insertion Order]
6. return -1; //Reverse of Insertion order
7. return 0; //Only 1st element will insert remaining are Duplicates

```
    }
 }
```

various possible implemetations of compare() method

As the same way if we want to change String order we do as follows

```
public class TreesetStringComp {
	public static void main(String[] args) {
		TreeSet t = new TreeSet(new MyComparators());
		t.add("HYDERABAD");
		t.add("VIJAYAWADA");
		t.add("BANGLORE");
		t.add("VIZAG");
		System.out.println(t);
	}
}
class MyComparators implements Comparator {
	public int compare(Object newObj, Object oldObj) {
		String s1 = (String) newObj;
		String s2 = (String) oldObj;
		int i1 = s1.length();
		int i2 = s2.length();
		if (i1 < i2) {
			return +1;
		} else if (i1 > i2) {
			return -1;
		} else {
			return 0;
		}
	}
}
```

```
[VIJAYAWADA, HYDERABAD, BANGLORE, VIZAG]
```

Comparable	Comparator
1.it is meant for **Defalut Nartural Sorting order**	1.it is meant for **Customized Nartural Sorting order**
2.present in **java.lang** package	2.present in **java.util** package
3.it defines only one method compareTo()	3.it defines 2 methods compare() & equals()
4.**String & all Wrapper classes** implments Comparable interface	4.only 2 GUI classes Collator&RulebasedCollator implemets Comparator

Propery	HashSet	LinkedHastSet	TreeSet
Underlying D.S	**Hashtable**	**LinkedList+Hashtable**	**Balenced Tree**
Duplicate Objects	**Not Allowed**	**Not Allowed**	**Not Allowed**
Index Order	**Not Preserved**	**Preserved**	**Not Preserved**
Sorting Order	**Not Applicable**	**Not Applicable**	**Allowed**
Hetrogenious Objects	**Allowed**	**Allowed**	**Not Allowed**
Null allowed	**Allowed**	**Allowed**	**Only 1st one(not <1.7V)**

As the part of 1.6 version the following two concepts introduced in collection framework

1. NavigableSet interface

2. NavigableMap interface

9.5.2. NavigableSet

It is the Child interface of SortedSet & it defines servaral methods for Navigation purpose

Methods

1. floor(e) - it returns highest element which is <=e

2. lower(e) - it returns highest element which is <e

3. ceiling(e) - it returns lowest element which is >=e

4. higher(e) - it returns lowest element which is >e

5. pollFisrt() - removes & returns first elemet

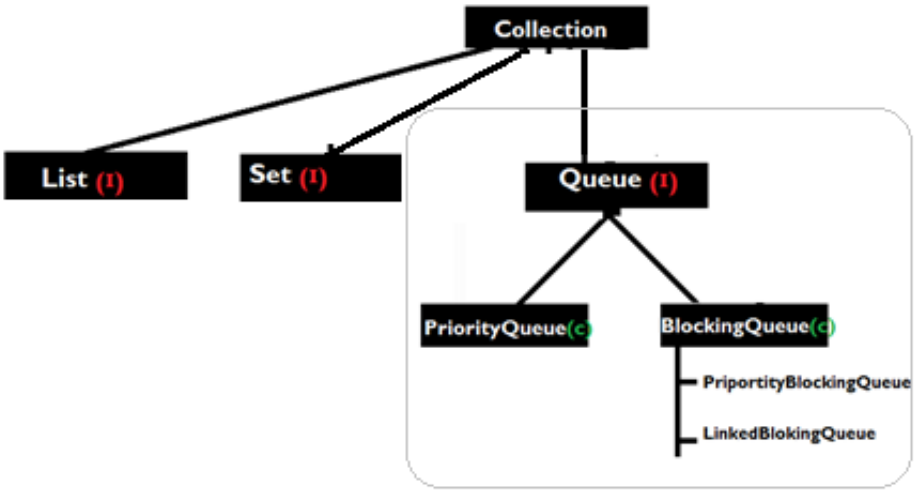
6. pollLast() - removes & returns Last elemet

7. desendingSet()- it returns NavigableSet in reverse order

```
public 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("All \t:" + t);
		System.out.println("descendingSet \t:" + t.descendingSet());
		System.out.println("floor \t:" + t.floor(3000));
		System.out.println("lower \t:" + t.lower(3000));
		System.out.println("ceiling\t:" + t.ceiling(3000));
		System.out.println("higher \t:" + t.higher(3000));
		System.out.println("pollFirst\t:" + t.pollFirst());
		System.out.println("pollLast\t:" + t.pollLast());
		System.out.println("After POLL \t:" + t);
	}
}
```

```
All     :[1000, 2000, 3000, 4000, 5000]
descendingSet   :[5000, 4000, 3000, 2000, 1000]
floor   :3000
lower   :2000
ceiling :3000
higher  :4000
pollFirst       :1000
pollLast:5000
After POLL      :[2000, 3000, 4000]
```

9.6 java.util.Queue (Interface 1.5 Version enhancements)



If we want to represent a group of individual objects prior to processing then we should go for Queue.for example before sending SMS message, all mobile numbers we have to store in some

datastricture.in which order we added mobile numbers in the same order only message should be delivered. For this FIFO (fisrt in first out) requirement QUEUE is the best choice

Usually QUEUE follows FIFO order, but based on our requirement we can implement our own priority order also (**PriortityQueue**)

From 1.5 Version onowards LinkedList class also implements QUEUE interface.LinkedList based implementation of queue always follows FIFO order

Queue Interface specific methods

1. Boolean offer (Object o)

To add an Object into the Queue

2. Object peek ()

To return HEAD elemenet of the Queue.if queue is empty it returns null

3. Object element ()

To return HEAD elemenet of the Queue.if queue is empty it throws RE: NoSuchElementException

4. Object poll ()

To remove &return HEAD elemenet of the Queue.if queue is empty it returns null

5. Object remove ()

To remove & return HEAD elemenet of the Queue.if queue is empty it throws RE: NoSuchElementException

9.6.1 PriortityQueue

- If we want to represent a group of individual objects prior to processing according to some priority then we should go for PriorityQueue
- The Priorty can either Def.Natural Sorting order or customized Sorting order defined by comparator
- **Insertion Order is NOT** preserved & it is based on some priorty
- **DUPLICATE** objects are **NOT** allowed
- If we are depending on Def.Natural Sorting order compulsory Objects should be Homogenious & comparable otherwise we will get **RE :ClassCastException**
- If we are defining our own Sorting by comaparator then objects need NOT be Homogenious & comparable.
- **Null is NOT** allowed even as the 1st element

1. PriorityQueue q = new PriorityQueue () **//11 capacity, Def. fill ratio = 0.75**

Creates an empty Queue Object with def. initial capacity, def. fill ratio 0.75 & Def.Sorting order

2. PriorityQueue q = new PriorityQueue (int intialcapacity) // Def. fill ratio = 0.75

3. PriorityQueue q = new PriorityQueue (int intialcapacity, Comparator c)

4. PriorityQueue q = new PriorityQueue (Collection c)

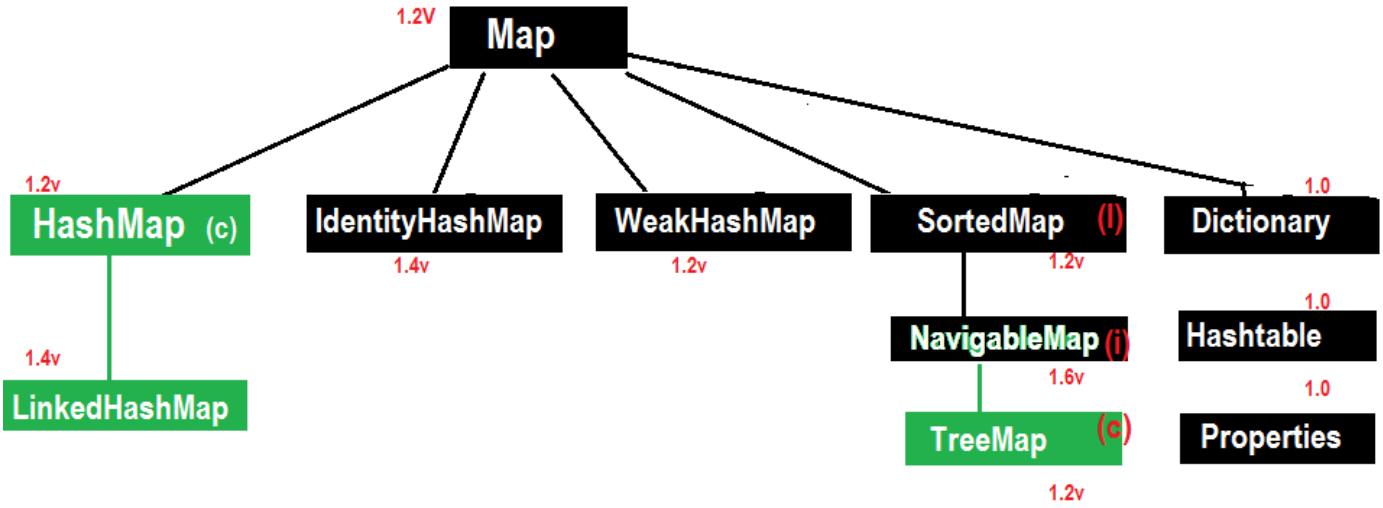
5. PriorityQueue q = new PriorityQueue (SortedSet s)

```
public class PriorityQDemo {
	public static void main(String[] args) {
		PriorityQueue q = new PriorityQueue();
		System.out.println(q.peek()); //null
	//System.out.println(q.element());//java.util.NoSuchElementException
		for (int i = 1; i <= 10; i++) {
			q.offer(i);
		}
		System.out.println(q);		//[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
		System.out.println(q.poll());// 1
		System.out.println(q);		//[2, 4, 3, 8, 5, 6, 7, 10, 9]
	}
}
```

```
null
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
1
[2, 4, 3, 8, 5, 6, 7, 10, 9]		//see Order Changed
```

Some platforms won't provide proper support for PriorityQueues

9.7 java.util.Map



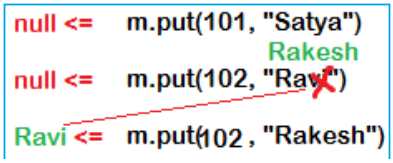
- Map is **not Child inteface of collection.**
- If we want to represent group of obejcts as **<Key, Value>** pairs then we use Map.

Key	Value
101	Satya
102	Ravi
103	Ravi
104	Rakesh

- Key, Value are Objects Only
- No Duplicates Key's are Allowed
- each <Key,value > pair is called 'Entry'

Methods

1. Object put(Object Key, Object value)



To add one <key, value> pair to the Map.

if the <key> is already present then oldvalue will be replaced with new Value & returns old value

2. Object putAll(Map m)

3. Void putAll(Map m)

4. Object get(Object key)

5. Object remove(Obejct key)

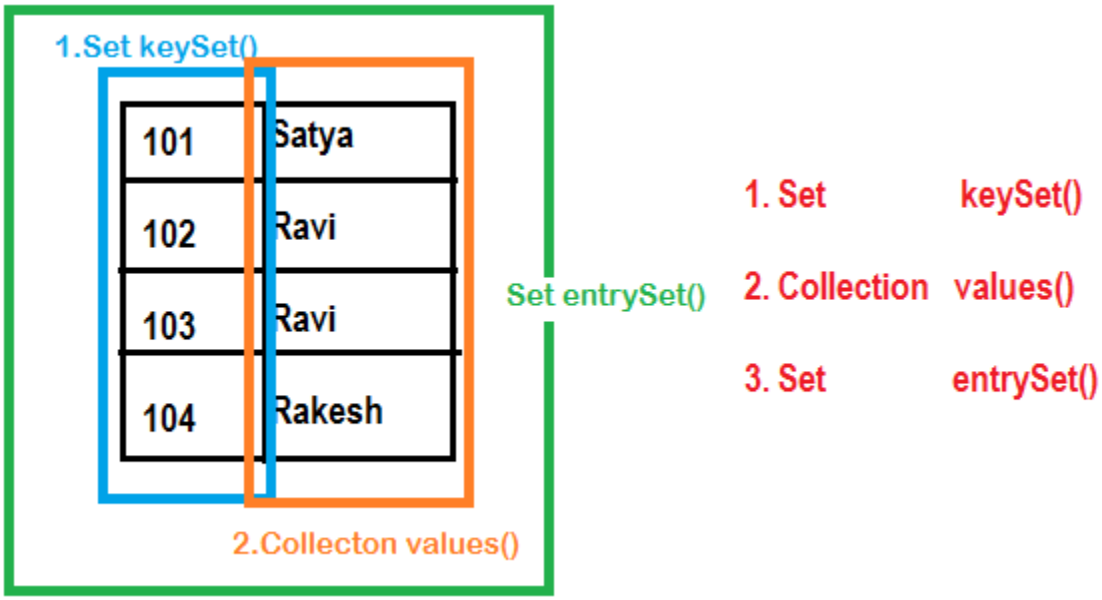
6. boolean containsKey(Object Key)

7. boolean containsValue(Object value)

8. boolean isEmpty()

9. int size()

10. void clear



9.7.1 Entry Interface

```
public interface Map {
                '
                '
        interface Entry{
Object      setValue(Object newVal)
Object      getKey();
Object      getValue();

        }
 }
```

9.7.2 HashMap

- The underlying datastructure is **Hashtable**
- Insertion order is not preserved & it is based on Hashcode of <Keys>
- Duplicate keys are NOT allowed, but values can be duplicated.
- Hetrogenious objects are allowed for both Key & Value
- **null is allowed for key(only once)**
- **null is allowed for Values(any no. of times)**
- HashMap implemets sericalizable & Clonable inferfaces but not RandomAccess
- HashMap is the best choice for **Seraching** opertions

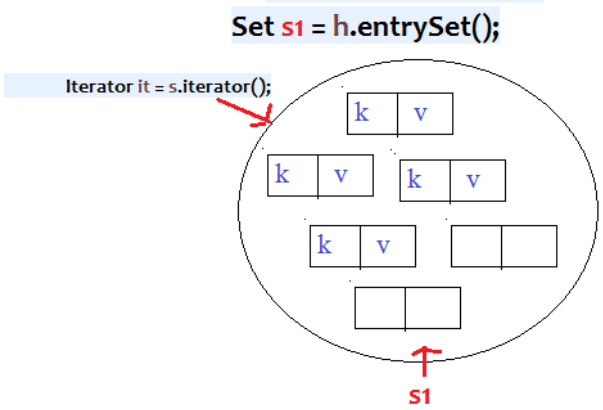
1. HashMap h = new HashMap () **//16 capacity, Def. fill ratio = 0.75**

Creates an empty Object with def. initial capacity 16 & def. fill ratio 0.75

2. HashMap h = new HashMap (int intialcapacity) **// Def. fill ratio = 0.75**

3. HashMap h = new HashMap (int intialcapacity, float fillRatio)

4. HashMap h = new HashMap (Map m)




```
public class HashMapDemo {
	public static void main(String[] args) {
		HashMap h = new HashMap();
		h.put("one", "Satya");
		h.put("two", "Ravi");
		h.put("three", "Rakesh");
		h.put("four", "Surya");
		System.out.println(h);// No Insertion Order

	System.out.println("adding exsting key:" + h.put("two", "Madhu"));
		System.out.println("All keys : " + h.keySet());
		System.out.println("All Values : " + h.values());
		System.out.println("Both Key-Values\n---------");

		Set s = h.entrySet();
		Iterator it = s.iterator();
		while (it.hasNext()) {
			Map.Entry m = (Map.Entry) it.next();
			System.out.println(m.getKey() + "\t : " + m.getValue());
		}
	}
}
```

```
{four=Surya, one=Satya, two=Ravi, three=Rakesh}
adding exsting key:Ravi
All keys : [four, one, two, three]
All Values : [Surya, Satya, Madhu, Rakesh]
Both Key-Values
---------
four	: Surya
one	: Satya
two	: Madhu
three	: Rakesh
```

HashMap	Hashtable
HashMap is non synchronized Performance is high because no threds waiting Nulls allowed for both <Key & Value> Introduced in 1.2 version	HashMap is synchronized Performance is Low because threads may wait Null is NOT allowed for both <Key & Value> Introduced in 1.0 version(Legacy)

By default HashMap is non-synchronized but we can get Sychronized version of HashMap by usinh synchronizedMap () of collections class

HashMap m = new HashMap()

Map m1 = Collections.synchronizedMap(m)

9.7.3 LinkedHashMap

- It is the child class of HashMap
- It is exactly same as HashMap (including methods&constrcutors) except following diffrences

HashMap	LinkedHashMap
Underlying D.S is Hashtable Insertion order is NOT preserved & it is based on hascode of KEYS Introduced in 1.2 version	Underlying D.S is Hashtable +LinkedList Insertion order is preserved Introduced in 1.4 version

```
public class LinkedHashMapDemo {
	public static void main(String[] args) {
		LinkedHashMap h = new LinkedHashMap();
		h.put("one", "Satya");
		h.put("two", "Ravi");
		h.put("three", "Rakesh");
		h.put("four", "Surya");
		System.out.println(h);//  Insertion Order Preserved

	System.out.println("adding exsting key:" + h.put("two", "Madhu"));
		System.out.println("All keys : " + h.keySet());
		System.out.println("All Values : " + h.values());
		System.out.println("Both Key-Values\n---------");
```

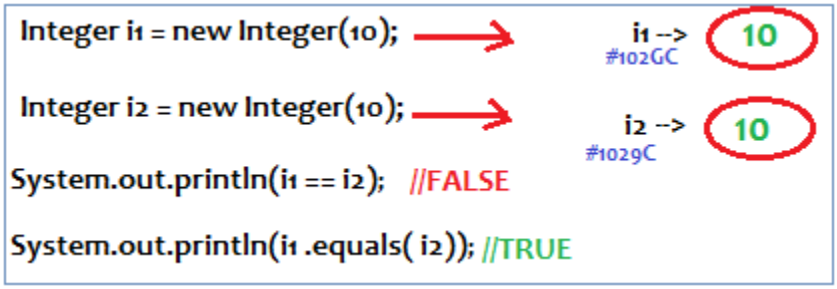
```
            Set s = h.entrySet();
            Iterator it = s.iterator();
            while (it.hasNext()) {
                  Map.Entry m = (Map.Entry) it.next();
                  System.out.println(m.getKey() + "\t : " + m.getValue());
            }
      }
}
```

```
{one=Satya, two=Ravi, three=Rakesh, four=Surya}
adding exsting key:Ravi
All keys : [one, two, three, four]
All Values : [Satya, Madhu, Rakesh, Surya]
Both Key-Values
---------
one       : Satya
two       : Madhu
three     : Rakesh
four      : Surya
```

LinkedHashSet & LinkedHashMap are commonly used for developing CACHE BASED APPPLICATIONS.

== is **for reference comparision** or **address comparision**

.equals () is for **content comparision**



9.7.4 IdentityHashMap

It is exactly same as HashMap (including methods&constructors) except following diffrences

In the case of Normal HashMap JVM will use **.equals()** method to identify Duplicate keys, which is ment for Content comparision

But, In the case of IdentityHashMap JVM will use **==** operator to identify Duplicate keys, which is ment **for reference comparision** or **address comparision**

HashMap Example	IdentityHashMap Example

```
public class IdentityHashMapDemo {
 public static void main(String[] a)
{
  HashMap m = new HashMap();
  m.put(new Integer(10), "Satya");
  m.put(new Integer(10), "Surya");
  System.out.println(m);
// {10=Surya}
}
}
```

{10=Surya}

```
public class IdentityHashMapDemo {
public static void main(String[] args)
{
IdentityHashMap  m = new IdentityHashMap();
  m.put(new Integer(10), "Satya");
  m.put(new Integer(10), "Surya");
  System.out.println(m);
// {10=Satya, 10=Surya}
      }
}
```

{10=Satya, 10=Surya}

9.7.5 WeakHashMap

It is exactly same as HashMap except following difference

In the case of HashMap eventhough Object doesn't have any reference it is **NOT elegible** for `gc()` if it is associated with HashMap that is HashMap dominates Garbage collector

But the case of WeakHashMap if Object doesn't have any references it is **elegible** for `gc()` eenthough Object associated with WeakHashMap that is Garbage collector dominates WeakHashMap

This Temp class is common for HashMap & WeakHapMap Demo's

```
class Temp {
	@Override
	public String toString() {
		return "Temp";
	}
	@Override
	protected void finalize() throws Throwable {
		System.out.println("Finalize Called");
	}
}
```

HashMap Demo with GC

```
public class HashMapdemo {
	public static void main(String[] args) throws InterruptedException {
		HashMap m = new HashMap();
		Temp t = new Temp();
		m.put(t, "Satya");
		System.out.println(m);
		t=null;
		System.gc();
		Thread.sleep(5000);
//main Thread Sleeping for 5 seconds
//Garbage collector takes control for 5 seconds
		System.out.println(m);
	}
}
```

```
{Temp=Satya}
{Temp=Satya}
```

In the above example Temp object is not eligible for gc() because it is associated with HashMap.in this case out put is `{Temp=Satya} {Temp=Satya}`

WeakHashMap Demo with GC

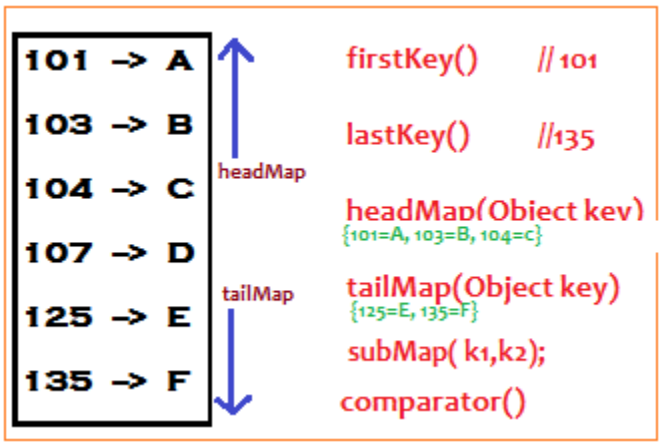
```
public class WeakHashMapdemo {
	public static void main(String[] args) throws InterruptedException {
		WeakHashMap m = new WeakHashMap();
		Temp t = new Temp();
		m.put(t, "Satya");
		System.out.println(m);
		t=null;
		System.gc();
		Thread.sleep(5000);
		System.out.println(m);
	}
}
```

```
{Temp=Satya}
Finalize Called
{}
```

In the above example Temp object is eligible for gc() because it is associated with HashMap.in this case out put is `{Temp=Satya} Finalize Called {}`

9.8 java.util.SortedMap

If we want to represt a group of objects as group of **<key,value>** pairs according to some sorting order of keys then we should go for SortedMap. Sorting is based on the Key but bot based on value



1. Object firstKey()

2. Object lastKey()

3. SortedMap headMap(Object key);

4. SortedMap tailMap(Object key);

5. SortedMap subMap(Obj k1, Obj k2);

9.8.1 TreeMap

- Underlying D.S is RED-BLACK TREE
- Insertion order is NOT preserved & it is based on some sorting order of KEYS

- DUPLICATE keys are NOT allowed, but values can be dublicated

If we are depending on default natural sorting Order then KEYS should be Homogenious & Comparable otherwise we will get Runtime exception saying **ClassCastException**

If we are defining our own sorting by comparator then KEYS should need not be Homogenious & Comparable.we can take Hetrogenious & non comaparable Objects also.

Null Accepatance

- For **EMPTY** TreeMap → **null key allowed as the only 1st Entry(up to 1.6V only)**
  If we try to enter 2nd entry it will show NE.
- For **NON-EMPTY** TreeMap → null **key not allowed.NullPointerException**

Constrcutors

1. TreeMap h = new TreeMap () **//Default. SortingOrder**

Creates an Empty TreeMap Object, all the elements inserted according to Default Natural SortingOrder

2. TreeMap h = new TreeMap (Comparator c) **//Customized. SortingOrder**

Creates an Empty TreeMap Object, all the elements inserted according to Customized Natural SortingOrder

3. TreeMap h = new TreeMap (Map c)

4. TreeMap h = new TreeMap (SortedMap s)

```
public class TreeMapDemo {
public static void main(String[] args) {
	TreeMap  t = new TreeMap();
	t.put(101, "A");
	t.put(104, "D");
	t.put(102, "B");
	t.put(103, "C");

//	t.put("sas", 101);
	//Exception in thread "main" java.lang.ClassCastException:
	//java.lang.Integer cannot be cast to java.lang.String

	System.out.println(t);
}
}
```

```
{101=A, 102=B, 103=C, 104=D}
```

9.8.2. NavigableMap

It is the Child interface of SortedMap & it defines servaral methods for Navigation purpose

1. floorKey(e) - it returns highest element which is <=e

2. lowerKey(e) - it returns highest element which is <e

3. ceilingKey(e) - it returns lowest element which is >=e

4. higherKey(e) - it returns lowest element which is >e

5. pollFisrtEntry() - removes & returns first elemet

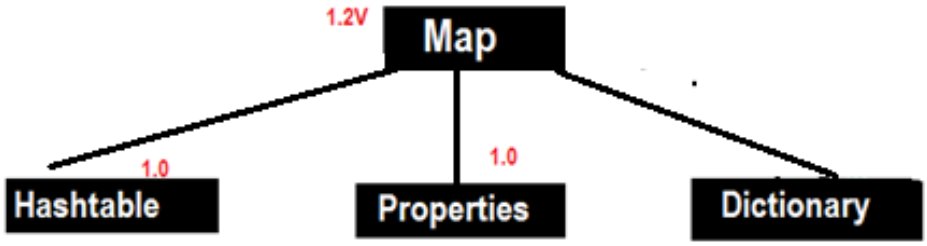
6. pollLastEntry() - removes & returns Last elemet

7. desendingMap() - it returns NavigableSet in reverse order

```
public class NavigableMapDemo {
	public static void main(String[] args) {
		TreeMap<String,String> t = new TreeMap<String,String>();
		t.put("1000","A");
		t.put("2000","B");
		t.put("3000","C");
		t.put("4000","D");
		t.put("5000","E");

		System.out.println("All \t:" + t);
		System.out.println("descendingMap \t:" + t.descendingMap());
		System.out.println("floor \t:" + t.floorKey("3000"));
		System.out.println("lower \t:" + t.lowerKey("3000"));
		System.out.println("ceiling\t:" + t.ceilingKey("3000"));
		System.out.println("higher \t:" + t.higherKey("3000"));
		System.out.println("pollFirst\t:" + t.pollFirstEntry());
		System.out.println("pollLast\t:" + t.pollLastEntry());
		System.out.println("After POLL \t:" + t);
	}
}
```

```
All     :{1000=A, 2000=B, 3000=C, 4000=D, 5000=E}
descendingMap   :{5000=E, 4000=D, 3000=C, 2000=B, 1000=A}
floor   :3000
lower   :2000
ceiling :3000
higher  :4000
pollFirst       :1000=A
```

9.9 Legacy Classes on Map



9.9.1 Hashtable

- Underlying D.S Hashtable for is Hashtable
- Insertion order is not preserved & it is based on Hashcode of keys
- DUPLICATE keys are NOT allowed & Values can be duplicated
- Hertogenious objects are allowed for both keys&values
- Null is NOT allowed for both key& value.otherwise we will get NullPointerException at runtime
- It implements Serializable, Clonable interfaces but not RandomAccess
- All methods are Synchronized, so Hashtable is Thread-Safe
- Hashtable is best choice for Search Operation

1. Hashtable h = new Hashtable () **//16 capacity, Def. fill ratio = 0.75**

Creates an empty Object with def. initial capacity **11** & def. fill ratio 0.75

2. Hashtable h = new Hashtable (int intialcapacity) **// Def. fill ratio = 0.75**

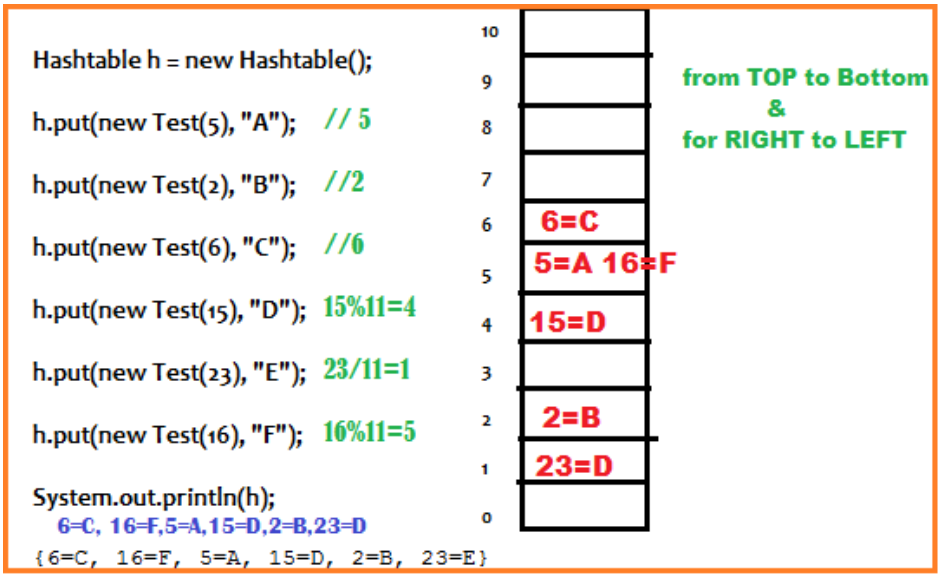
3. Hashtable h = new Hashtable (int intialcapacity, float fillRatio)

4. Hashtable h = new Hashtable (Map m)

```
class Test {
	int i;
	Test(int i) {
		this.i = i;
	}
	@Override
	public int hashCode() {
		return i;
	}
	@Override
	public String toString() {
		return i + "";
	}
}
public class HashtableDemo {
	public static void main(String[] args) {
		Hashtable h = new Hashtable();
```

```
                h.put(new Test(5), "A");
                h.put(new Test(2), "B");
                h.put(new Test(6), "C");
                h.put(new Test(15), "D");
                h.put(new Test(23), "E");
                h.put(new Test(16), "F");
                System.out.println(h);
        }
}
```

```
{6=C, 16=F, 5=A, 15=D, 2=B, 23=E}
```



9.9.2 Properties

In our if any thing which changes frequently like Database names, username, password etc we use properties file to store those & java programe used to read properties file

1. Properties p = new Properties ()

KEY & Values must be String type

Methods

1. **String getProperty(String name);**
2. **String setProperty(String name, value);**
3. **Enumaration propertyNames();**

4. **Void load(InputStream is)**
   Load properties from properties file into java properties Object

5. **Void store(OutputStream is, String commet)**

Store java properties Object into properties file

```
uname=satya             //abc.properties before
port=8080
```

```
public class PropertiesDemo {
public static void main(String[] args) throws  Exception {
	Properties p = new Properties();
	FileInputStream fis = new FileInputStream("abc.properties");
	p.load(fis);
	System.out.println(p);
	System.out.println("Uname : "+p.getProperty("uname"));

	p.setProperty("port", "8080");
	FileOutputStream fos = new FileOutputStream("abc.properties");
	p.store(fos, "Port Number comment added");
}
}
```

```
#Port Number comment added              //abc.properties After
#Mon Sep 12 20:38:33 IST 2016
uname=satya
port=8080
pwd=smlcodes
```

9.10 Collections utility class

Collections class defines servaral utility methods for collection Objects like Sorting, Searching, Reversing , synchronization etc

9.10.1 Sorting

1. Public static List sort (List l)

To sort Based on **Def.Natural Sorting order**.in this case List should contain Homogenious & Comarable objects otherwise we will get **RE: ClassCastException.**List Shoul not contain NULL otherwise we will get **NullPointerException**

2. Public static List sort (List l, Comparable c)

To sort Based on **Costomized Sorting order**.List may contain NULL no problem

9.10.2 Searching

1. Public static int binarySearch (List l, Object target)

If the List is sorted based on **Def.Natural Sorting order,** then we have to use this method

2. Public static int binarySearch (List l, Object target, Comparator c)

If the List is sorted based on **Customized Sorting order,** then we have to use this method

Successful search returns Index, UnSuccessful search returns InsertionPoint

- **InsertionPoint**: is the location where we can place Target element in the SortedList
- Before calling **binarySearch() List Should be Sorted.Otherwise we will get Unpredectable Results**
- If the List is sored according to Comparator then at the time of search operation also we have to pass same comparator Object **Otherwise we will get Unpredectable Results**

9.10.3 Reversing

Public static void reverse (List l)

To reverse order of **elemets of List**. Here we will pass List Object

Public static void reverseOrder (Comapartor l)

To reverse order of **Comparator**. Here we will pass Comparator Object

9.10.3 Synchronizing

- **static Collection synchronizedCollection(Collection c)**
- **static List synchronizedList(List list)**
- **static Map synchronizedMap(Map m)**
- **static Set synchronizedSet(Set s)**
- **static SortedMap synchronizedSortedMap(SortedMap m)**
- **static SortedSet synchronizedSortedSet(SortedSet s)**

9.11 Arrays utility class (java.util.Arrays)

Arrays utility class (**java.util.Arrays**) defines servaral utility methods for Primitive arrys & Object arrays like Sorting, Searching, and List Converting etc

9.11.1 Sorting

1. Public static void sort (PrimitiveArray [] a)

2. Public static void sort (Object [] a)

3. Public static void sort (Object [] a, Comparator c)

9.11.2 Searching

1. Public static int binarySearch (Primitive[] p, Primitive Key)

2. Public static int binarySearch (Object[] o, Object Key)

3. Public static int binarySearch (Object[] o, Object Key, Comparator c)

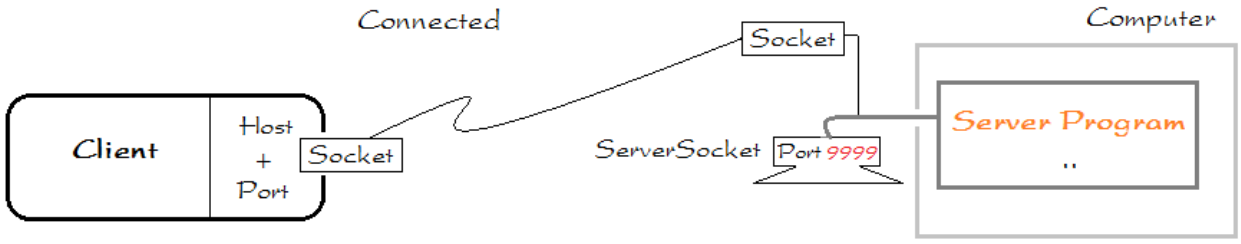
9.11.3 ArrayToList Convering

Public static List asList (a[])

Collections Rough work Area

X. Java Networking (java.net. *)

While working with Network concepts, there is always exist **Client & Server**. Network programming is nothing but communication between clinet& Server for sending/receving data



A socket is an endpoint between two way communications.

10.1 Communcating with Internal Applications

We have following classes to deal with Socket Programming in java. Java.net.* is root packge of Socket Programming

1. **java.net.Socket**

   This class used to implement **client side Socket**

Method	Description
1) public InputStream **getInputStream()**	Is used to Read the data from Scoket
2)public OutputStream **getOutputStream()**	Is used to write the data to Scoket
3) public synchronized void **close()**	closes this socket

2. **java.net.ServerSocket**

   This class used to implement **Server side Socket**

Method	Description
1) public Socket **accept()**	Establish a connection between server and client.
2) public synchronized void **close()**	Closes the server socket.

Steps to write SERVER program

1. **Create a Server using ServerSocket(int port)** which is running on port : 9999 for example
2. Accept clients by calling `ss.accept();`
3. Read data from Cilent get using `s.getInputStream());`
4. Close connections

```
public class Server {
	public static void main(String[] args) throws Exception {
		ServerSocket ss=new ServerSocket(9999);
		System.out.println("Server Started ......");
		Socket s=ss.accept();//waiting for client
		DataInputStream dis=new DataInputStream(s.getInputStream());
		String  str=(String)dis.readUTF();
		System.out.println("Recived : "+str);
		ss.close();
	}
}
```

Steps to write CLIENT program

1. **Connect with Server using Socket(int port)** which is running on port : 9999 for example
2. Write & send data to Server by using `s.getOutputStream()`
3. Close connections

```
public class Client {
	public static void main(String[] args) throws Exception {
		Socket s=new Socket("localhost",9999);
		DataOutputStream dout=new DataOutputStream(s.getOutputStream());
		dout.writeUTF("SmlCodes.com");
		dout.flush();
		dout.close();
		s.close();
	}
}
```

```
Console
<terminated> Server [Java Application
Server Started ......
Recived : SmlCodes.com
```

The above classes are Not recommened to Connect Worldwideweb (Ex. Google.com). to dealing with www we have URL classes

10.2 Communcating with Worldwide Web

1. Java.net.URL

This class used to get URL information like hostname, port number, etc

Method	Description
public String **getProtocol()**	It returns the protocol of the URL.
public String **getHost()**	It returns the host name of the URL.
public String **getPort()**	It returns the Port Number of the URL.
public String **getFile()**	It returns the file name of the URL.
public URLConnection **openConnection()**	it returns the instance of URLConnection

2. Java.net.URLConnection

Used for performing read & write operations on resource referenced by the URL

- **Object getContent()**
- **InputStream getInputStream()**
- **OutputStream getOutputStream()**

3. Java.net.HttpURLConnection

HttpURLConnection class is same as **URLConnection** but is specific to HTTP protocol only.

4. Java.net.InetAddress

Java InetAddress class represents an IP address. The java.net.InetAddress class provides methods to get the IP of any host name.

Method	Description
public static InetAddress **getByName(String host)**	it returns the instance of InetAddress of host
public static InetAddress **getLocalHost()**	Get InetAdddress of local host
public String **getHostName()**	It returns the host name of the IP address.
public String **getHostAddress()**	it returns the IP address in string format.

```java
public class URLDemo {
	public static void main(String[] args) throws Exception {
		System.out.println("1. Java.net.URL \n -------");
		URL url = new URL("http://www.smlcodes.com/java");
		System.out.println("Protocol:" + url.getProtocol());
		System.out.println("Host\t:" + url.getHost());
		System.out.println("Port\t:" + url.getPort());
		System.out.println("File\t:" + url.getFile());

		System.out.println("\n2. Java.net.URLConnection\n -------");
		URLConnection con = url.openConnection();
		InputStream is = con.getInputStream();
		int i;
		while ((i = is.read()) != -1) {
			System.out.print((char)i);
		}
		is.close();

		System.out.println("\n3. Java.net.HttpURLConnection\n -------");
		HttpURLConnection http = (HttpURLConnection) url.openConnection();
		InputStream is2 = http.getInputStream();
		int j;
		while ((j = is2.read()) != -1) {
			System.out.print((char)j);
		}

		System.out.println("4. Java.net.InetAddress \n -------");
		InetAddress ip = InetAddress.getByName("localhost");
		System.out.println("IP ADDRESS : " + ip.getHostAddress());
		System.out.println("HOST NAME : " + ip.getHostName());
		System.out.println("LOCAL PC NAME ,IP : " + ip.getLocalHost());
	}
}
```

```
1. Java.net.URL
 -------
Protocol:http
Host    :www.smlcodes.com
Port    :-1
File    :/java

2. Java.net.URLConnection
 -------
<html> . . COMPLETE PAGE SOURCE </html>


3. Java.net.HttpURLConnection
 -------
<html> . . COMPLETE PAGE SOURCE </html>

4. Java.net.InetAddress
 -------
IP ADDRESS : 127.0.0.1
HOST NAME : localhost
LOCAL PC NAME : IPHYDPCM90480L/10.69.19.68
```

XI. Features 1.x to Till Now

JDK Alpha and Beta (1995)

JDK 1.0 (23rd Jan, 1996) -

JDK 1.1 (19th Feb, 1997)

- AWT event model
- Inner classes
- JavaBeans
- JDBC
- RMI,Reflection
- JIT (Just In Time) compiler for Windows

J2SE 1.2 (8th Dec, 1998) – Playground

- **strictfp** keyword
- Swing graphical API
- Sun's JVM was equipped with a JIT compiler for the first time
- Java plug-in
- **Collections framework**

J2SE 1.3 (8th May, 2000) - Kestrel

- HotSpot JVM
- Java Naming and Directory Interface (JNDI)
- Java Platform Debugger Architecture (JPDA)
- JavaSound
- Synthetic proxy classes

J2SE 1.4 (6th Feb, 2002) – Merlin

- **assert keyword**
- **Regular expressions**
- Exception chaining
- Internet Protocol version 6 (IPv6) support
- New I/O; NIO
- **Logging API**
- Image I/O API
- Integrated XML parser and XSLT processor (JAXP)

- Integrated security and cryptography extensions (JCE, JSSE, JAAS)
- Java Web Start
- Preferences API (java.util.prefs)

J2SE 5.0 (30th Sep, 2004) – Tiger

- **Generics**
- **Annotations**
- **Autoboxing/unboxing**
- **Enumerations**
- **Varargs**
- **Enhanced for each loop**
- **Static imports**
- **New concurrency utilities in java.util.concurrent**
- **Scanner class for parsing data from various input streams and buffers**.

Java SE 6 (11th Dec, 2006) – Mustang

- Scripting Language Support
- Performance improvements
- JAX-WS
- JDBC 4.0
- Java Compiler API
- JAXB 2.0 and StAX parser
- Pluggable annotations
- New GC algorithms

Java SE 7 (28th July, 2011) – Dolphin

- JVM support for dynamic languages
- Compressed 64-bit pointers
- **Strings in switch**
- Automatic resource management in try-statement
- The diamond operator
- Simplified varargs method declaration
- Binary integer literals
- Underscores in numeric literals
- Improved exception handling
- ForkJoin Framework
- NIO 2.0 having support for multiple file systems, file metadata and symbolic links
- WatchService
- Timsort is used to sort collections and arrays of objects instead of merge sort
- APIs for the graphics features

- Support for new network protocols, including SCTP and Sockets Direct Protocol

Java SE 8 (18th March, 2014) - Code name culture dropped

- **Lambda expression support in APIs**
- Functional interface and default methods
- Optionals
- Nashorn – JavaScript runtime which allows developers to embed JavaScript code within applications
- Annotation on Java Types
- Unsigned Integer Arithmetic
- Repeating annotations
- New Date and Time API
- Statically-linked JNI libraries
- Launch JavaFX applications from jar files
- Remove the permanent generation from GC

Java SE 9 Expected: September 22, 2016

- Support for multi-gigabyte heaps
- Better native code integration
- Self-tuning JVM
- Java Module System
- Money and Currency API
- jshell: The Java Shell
- Automatic parallelization

11.1 Assert keyword

Assert keyword is used to check the given statement is **TRUE or FALSE**

There are two types of using assert in our program

```
1. assert (boolean expression);                          //Simple assert

2. assert (boolean expression1) : (anytype expression2);  //Agumented assert
```

Assertion is disabled by default. To enable we have to use **java –ea classname** or **-enableassertions**

The main advantage of assert is for **DEBUGGING**. If we write s.o.p's for debugging after completion of code we have to munually remove the s.o.p's. But if we use assertions for debugging after completion of code we don't need to remove the code, just DISABLING assertion is enough

11.1.1 Simple assert

```
assert (boolean expression);
```

- Here the Expression Should be **Boolean** type.
- If expression is TRUE it wont return anything,
- Otherwise it will throws **Runtime Exception** : **java.lang.AssertionError: Not valid**

```
public class AssertDemo {
public static void main(String[] args) {
	int i = 100;
	assert (i>10);
	System.out.println(i); //100
}
}
```

If we give `int i = 10;` it will throws **java.lang.AssertionError: Not valid**

11.1.2 Agumented assert

```
2. assert (boolean expression1) : (anytype expression2);	//Agumented assert
```

Expression2→ is used to Disply some message along with Error Message

- Here the 1st Expression Should be **Boolean** type, 2nd Expression can be **Any type**
- If expression is TRUE it wont return anything,
- Otherwise it will throws **Runtime Exception** : **java.lang.AssertionError: expression2**

```
public class AssertDemo {
	public static void main(String[] args) {
		int i = 1;
		assert (i > 10) : "This is Anytype";
		System.out.println(i);
	}
}
```

```
Exception in thread "main" java.lang.AssertionError: This is Anytype
at features.AssertDemo.main(AssertDemo.java:6)
```

To ENABLE assertions we have to use **java –ea classname or –enableassertions**

To DISABLE assertions we have to use **java –da classname or –disableassertions**

To ENABLE assertions in **ECLIPSE** Rightclickon File → Run As → Run Configurations

→Apply→Save

11.2 Enhanced for each loop

- For-each loop introduced in **Java 1.5 version as an enhancement of traditional for-loop**
- Main advantage is we **don't need to write extra code** traverse over array / collections
- Mainly used for traverse on **Array Elements & Collection Elements**

```
for (Datatype temp_variable : Array/Collection Variable){}
```

11.2.1 for-each loop on Array Elements

```
public class Foreach {
	public static void main(String[] args) {
	int i[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
		for (int b : i) {
			System.out.print(b+", ");
		}
	}
}
```

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
```

11.2.2 for-each loop on Collection Elements

```
public class Foreach {
	public static void main(String[] args) {
		ArrayList<String> l = new ArrayList<String>();
		l.add("A");
		l.add("B");
		l.add("C");
		l.add("D");
		l.add("E");
		for (String s : l) {
			System.out.print(s + ",");
		}
	}
}
```

```
A, B, C, D, E
```

11.3 Var-args

Some times we don't know no.of arguments used in our method implementation.var...args are introduced in 1.5v to solve these type of situations

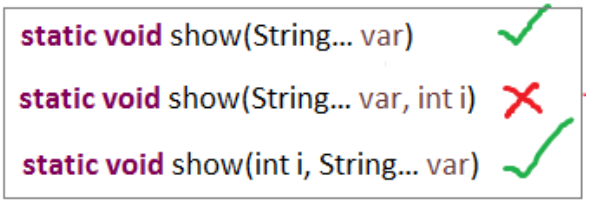
```
static returntype methodname(Datatype ... variblename)
```

```
public class Varargs {
	static void show(String… var) {
		System.out.println("Show() called");
	}
	public static void main(String[] args) {
		Varargs v = new Varargs();
		v.show();
		v.show("A");
		v.show("A", "B", "C");
	}
}
```

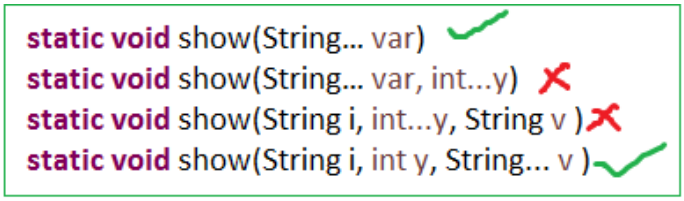
```
Show() called
Show() called
Show() called
```

Rules for using Var-args

1. Var-args must be as the last argument in method signature



2. Only one Var-arg is allowed per a Method



11.4 Static imports

From 1.5 version onwords we can access any static member of a class directly. There is no need to qualify it by the class name.

```
import static java.lang.System.*;

public class StaticImport {
	public static void main(String[] args) {
		out.println("Static Import");
	}
}
```

11.5 Enums

Enum in java is a data type that contains fixed set of constants.

- enums improves **type safety**
- easily used in **switch**
- enum can be traversed
- enum class is a just like normal class, we can write FIELDS, Constrcutors, Methods in enum class
- enum **CANNOT extend any class** because it internally extends Enum class
- enum may **implement many interfaces**

ALL fields in Enums by Default **PUBLIC STATIC FINAL**

1. Simple ENUM Example

```
enum Days {
	MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}

public class EnumDemo {
	public static void main(String[] args) {
		for (Days s : Days.values()) {
			System.out.print(s + ",");
		}
	}

}
```

```
MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY,
```

2. Enum with Values

To write Enum with values we have to follow below steps

- Enum values must be placed inside ( ) → ***MONDAY(1)***
- Take public instance variable to Store enum value → ***public int Enum_Value;***
- Write **private constructor** which can take enum value as argument→ **private Days(int Enu_Val)**

```
enum Days {
MONDAY(1), TUESDAY(2), WEDNESDAY(3), THURSDAY(4), FRIDAY(5), SATURDAY(6), SUNDAY(7);
	public int Enum_Value;
	private Days(int Enum_Value) {
		this.Enum_Value = Enum_Value;
	}
}
public class EnumDemo {
	public static void main(String[] args) {
		for (Days s : Days.values()) {
			System.out.println(s + ":"+s.Enum_Value);
		}
	}
}
```

```
MONDAY  :1,  TUESDAY:2,  WEDNESDAY:3,  THURSDAY  :4,  FRIDAY:5,  SATURDAY:6  SUNDAY:7
```

11.6 Annotations

Annotations in java are used to **provide additional information**, so it is an **alternative option for XML and java marker interfaces**

Java @Annotation is a represents **metadata**. It can be attached at the top of class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM

1. Built-In Annotations in Java

1) @Override

It will indicates that the Subclass is overriding Parent class method in its class

```
public class Test {
	@Override
	public String toString() {
		return "toString() Override";
	}
}
```

2) @SuppressWarnings

If we use **ArrayList** directly without using generic collection like **ArrayList<String>,** at compile time it shows warnings like

```
C:\Users\kaveti_S\Desktop\temp>javac Test.java
Note: Test.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

To avoid those type of warnings we can use **@SuppressWarnings.**It will Supress/reduces/Avoids those type of warings. In above it says Recomple with **unchecked** .So by adding **@SuppressWarnings("unchecked")** it wont shows any warings at compile time

```
public class Test {
	@SuppressWarnings("unchecked")
	public static void main(String args[]) {
		ArrayList list = new ArrayList();
		list.add("A");
		list.add("B");
		list.add("C");

		for (Object obj : list)
			System.out.println(obj);
	}
}
```

3) @Deprecated

@Deprecated annoation informs user that it may be removed in the future versions. So, it is better not to use such methods.

2. Custom Annotations

We can create our Own Annonations by following below Steps

- Add **@interface** before annotation name
- Every **method should return any one primitive** value (String,Int,etc)
- We can give default value to methods by using **default val**;
- Method should not throws any exceptions

We have 3 types of Annotations

1) **Marker Annotation**
2) **Single-valued Annotation**
3) **Multi-valued Annotation**

1. Marker Annotation: An annotation that has no method, is called marker annotation

```
@interface MyAnnotation{}
```

2. Single-Valued Annotation: An annotation that has one method is called single-value annotation

```
@interface MyAnnotation{
    int value() default 0;
}
```

3. Multi-Valued Annotation: An annotation that has more than one method

```
@interface MyAnnotation{
    int value1() default 1;
    String value2() default "";
    String value3() default "xyz";
}
```

For every annontation we can place **@Target, @Retention annotations** to specify where we are going to use our custom annotation

@Target annotation is used to specify at which type, the annotation is used

Element Types	Where the annotation can be applied
ElementType.TYPE	class, interface or enumeration
ElementType.FIELD	fields
ElementType.METHOD	methods
ElementType.CONSTRUCTOR	constructors
ElementType.LOCAL_VARIABLE	local variables
ElementType.ANNOTATION_TYPE	annotation type
ElementType.PARAMETER	parameter

@Retention annotation is used to specify to what level annotation will be available.

RetentionPolicy	Availability
RetentionPolicy.SOURCE	Refers to the source code, discarded during compilation. It will not be available in the compiled class.
RetentionPolicy.CLASS	Refers to the .class file, available to java compiler but not to JVM. It is included in the class file.
RetentionPolicy.RUNTIME	refers to the runtime, available to java compiler and JVM

Simple Custom annotation Example

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface MyAnnotation{
int value();
}
```

11.7 Generics

Generics are introduced in Java 1.5 Version to solve **Type-safity** & **Type-casting** problems

```
public class Student {
	public static void main(String[] args) {
		ArrayList l = new ArrayList(); //1
		l.add("Satya");
		String s = (String) l.get(0); //2
		System.out.println(s);
	}
}
```

- At **line 1.** ArrayList is **NOT generic** type – so we can add any type of elements results **Type-Safity**
- At **line 2.** It returns added String data as Object. So manullay we have to **Type-cast to String**

To resolve above problems Genrics are indrodued

```
public class Student {
	public static void main(String[] args) {
		ArrayList<String> l = new ArrayList<String>(); // 1
		l.add("Satya");
		String s = l.get(0); // 2
		System.out.println(s);
	}
}
```

Generics solves ClassCastException, beacause TYPE-CHECKING done at COMPILE-TIME itself

Generics can be used in following areas

- ***Class level***
- ***Interface level***
- ***Constructor level.***
- ***Method level***

Sample Generic class

```
class SmlGen<T> {
	T obj;

	void add(T obj) {
		this.obj = obj;
	}

	T get() {
		return obj;
	}
}
```

The **<T>** indicates that it can refer to any type (like String, Integer, and Student etc.). The type you specify for the class, will be used to store and retrieve the data

1. Type Parameter<T> Naming Conventions

In above <T> refers any type. Similary we have to follow below naming convensions to where to use which Naming conevsions. It improves readability. These are NOT compulsory but recommended to use

1. **T – Type - Any Type, can be use any level (Class, Interface, Methods...)**
2. **N – Number - Indicates it allows Number Types (int,long,float etec)**
3. **E – Element - Exclusivly used in Collection (ArrayList<E>)**
4. **K – Key - Map KEY area**
5. **V – Value - Map Value area**
6. **S,U,V etc. - 2nd, 3rd, 4th types**

1. Genrics at Class /Method /Constrcutor level

A generic class is defined with the following format:

```
class name<T1, T2, ..., Tn> {
---
 }
```

```
class SmlGen<T> {                       //1.Class Level
	T obj;
	public SmlGen() {
			}

	public SmlGen(T obj) {          //2.Constrcutor Level
		this.obj = obj;
	}

	void add(T obj) {
		this.obj = obj;
	}

	T get() {                       //3.Method Level
		return obj;
	}
}

public class Student {
	public static void main(String[] args) {
		SmlGen<String> s = new SmlGen<String>();
		s.add("Satya");
		System.out.println(s.get());
	}
}
```

2. Genrics at Interface level

A generic interface is same as gereric class.

```
public interface List<T> extends Collection<T> {
...
}
```

2. Wildcard in Generics

? Operator is used to represents wilcards in generics. We have to types of Wildcards

1. **Unbounded wildcards**
2. **Bounded wildcards**

1. Unbounded wildcards

Unbounded wildcard looks like `<?> -` means the generic can be any type.it is not bounded with anytype.

2. Bounded wildcards

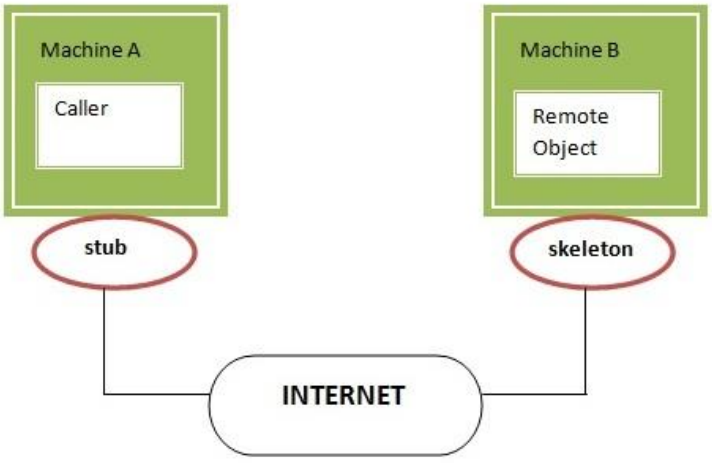
`<? extends T>` and `<? super T>` are examples of bounded wildcards

`<? extends T>` : means it can accept the **Child class Objects of the type<T>**

`<? super T>` : means it can accept the **Partent class Objects of the type<T>**

11.8 RMI

RMI used to invoke methods which is running on one JVM from another JVM



1. Stub

The stub is an object, acts as a gateway for the client side.if we invokes method on the stub object, it does the following tasks:

- It **initiates a connection with remote Virtual Machine (JVM),**
- It **writes and sends** (marshals) the parameters to the remote Virtual Machine (JVM),
- It waits for the result
- It **reads** (unmarshals) the **return value or exception**, and

2. Skeleton: The skeleton is an object, acts as a gateway for the server side object. All the incoming requests are routed through it. When the skeleton receives the incoming request, it does the following tasks:

- It reads the parameter for the remote method
- It invokes the method on the actual remote object, and
- It writes and transmits (marshals) the result to the caller.

Steps to Create Skeleton / Client

1. Create an **Interface by implementing Remote** interface with methods you want to share
2. Create a **Class** by exctending `UnicastRemoteObject` & also implement above methods
3. Create a **class to Share the Remote Class Object** over the Network

1. Interface by implementing Remote

```
public interface RemoteInterface extends Remote {
      public String show(String name);
}
```

2.Class by exctending UnicastRemoteObject

```
public class RemoteClass extends UnicastRemoteObject implements
RemoteInterface {
   protected RemoteClass() throws RemoteException {
         super();
   }
   @Override
   public String show(String name) {
         // TODO Auto-generated method stub
         return "Your Name Is : " + name;
   }
}
```

3. Create a class to Share the Remote Class Object over the Network

```
public class RemoteObject {
public static void main(String[] args) throws RemoteException,
MalformedURLException, AlreadyBoundException {
   RemoteInterface obj = new RemoteClass();
   Naming.rebind("obj",  obj);
 }
}
```

4.stub class - Client

```
public class Client {
      public static void main(String args[]) throws Exception
RemoteInterface st=(RemoteInterface) Naming.lookup("rmi://"+args[0]+ "/obj");
            System.out.println(st.show("Satya"));
      }
}
```

11.9 RegExp

Java.util.regex or Regular Expression is an API to **define pattern for searching or manipulating strings.** It is widely used to define constraint on strings such as **password and email validation.**

It provides following classes are widely used in java regular expression.

- Pattern class -it represents the Complied pattern
- Matcher class -used for performing matching operations on complied pattern
- PatternSyntaxException - checks syntax error in a regular expression pattern.

1. Pattern class it represents the Complied pattern

Method	Description
static Pattern **compile**(String regex)	Compiles the given regex and return the instance of pattern.
Matcher **matcher**(CharSequence input)	Retuns charsequence to be comapir with patten
static boolean **matches(String regex, String CharSequence)**	Direcly we can compare Expression with Sequence
String pattern()	returns the regex pattern.

2. Matcher class -used for performing matching operations on complied pattern

Method	Description
boolean **matches()**	Test whether the regular expression matches the pattern.
boolean find()	Finds the next expression that matches the pattern.
boolean find(int start)	Finds the next expression that matches the pattern from the given start number.
String group()	Returns the matched subsequence.
int start()	Returns the starting index of the matched subsequence.
int end()	Returns the ending index of the matched subsequence.
int groupCount()	Returns the total number of the matched subsequence.

```
public class REDemo {
   public static void main(String[] args) {
       Pattern p = Pattern.compile(".a");// only 2 char end with a
       Matcher m = p.matcher("sa");
       boolean b1 = m.matches();
       System.out.println(b1);//TRUE

       boolean b2 = Pattern.matches("s.", "sa"); //only 2 char Start with s
       System.out.println(b2); //TRUE
       }
}
```

1. Regex Character classes

Character Class	Description
[abc]	a, b, or c (simple class)
[^abc]	Any character except a, b, or c (negation)
[a-zA-Z]	a through z or A through Z, inclusive (range)
[a-d[m-p]]	a through d, or m through p: [a-dm-p] (union)
[a-z&&[def]]	d, e, or f (intersection)
[a-z&&[^bc]]	a through z, except for b and c: [ad-z] (subtraction)
[a-z&&[^m-p]]	a through z, and not m through p: [a-lq-z](subtraction)

2. Regex Quantifiers

The quantifiers specify the number of occurrences of a character.

Regex	Description
X?	X occurs once or not at all
X+	X occurs once or more times
X*	X occurs zero or more times
X{n}	X occurs n times only
X{n,}	X occurs n or more times
X{y,z}	X occurs at least y times but less than z times

3. Regex Metacharacters

The regular expression metacharacters work as a short codes.

Regex	Description
. (dot)	Any character (may or may not match terminator)
\d	Any digits, short of [0-9]
\D	Any non-digit, short for [^0-9]
\s	Any whitespace character, short for [\t\n\x0B\f\r]
\S	Any non-whitespace character, short for [^\s]
\w	Any word character, short for [a-zA-Z_0-9]
\W	Any non-word character, short for [^\w]
\b	A word boundary
\B	A non word boundary

```
public class REDemo {
	public static void main(String[] args) {
S.o.p("1.Regex Character classes\n----------------");
S.o.p(Pattern.matches("[amn]", "abcd"));//false (not a or m or n)
S.o.p(Pattern.matches("[amn]", "a"));//true (among a or m or n)
S.o.p(Pattern.matches("[amn]", "ammmna"));//false(m &a morethan once)

S.o.p("\n2.Regex Quantifiers\n----------------");
S.o.p("? quantifier ....");
S.o.p(Pattern.matches("[amn]?", "a"));//true (a or m or n comes one time)
S.o.p(Pattern.matches("[amn]?", "aaa"));//false (a comes more than one time)
S.o.p(Pattern.matches("[amn]?", "aammmnn"));//false (a m and n comes more
than one time)

S.o.p("+ quantifier ....");
S.o.p(Pattern.matches("[amn]+", "a"));//true (a or m or n once or more times)
S.o.p(Pattern.matches("[amn]+", "aaa"));//true (a comes more than one time)


S.o.p("\n3.Regex Metacharacters\n----------------\n");
S.o.p(Pattern.matches("\\d", "abc"));//false (non-digit)
S.o.p(Pattern.matches("\\d", "1"));//true (digit and comes once)
S.o.p(Pattern.matches("\\d", "4443"));//false (digit but comes more than 1)
	}
}
```

11.10 Logging API

In common we use **System.out.println ()** statements for DEBUGGING. But these are printed at console and they will lost after closing the Console.so these results are not savable

To overcome these problems apache released **Log4j.** With Log4j we can store the flow details of our Java/J2EE in a file or databases

We have mainly 3 components to work with Log4j

- **Logger class** **-for printing LOG messages**
- **Appender interface** **-to store messages in Files/Databases**
- **Layout** **-which Fomate the message should Save(HTML,Text,etc)**

1. Logger class

- Logger is a class, in org.apache.log4j.*
- We need to create **Logger object one per java class**,it will enables Log4j in our java class
- Logger methods are used to generate log statements in a java class instead of sopls
- So in order to get an object of Logger class, we need to call a **static factory method**

```
static Logger log = Logger.getLogger(YourClassName.class.getName())
```

We have following methods to print debugging statements on Logger

1. **log.debug (" ")**
2. **log.info ("")**
3. **log.warn ("")**
4. **log.error ("")**
5. **log.fatal ("")**

Here **human identification purpose names are different**, all 5 methods will print one text message only.

Priority Order: debug < info < warn < error < fatal

2. Appender interface

Appender job is to write the messages into the **external file or database or SMTP**In log4j we have different Appender implementation classes

- **ConsoleAppende**r [Writing into console]
- **FileAppender** [writing into a file]
- **JDBCAppender** [For Databases]
- **SMTPAppender** [sent logs via Mails]
- **SocketAppender** [For remote storage]

3. Layout

This component specifies the format in which the log **statements are written into the destination** by the appender

- **SimpleLayout**
- **PatternLayout**
- **HTMLLayout**
- **XMLLayout**

Simple Hello world

```
public class LogDemo {
public static void main(String[] args) {
    Logger logger = Logger.getLogger(LogDemo.class.getName());
    Layout layout = new SimpleLayout();
    Appender a = new ConsoleAppender(layout);
    logger.addAppender(a);

    logger.debug("Debug Message");
    logger.info("Info Message");
    logger.warn("Warning Message");
    logger.error("Error Message");
    logger.fatal("Fatal Message");
}
}
```

In above Example we used **Layout, Appenders** programmatically which is NOT RECOMMENDED.we have to use **log4j.properties** to configure those.

Log4j.properties Structure

log4j.rootLogger=DEBUG, CONSOLE, LOGFILE

log4j.appender.CONSOLE=

log4j.appender.CONSOLE.layout=

log4j.appender.CONSOLE.layout.ConversionPattern=

log4j.appender.LOGFILE=

log4j.appender.LOGFILE.File=

log4j.appender.LOGFILE.MaxFileSize=

log4j.appender.LOGFILE.layout=

log4j.appender.LOGFILE.layout.ConversionPattern=

If we use .properties file, we **no need to import any related classes** into our java class

if we wrote **log4j.rootLogger = WARN,abc** then it will prints the messages in l.warn(), l.error(), l.fatal() and ignores l.debug(), l.info(). Means >Warn level only it prints

Make sure LOG file should be placed in /src Folder

Example programe to Store LOG's in a FILE

```
public class LogDemo {
	static Logger logger = Logger.getLogger(LogDemo.class.getName());
	public static void main(String[] args) {
		logger.debug("Debug Message");
		logger.info("Info Message");
		logger.warn("Warning Message");
		logger.error("Error Message");
		logger.fatal("Fatal Message");
	}
}
```

Log4j.properties

```
log4j.rootLogger = DEBUG,abc
log4j.appender.abc = org.apache.log4j.FileAppender
log4j.appender.abc.file = logfile.log
log4j.appender.abc.layout = org.apache.log4j.SimpleLayout
```

logfile.log

```
DEBUG - Debug Message
INFO - Info Message
WARN - Warning Message
ERROR - Error Message
FATAL - Fatal Message
```

The above example only saves log's to file. You can't see logs on console .if want both use below. Use same java program, but change log4j.properties file

log4j.properties in Real world Applications

```
log4j.rootLogger=DEBUG,CONSOLE,LOGFILE
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
log4j.appender.CONSOLE.layout.ConversionPattern=%-4r [%t] %-5p %c %x – %m%n
log4j.appender.LOGFILE=org.apache.log4j.RollingFileAppender
log4j.appender.LOGFILE.File=logfile.log
log4j.appender.LOGFILE.MaxFileSize=1kb
log4j.appender.LOGFILE.layout=org.apache.log4j.PatternLayout
log4j.appender.LOGFILE.layout.ConversionPattern=[%t] %-5p %c %d{dd/MM/yyyy HH:mm:ss} – %m%n
```

```
[main] DEBUG log.LogDemo 15/09/2016 19:23:38 â?? Debug Message
[main] INFO  log.LogDemo 15/09/2016 19:23:38 â?? Info Message
[main] WARN  log.LogDemo 15/09/2016 19:23:38 â?? Warning Message
[main] ERROR log.LogDemo 15/09/2016 19:23:38 â?? Error Message
[main] FATAL log.LogDemo 15/09/2016 19:23:38 â?? Fatal Message
```

XII Design Pattrens

Design patterns are set rules provided by industry experts to avoid recurringly occurring software development problems

Christopher Alexander was the first person who invented all the above Design Patterns in 1977.in 1995 four persons named as Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides wrote a book which covers 23 Design patterns.now we are covering those Design patterns

We have 3 types of Design Patterns in java

1. Creational Design Pattern -Deals with the way of creating objects
2. Structural design patterns - simplifies the structure by identifying the relationships
3. Behavioral Design Patterns - the interaction and responsibility of objects

1. Factory method

Factory method is a method whose return type must be similar to its class name

```
public class Factory {
	public static Factory getFactory() {
		return new Factory();
	}
	public static void main(String[] args) {
		Factory f = Factory.getFactory();
	}
}
```

Factory D.P Says define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate

2. Abstract Factory Pattern

Abstract Factory Pattern says that just **define an interface or abstract class for creating families of related (or dependent) objects but without specifying their concrete sub-classes**.

That means Abstract Factory lets a class returns a factory of classes. So, this is the reason that Abstract Factory Pattern is one level higher than the Factory Pattern.An Abstract Factory Pattern is also known as Kit.

3. Singleton Pattern

A Singleton class is one which allows us to create only one object for JVM.

Rules:

- Create Singleton class **Object make it as PRIVATE**
- Create **PRIVATE contrcutor**
- Every Singleton class contains **at least one factory method**

```
class Student {
	private static Student st;
	private Student() {
		System.out.println("OBJECET Created FIRST TIME");
	}
	public static Student getObject() {
		if (st == null) {
			st = new Student();
		} else {
			System.out.println("OBJECET ALREDAY CREATED");
		}
		return st;
	}
}

public class Singleton {
	public static void main(String[] args) {
		Student s1 = Student.getObject();
		Student s2 = Student.getObject();
		System.out.println(s1.hashCode());
		System.out.println(s2.hashCode());
	}
}
```

```
OBJECET Created FIRST TIME
OBJECET ALREDAY CREATED
366712642
366712642
```

Singleton pattern is mostly used in multi-threaded and database applications. It is used in logging, caching, thread pools, configuration settings etc

4. Prototype Pattern

Prototype Pattern says that **instead of creating new Object clone an Existing Object use it**

5. Builder Pattern

Builder Pattern says that "**construct a complex object from simple objects using step-by-step approach"**

XIII EJB

EJB (Enterprise Java Bean) is used to develop scalable, robust and secured enterprise applications in java.Unlike RMI, middleware services such as security, transaction management etc. are provided by EJB Container to all EJB applications.

To run EJB application, you need an application server (EJB Container) such as Jboss, Glassfish, Weblogic, Websphere etc. It performs:

- **life cycle management,**
- **security,**
- **transaction management, and**
- **object pooling**

Difference between RMI and EJB

Both RMI and EJB, provides services to access an object running in another JVM (known as remote object) from another JVM. The differences between RMI and EJB are

RMI	EJB
In RMI, middleware services such as security, transaction management, object pooling etc. need to be done by the java programmer.	In EJB, middleware services are provided by EJB Container automatically.
RMI is not a server-side component. It is not required to be deployed on the server.	EJB is a server-side component, it is required to be deployed on the server.
RMI is built on the top of socket programming.	EJB technology is built on the top of RMI.

EJB and Webservice

- **In EJB, bean component and bean client both must be written in java language**.
- If bean client need to be written in other language such as **.net**, **php** etc, we need to go with **webservices** (SOAP or REST). So EJB with web service will be better option.

Disadvantages of EJB

- Requires application server
- Requires only java client. For other language client, you need to go for webservice.
- Complex to understand and develop ejb applications

References

https://docs.oracle.com/javase/

http://www.javatpoint.com/

http://tutorials.jenkov.com/

https://www.jdoodle.com/faq

https://www.javacodegeeks.com/2015/09/the-java-util-concurrent-package.html

http://www.vogella.com/tutorials/JavaConcurrency/article.html

Synchronization - http://www.journaldev.com/1061/thread-safety-in-java

Java.util.Concurency- https://www.javacodegeeks.com/2015/09/the-java-util-concurrent-package.html

String Constant pool: http://www.java67.com/2014/08/difference-between-string-literal-and-new-String-object-Java.html

Design Patterns - http://www.java2novice.com/java-design-patterns/

Log4j - http://www.java4s.com/log4j-tutorials/

Book -1

Book -2