

JAVA  
again

# Table of Content

---

<b>TABLE OF CONTENT</b> .....	<b>2</b>
<b>1.LANGUAGE FUNDAMENTALS</b> .....	<b>5</b>
FEATURES OF JAVA.....	5
JAVA – JDK, JRE AND JVM.....	7
IDENTIFIERS .....	9
KEYWORDS .....	10
COMMENTS .....	10
DATATYPES.....	10
LITERALS .....	11
ARRAYS .....	13
TYPES OF VARIABLES.....	15
MAIN() METHOD.....	17
OPERATORS .....	17
FLOW CONTROL .....	20
FUNDAMENTALS – INTERVIEW QUESTIONS .....	25
DATA TYPES .....	27
<b>2.CLASS DECLARATION &amp; ACCESS MODIFIERS</b> .....	<b>29</b>
JAVA SOURCE FILE STRUCTURE .....	29
JAVA ACCESS MODIFIERS .....	29
CLASS MODIFIERS – APPLICABLE ONLY FOR CLASSES .....	29
MEMBER MODIFIERS – APPLICABLE FOR METHODS & VARIABLES .....	31
NESTED CLASSES .....	34
<b>3. INTERFACES</b> .....	<b>39</b>
INTERFACE ENHANCEMENTS .....	41
<b>4.OOPS</b> .....	<b>44</b>
1.DATA HIDING .....	44
2.ABSTRACTION .....	44
3. ENCAPSULATION .....	44
4. INHERITANCE.....	45
4. STATIC & INSTANCE CONTROL FLOWS .....	51
5. CONSTRUCTOR.....	54
<b>5.EXCEPTION HANDLING</b> .....	<b>57</b>
DEFAULT EXCEPTION HANDLING .....	57
EXCEPTION HIERARCHY .....	58
USING TRY, CATCH, FINALLY .....	59
USER DEFINED EXCEPTIONS.....	63
EXCEPTION HANDLING WITH METHOD OVERRIDING IN JAVA .....	64
JAVA 1.7 EXCEPTION HANDLING ENHANCEMENTS .....	64
EXCEPTION HANDLING INTERVIEW QUESTIONS.....	67
<b>6. JAVA. LANG PACKAGE</b> .....	<b>68</b>
1.OBJECT CLASS .....	69
2.STRING CLASS.....	75
3,4.STRINGBUFFER, STRINGBUILDER CLASSES .....	78
5.WRAPPER CLASSES .....	81

GARBAGE COLLECTION.....	84
JAVA REFLECTION API (JAVA.LANG.CLASS).....	94
INTERVIEW QUESTIONS.....	95
<b>7. JAVA.IO .....</b>	<b>97</b>
BYTE STREAMS.....	98
CHARACTER STREAMS.....	99
BUFFERED STREAMS .....	100
DATA STREAMS .....	101
OBJECT STREAMS.....	102
PRINTF AND FORMAT METHODS.....	105
JAVA NIO(Non-BLOCKING I/O)-1.4.....	105
<b>8.THREADS .....</b>	<b>107</b>
INTRODUCTION TO MULTI-THREADING.....	107
WHAT IS THREAD .....	108
THREAD LIFE CYCLES (THREAD STATES).....	109
JAVA.LANG.THREAD CLASS .....	110
JAVA.LANG.RUNNABLE INTERFACE.....	112
INTERRUPTING A THREAD.....	114
JOINING A THREAD (JOIN () METHOD).....	116
THREAD PRIORITY .....	117
DAEMON THREAD.....	117
THREAD GROUP .....	118
SYNCHRONIZATION .....	119
INTER THREAD COMMUNICATION .....	120
CALLABLE INTERFACE.....	124
<b>9. JAVA.UTIL.CONCURRENCY.....</b>	<b>125</b>
LOCK INTERFACE .....	125
READWRITELOCK INTERFACE .....	131
CONDITIONS.....	132
<b>10. EXECUTOR FRAMEWORK – THREADPOOLS .....</b>	<b>136</b>
TYPES OF THREADPOOLS .....	139
EXECUTORSERVICE API .....	142
SYNCHRONIZATION UTILITIES – MORE OPTIONS FOR DOING SYNCHRONIZATION.....	154
ATOMIC VARIABLES .....	165
TERMS .....	169
THREADS INTERVIEW QUESTIONS.....	170
<b>11.COLLECTIONS FRAMEWORK .....</b>	<b>180</b>
TRADITIONAL DATA STRUCTURES.....	180
TYPES OF DATA STRUCTURES .....	180
SORTING ALGORITHMS .....	189
SEARCHING ALGORITHMS .....	194
JAVA COLLECTIONS FRAMEWORK.....	195
LIST.....	196
MAP .....	204
SET.....	220
COMPARABLE & COMPARATOR .....	226
QUEUE.....	234
DEQUE (DOUBLE ENDED QUEUE).....	236

CONCURRENT COLLECTIONS.....	237
JAVA.UTIL.ARRAYS.....	242
JAVA.UTIL.COLLECTIONS CLASS.....	243
<b>12. STREAM API.....</b>	<b>245</b>
FUNCTIONAL INTERFACES.....	245
LAMBDA.....	246
STREAMS.....	248
<b>13. JAVA UI (APPLETS/SWINGS).....</b>	<b>259</b>
1. APPLET BASICS.....	259
2. SWING BASICS.....	260
3. AWT (ABSTRACT WINDOWING TOOLKIT).....	262
4. EVENTS HANDLING.....	264
5. COMPONENTS.....	266
<b>FEATURES BY VERSION.....</b>	<b>271</b>
JAVA 4 (2002).....	271
JAVA 5 (2005).....	278
JAVA 7 (2011).....	285
JAVA 8 (2014).....	286
JAVA 9 (2017).....	288
JAVA10 (MARCH 2018).....	289
JAVA 11 (SEPTEMBER 2018).....	290
JAVA 12 (MARCH 2019).....	290
<b>JAVA 13 (SEPTEMBER 2019).....</b>	<b>290</b>
<b>JAVA 14 (MARCH 2020).....</b>	<b>291</b>
JAVA 15 (SEPTEMBER 2020).....	291
JAVA 16 (MARCH 2021) – LATEST.....	292
<b>REF.....</b>	<b>294</b>

Change Table Content --> Top --> REFERENCES (5<sup>th</sup> Tab) --> update content



# 1. Language Fundamentals

Java is a platform independent programming language which is introduced by **James Gosling** and his teammates 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.

## Features of Java

### 1. Simple – No Pointers Dude!

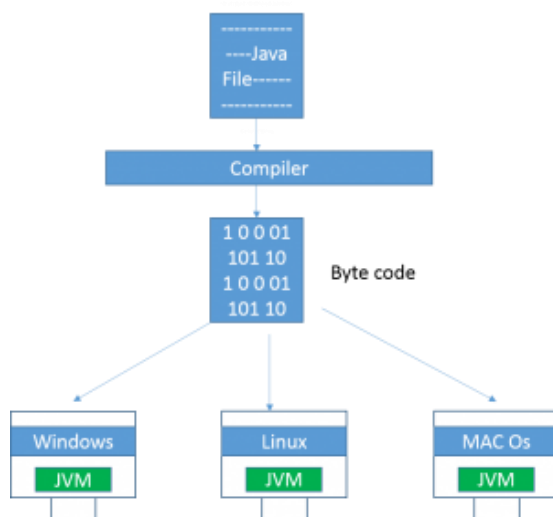
Compare with previous Object-oriented language C++ they removed many confusing and/or rarely-used features e.g., explicit pointers, operator overloading etc. So now no confusions, clean syntax makes java as **Simple**

### 2. Object-oriented – All about java Basics

Java based on OOPs. 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 byte-code 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 create 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 is 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 make 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

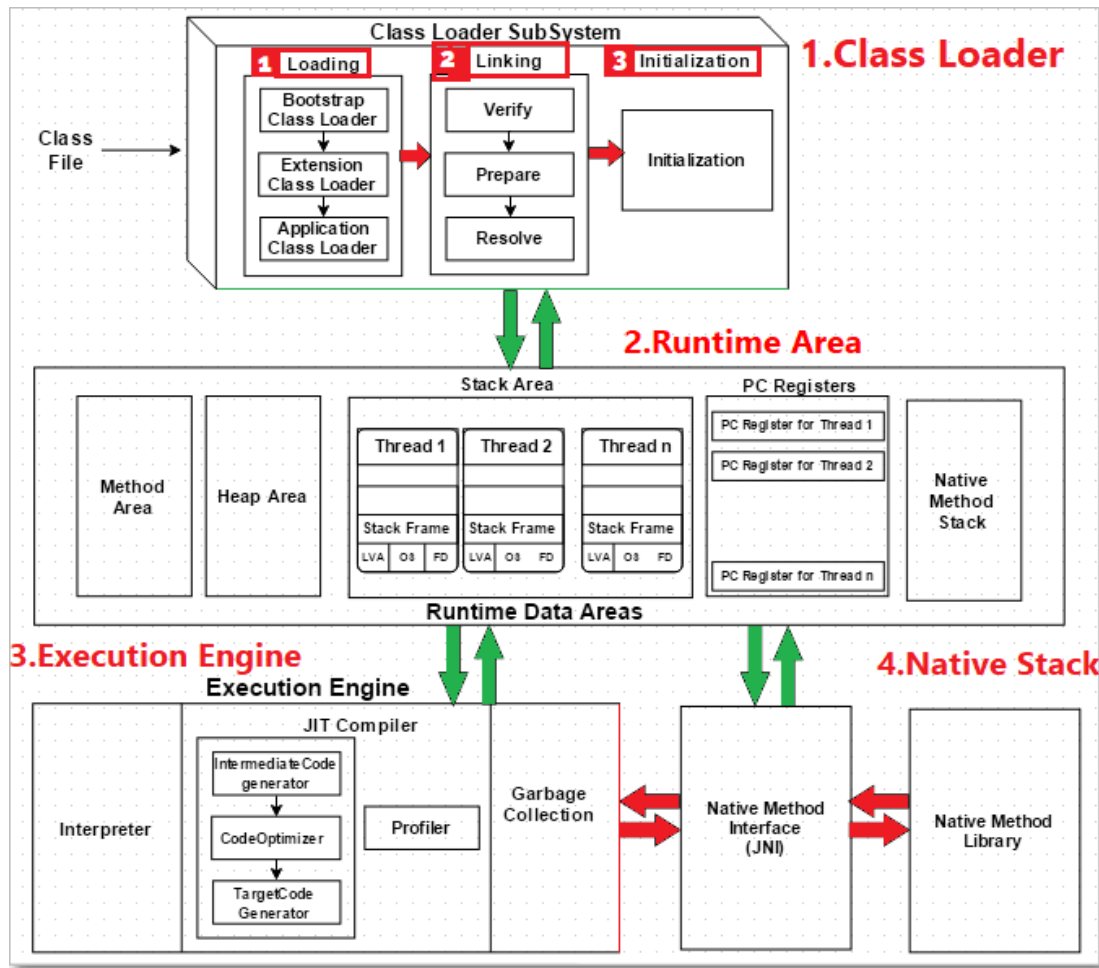
A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it shares the same memory. Threads are important for multi-media, Web applications etc.

#### Class Path:

**CLASSPATH** is a parameter that tells the JVM where to look for classes and packages. it is specific to **Java** only. But **PATH** is specific to system.

When you have set of jar files which are always required during your application runtime, then it's probably best to add them in machine's environment variable 'CLASSPATH'.

During application runtime, application class loader will always scan the jar files and classes at specified paths in this variable.



## ClassLoader

### 1.Loading the Class:

When a Java program is converted into **.class** file by Java compiler **ClassLoader** is responsible to load that class file from file system or any other location.

Our Java class is depending up on any other class, let's say `JdbcDriver.class`, it will search by following Class Loaders

- **Bootstrap ClassLoader** - **JRE/lib/rt.jar**  
First bootstrap class loader tries to find the class. It scans the **rt.jar** file in JRE **lib** folder.
- **Extension ClassLoader** - **JRE/lib/ext** or any directory denoted by `java.ext.dirs`  
If class is not found, then extension class loader searches the class file in inside **lib\ext** folder
- **Application ClassLoader** - **CLASSPATH environment variable, -classpath or -cp option**  
Again, if class is not found then application ClassLoader searches all the Jar files and classes in **CLASSPATH** environment variable of system.

If class is found by any loader then class is loaded by class loader; else **ClassNotFoundException** is thrown

**2.Linking:** once Class is loaded it performs below operations

- **Bytecode verifier** will verify whether the generated bytecode is proper or not.
- **Prepare (memory allocation):** allocates memory **to static variables & methods.**
- **Resolve** – All symbolic memory references are replaced with original references from Method Area.

**3.Initialization:** In prepare only memory is allocated, here all **static variables will be assigned** with the original values and the **static blocks will be executed.**

## Runtime area

**fields (Data members)** and **methods** are also known as **class members.**

- **Method Area:** all **Class level** Data members, Method definitions stored here.
- **Heap** All **Objects** & instance variable Data stored Here.
- **Stacks:** All **Methods executions & Thread Executions** done here. Stores local variables, and intermediate results. Each thread has its own JVM stack, created simultaneously as the thread is created. So, all such local variables are called **thread-local variables.**
- **PC registers:** store the physical memory address of the statements which is currently executing. In Java, each thread has its separate PC register.
- **Native Method Stack:** Java supports and uses **native code** as well. Many low-level codes is written in languages like C and C++. Native method stacks hold the instruction of native code.

## Execution Engine

All code assigned to JVM is executed by an **execution engine.** The execution engine reads the byte code and executes line by line. It uses two inbuilt tools – **interpreter** and **JIT compiler to convert the bytecode to machine code and execute it.**

1. **Interpreter** converts each byte-code instruction to corresponding native instruction. It directly executes the bytecode only one instruction at a time and **does not perform any optimization.**
2. **JIT Compiler takes a block of code** (not one statement at a time as interpreter), optimize the code and then translate it to optimized machine code. **To improve performance, it will Optimizes the bytecode**
3. **Garbage Collection:** Once code Execution done, it will clear the memory.

## Java Native Interface

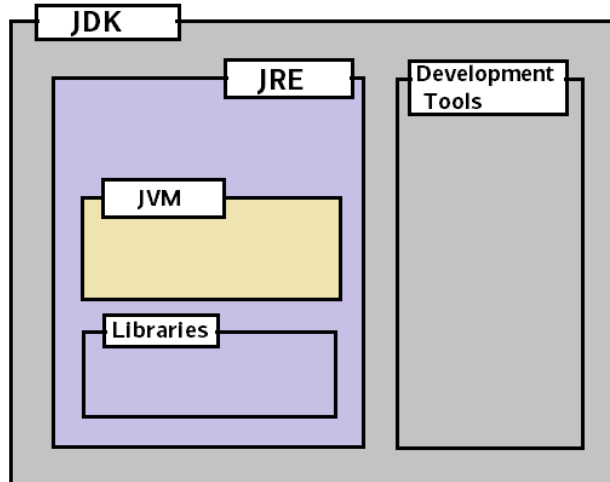
- **Java Native Interface (JNI):** It is an interface which interacts with the Native Method Libraries and provides the native libraries (C, C++) required for the execution.
- **Native Method Libraries:** It is a collection of the Native Libraries which are required by the Execution Engine.

## Differences between JDK, JRE and JVM

**JVM** = Just a Specification. Hotspot VM is implementation of IT

**JRE** = JVM + libraries to run Java application.

**JDK** = JRE + tools to develop Java Application.



### Development Tools

- **Basic Tools** (javac, java, javadoc, apt, appletviewer, jar, jdb, javah, javap, extcheck)
- **Security Tools** (keytool, jarsigner, policytool, kinit, klist, ktab)
- **Internationalization Tools** (native2ascii)
- **Remote Method Invocation (RMI) Tools** (rmic, rmiregistry, rmid, serialver)
- **Java IDL and RMI-IIOP Tools** (tnameserv, idlj, orbd, servertool)
- **Java Deployment Tools** (pack200, unpack200)
- **Java Plug-in Tools** (htmlconverter)
- **Java Web Start Tools** (javaws)

## Identifiers

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

In Above **Hello**, **main**, **args** are called **Identifiers**

### Rules for defining Identifiers

- Only Allowed characters are
  - a-z
  - A-Z
  - 0-9
  - \$
  - -
- if we are using any other symbol, we will get Compile time error "**IllegalCharacter**".
- Identifier should **not start with Number**.
- There is no length limit for java identifiers, but it is not recommended to take more than **15** length.
- All Java class names, Interface names can use as an Identifier, **but it's not recommended**

```
public class Test {  
    int Runnable = 10;  
    int Integer = 20;  
}
```

## Keywords

Some identifiers are reserved to associate some functionality or to represent values, such type of reserved identifiers are called "ReservedWords" / "Keywords"

abstract	continue	for	new	switch
<b>assert***</b>	default	<b>goto*</b>	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	<b>enum****</b>	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	<b>strictfp**</b>	volatile
const*	float	native	super	while

*	not used
**	added in 1.2
***	added in 1.4
****	added in 5.0

## Comments

```
// Single Line Comment
```

The compiler ignores everything from // to the end of the line

```
/* Multi Line comment */
```

The compiler ignores everything from /\* to \*/.

```
/** documentation */
```

This indicates a documentation comment (*doc comment*, for short). The compiler ignores this kind of comment, just like it ignores comments that use /\* and \*/. The **javadoc** tool uses doc comments when preparing automatically generated documentation.

```
javadoc <file_name> or javadoc <package_name> or javadoc *.java
```

## Datatypes

Data Type	Default Value	Default size
Boolean	FALSE	1 bit
Char	'\u0000'	2 byte
Byte	0	1 byte
Short	0	2 byte
Int	0	4 byte
Long	0L	8 byte
Float	0.0f	4 byte
Double	0.0d	8 byte

**int:** The size of int is always fixed irrespective of platform. Hence the chance of failing java program is very less even if you are changing the platform. Hence Java is considered as **Robust** in nature.

# Literals

A literal represents a constant value which can be assigned to the variables

```
int x = 10; int - Datatype, x - variable, 10 - Literal
```

## 1. Integral Literal:

We can specify an integral literal in the following ways.

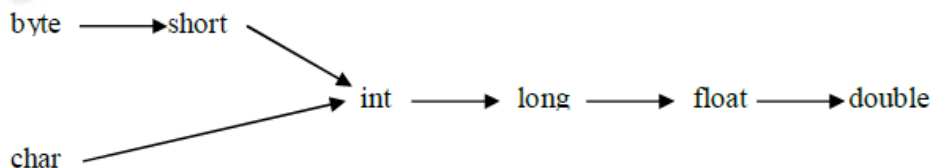
- **Decimal literals:** allowed digits are 0 to 9
- **Binary literals** (digits 0–1): which uses the number 0 followed by b or B as a prefix—for example, 0b10
- **Octal literals** (digits 0–7) : which uses the number 0 as a prefix—for example, 017
- **Hexadecimal literals** (digits 0–9 and letters A–F), which uses the number 0 followed by x or X as a prefix—for example, 0xFF(A=10, B=11, c=12, D=13, E=14, F=15)

```
System.out.println(56); // 56
System.out.println(0b11); // 3 // 1*(2)1 + 1*(2)0 → (1*2) + (1*1) == 3
System.out.println(017); // 15 // 1*(8)1 + 7*(8)0 → (1*8) + (7*1) == 15
System.out.println(0x1F); // 31 // 1*(16)1 + F*(16)0 → (1*16) + (15*1) == 31
```

By default, every **integral literal is of int datatype**. An integral literal is of long type, should suffixing with **l or L**

- 10 - int value.
- 10L - long value

There is no way to specify explicitly an integral literal is of type **byte and short**. If the integral literal is within the range of byte then the JVM by default treats it as byte literal.



## 2. Floating – Point literals

By default, floating-point literals are **double** type. we can specify explicitly as float type by suffixing with **'f' or 'F'**.

```
float f = 10.5; // C.E Type mismatch: cannot convert from double to float
float f = 10.5f;
```

Floating point literals can be specified only in decimal form. i.e we can't use octal and hex decimal representation for floating point literals. But we can assign Octal & hex integer values to float.

```
Double d = 0x123.456; // C.E Invalid hex literal number
Double d = 0x123; //But we can assign Octal & hexa interger values to float
```

added in Java 7. You can have underscores in numbers to make them easier to read:

You can add underscores anywhere **except**

- at the beginning of a literal
- the end of a literal
- right before a decimal point, or right after a decimal point.
- Prior to an F or L suffix

```
int million1 = 1000000;
int million2 = 1_000_000;

double notAtStart = _1000.00; // DOES NOT COMPILE
double notAtEnd = 1000.00_; // DOES NOT COMPILE

double notByDecimal = 1000_.00; // DOES NOT COMPILE
double annoyingButLegal = 1_00_0.0_0; // this one compiles
```

The following example shows other ways you can use the underscore in numeric literals:

```
long creditCardNumber = 1234_5678_9012_3456L;
long socialSecurityNumber = 999_99_9999L;
float pi = 3.14_15F;
long hexBytes = 0xFF_EC_DE_5E;
long hexWords = 0xCAFE_BABE;
long maxLong = 0x7fff_ffff_ffff_ffffL;
byte nybbles = 0b0010_0101;
long bytes = 0b11010010_01101001_10010100_10010010;
```

Following are Invalid Locations

```
long x1 = 999_99_9999_L; // Before L
int x2 = 52_; // At the End Not allowed
int x3 = 5_____2; //correct : Any no.of_'s will allowed between numbers

int x4 = 0_x52; // cannot put underscores in the 0x radix prefix
int x5 = 0x_52; // cannot put underscores at the beginning of a number

int x6 = 0x5_2; correct
```

### 3. Character literal

A char literal can be represented as a single character within single quotes.

```
char ch = 'a';
char ch = 'ab'; C.E: unclosed character literal.
```

we can represent a char literal by its Unicode value. For the allowed Unicode values are 0 to 65535.

```
char ch = 97;
System.out.println(ch); O/P: a

char ch = 65535;
char ch = 65536; C.E : possible loss of precision found : int required :char
```

we can represent a char literal by using Unicode representation which is nothing but \uxxxx'(0-F)

```
char ch = '\u0061'
System.out.println(ch); --> O/P:a
char ch = '\ubeef'; --> O/P: ? (No character defined with this value)
char ch = '\uface'; --> O/P: ?
```

we can also represent a char literal by using escape character.

```
char ch = '\b';
char ch = '\n';
char ch = '\l';
```



# Arrays

An array is a data structure that represents an index collection of fixed no. of homogeneous data elements.

## 1. Declaring Arrays

you can type the [] before or after the name, and adding a space is optional. This means that all four of these statements do the exact same thing:

```
int[] numAnimals;  
int [] numAnimals2;  
int numAnimals3[];  
int numAnimals4 [];
```

The following are the valid declarations for multidimensional arrays.

```
int[][] a;  
int a[][];  
int [][]a;  
int[] a[];  
int[] []a;
```

we can specify the dimension before name of variable also. but this facility is available only for the first variable.

```
int[] a[],b[]; //Correct  
int[] []a,b[]; //Correct  
int[] []a,[]b; //Wrong
```

## 2. Construction of Arrays

**Single Dimension:** Arrays are internally implemented as object. Hence by using new operator we can construct an array.

Compulsory at the time of construction we should specify the size otherwise compile time error.

```
int[] a = new int[10]; ✓  
int[] a = new int[]; C.E : error: array dimension missing
```

It is legal to have an array with size 0 there is no C.E or R.E

```
int[] a = new int[0]; // it will create Empty Array [] (no elements inside it)  
System.out.println(a); //[I@15db9742  
System.out.println(a[0]); //Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
```

If we are specifying array size with some -ve integer, we will get R.E:NegativeArraySizeException.

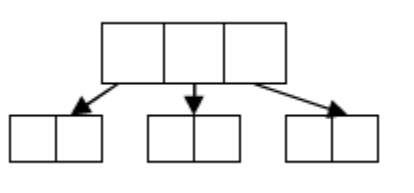
```
int[] a = new int[-10]; // Exception in thread "main" java.lang.NegativeArraySizeException
```

The only allowed Data type to allow the size are **byte, short, char, int**. if we are using any other datatype, we will get a C.E.

```
int[] a = new int[10];  
int[] a = new int[10L]; --> C.E:incompatible types:possible loss of precision found: long required: int  
int[] a = new int[10L]; --> C.E:incompatible types:possible loss of precision found: long required: int
```

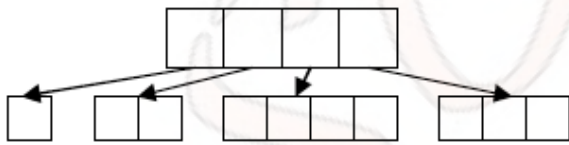
**Multi Dimension:** In java multidimensional arrays are implemented as single dimension arrays. This approach improves performance with respect to memory.

`int[][] a = new int[3][2];` → First Row has 3 locations, each location has 2 elements, total tree height will be 2

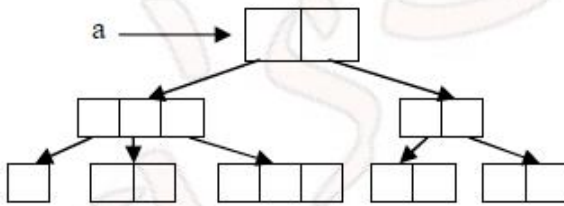


`int[][] a = new int[4][[]]`

`a[0] = new int[1];`  
`a[1] = new int[2];`  
`a[2] = new int[4];`  
`a[3] = new int[3];`



declare an array for the following diagram



`a[][][] = new int[2][][[]];`  
`a[0] = new int[3];`  
`a[0][1] = new int[1];`  
`a[0][2] = new int[2];`  
`a[0][3] = new int[3];`  
`a[1] = new int[2][2];`

### 3. Initialization of arrays

Once we created an array all its elements initialized with default values.

```
int[] a = new int[3];
System.out.println(a[0]); O/P: 0
System.out.println(a); O/P: [I@10b62c9
```

```
int[][] a = new int[3][2];
System.out.println(a); --> [I@10b62c9
System.out.println(a[0]); --> [[I@82ba41
System.out.println(a[0][0]); --> 0
```

```
int[][] a = new int[3][[]];
System.out.println(a); --> [I@10b62c9
System.out.println(a[0]); --> null
System.out.println(a[0][0]); --> NullPointerException
```

### 4. Declaration and Initialization Array in a single line

```
int[] a = {10,20,30};
String[] s = {"Chiru","Allu","Ram","Akil"}
```

## 5.length Vs length():

**length:** It is the final variable applicable only for array objects. It represents the size of the array.

```
int [] a = new int[5];
System.out.println(a.length()); --> C.E
System.out.println(a.length); --> 5
```

**length():** It is the final method applicable only for String Objects. Ex:

```
String s = "raju";
System.out.println(s.length); --> C.E
System.out.println(s.length()); --> 4
```

## Types of Variables

### Declaring Multiple Variables

```
int i1, i2, i3 = 0;
```

As you should expect, three variables were declared: **i1**, **i2**, **i3**. However, only one of those values was initialized: **i3**. The other two remain declared but not yet initialized.

```
int num, String value; // DOES NOT COMPILE
```

This code doesn't compile because it tries to declare multiple variables of different types in the same statement.

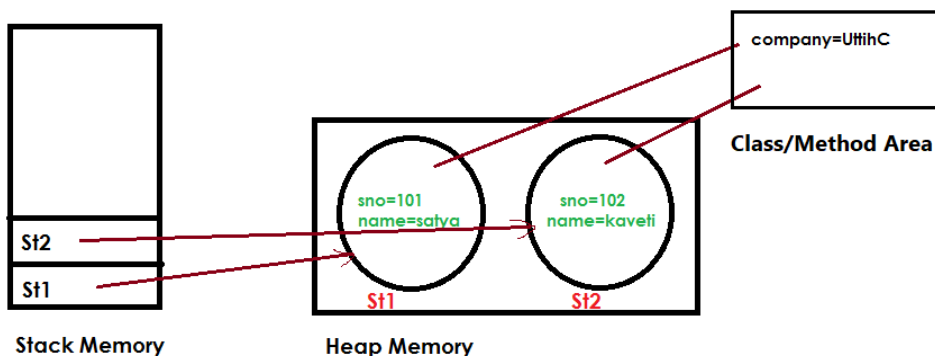
### 1.Instance Variables

- If the value of a variable is varied from object to object. Such type of variables are called as instance variables. For every object a separate copy of instance variables will be created.
- Instance variables will be created at the time of object creation and will be destroyed at the time of object destruction
- All the Instance variables Stored in "**Heap area**"

```
public class Demo
{
    int count = 20;    //1 - Instance variable
}
```

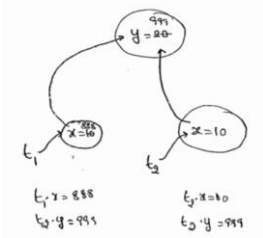
### 2.Staic Variables

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



- If the value of a variable is fixed for all objects, then we have to declare at class level by using **static** keyword. For the static variables a single copy will be created at class level and shared by all objects of that class.
- Static variables will be created at the time of class loading and destroyed at the time of unloading.
- All Static variables are Stored in **"Method Area"**

```
public class Test {
    int x;
    static int y = 20;
    public static void main(String arg[]) {
        Test t1 = new Test();
        t1.x = 888;
        t1.y = 999;
        Test t2 = new Test();
        System.out.println(t2.x + " : " + t2.y); //0 : 999
    }
}
```



### 3. Local variables

- If we are declaring a variable within a method or constants or block such type of variables are called local variables.
- For the local variables JVM won't provide any default values. **Before using a local variable, we should perform initialization explicitly otherwise compile time error.**

```
public class Test {
    public static void main(String arg[]) {
        int i;
        System.out.println(i);
    }
}
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
The local variable i may not have been initialized

public class Test {
    public static void main(String arg[]) {
        int i;
        System.out.println("Here i not used");//No error
    }
}
```

- It is not recommended to perform initialization of local variables in logical blocks because there is no guaranty of execution these blocks at runtime.

```
public class Test {
    public static void main(String arg[]) {
        int i;
        if(arg.length>0){
            i=10;
        }
        System.out.println(i);
    }
}
Test.java:9: error: variable i might not have been initialized
        System.out.println(i);
```

```
class Test {
    public static void main(String arg[]) {
        int i;
```

```
        if (arg.length > 0) {
            i = 10;
        } else {
            i = 20;
        }
        System.out.println(i);
    }
}
//Because if we give arguments 10 will be initialized other wise 20 will be initialized.
It is not good programming practice to perform initialization in logical blocks for local
variables because they may not execute at runtime.
```

- The only applicable modifier for the local variable is **final**. If we are using any other modifier, we will get compile time error.

```
public class Test {
    public static void main(String arg[]) {
        public int a;
        protected int b;
        private int c;
    }
}
Test.java:7: error: illegal start of expression
    public int a;
```

## main() method

JVM always calls main method to start the program. **Compiler is not responsible** to check whether the class contain `main()` or not. Hence, if we don't have main method, **we won't get any C.E. But at runtime JVM raises NoSuchMethodError:main**

## Operators

```
System.out.print(9 / 3); // Outputs 3
System.out.print(9 % 3); // Outputs 0

System.out.print(10 / 3); // Outputs 3
System.out.print(10 % 3); // Outputs 1

System.out.print(11 / 3); // Outputs 3
System.out.print(11 % 3); // Outputs 2

System.out.print(12 / 3); // Outputs 4
System.out.print(12 % 3); // Outputs 0
```

### Numeric Promotion Rules

1. If two values have different data types, Java will automatically promote one of the values to the larger of the two data types.
2. If one of the values is **integral** and the other is **floating**-point, Java will automatically promote the integral value to the floating-point value's data type.
3. Smaller data types, namely **byte**, **short**, and **char**, are first promoted to **int** any time they're used with a Java binary arithmetic operator, even if neither of the operands is int.
4. After all promotion has occurred and the operands have the same data type, the resulting value will have the same data type as its promoted operands.

### What is the data type of x / y?

```
short x = 10;  
short y = 3;
```

In this case, we must apply the third rule, namely that

- first x and y will both be promoted to int
- After promotion, operation will perform, resulting output of **int** type.

### What is the data type of x \* y / z?

All expressions will be calculated from left to right (----->)

```
short x = 14;  
float y = 13;  
double z = 30;
```

we evaluate the multiple and division from left-to-right. In this case, we must apply all of the rules.

- First, x will automatically be promoted to int solely because it is a short and it is being used in an arithmetic binary operation.
- The promoted x value will automatically promoted to a **float** so that it can be multiplied with y.
- The result of (x \* y)/z will then be automatically promoted to a **double**. Final value will be **double**

## 1.Increment/ Decrement

Expression	initial value of x	final value of x	final value of y
y = ++x	4	5	5
y = x++	4	5	4
y = --x	4	3	3
y = x--	4	3	4

- For the final variables we can't apply increment or decrement operators

```
final int i=10;  
i++;  
(or)  
i=20;  
Test.java:8: error: cannot assign a value to final variable i  
    i++;  
    ^
```

- We can apply increment or decrement operators even for **floating point data types** also.

```
double d = 10.5;  
d++;  
System.out.println(d);// 11.5
```

### How this following expression is evaluated?

```
int x = 3;  
int y = ++x * 5 / x-- + --x;  
      (----->)  
// All expressions will calculated from left to right (----->)  
  
System.out.println("x is " + x);  
System.out.println("y is " + y);
```

```
int y = 4 * 5 / x-- + --x; // x assigned value of 4  
int y = 4 * 5 / 4 + --x; // x assigned value of 3  
int y = 4 * 5 / 4 + 2; // x assigned value of 2
```

we evaluate the multiple and division from left-to-right, and finish with the addition. The result is then printed: x is 2 & y is 7

## Does it work?

```
long t = 192301398193810323; // DOES NOT COMPILE
```

It does not compile because Java interprets the literal as an `int` and notices that the value is larger than `int` allows. The literal would need a postfix `L` to be considered a long

```
short x = 10;
short y = 3;
short z = x * y; // DOES NOT COMPILE
```

```
short x = 10;
short y = 3;
short z = (short)(x * y);
```

```
long x = 10;
int y = 5;
y = y * x; // DOES NOT COMPILE
```

In last line could be fixed with an explicit cast to `(int)`, but there's a better way using the compound assignment operator:

```
long x = 10;
int y = 5;
y *= x; //is equals to y=(int) y*x;
```

The compound operator will first cast `x` to a long, apply the multiplication of two long values, and then cast the result to an int.

```
long x = 5;
long y = (x=3);
System.out.println(x); // Outputs 3
System.out.println(y); // Also, outputs 3
```

## Infinity and -Infinity

- In Integer Arithmetic (`byte, int, short`), if anything divide 0 will get R.E: A.E: Divide by 0.
- But In Floating point arithmetic, if anything divides by 0.0, we will get **Infinity/ -Infinity**

```
System.out.println(10 / 0.0); // Infinity
System.out.println(-10 / 0.0); // -Infinity
```

## NaN – Not a Number

- In Integer Arithmetic (`byte, int, short`), 0 divide 0 will get R.E: A.E: Divide by 0.
- But In Floating point arithmetic, 0 divide by 0.0, we will get **Nan (no -Nan is there)**

```
System.out.println(0.0 / 0.0); // NaN
System.out.println(-0.0 / 0.0); // NaN
```

## Equality Operators (==)

The comparisons for equality are limited to same Data Types, so you cannot mix and match types. For example, each of the following would result in a compiler error:

```
boolean x = (true == 3); // DOES NOT COMPILE
boolean y = (false != "Giraffe"); // DOES NOT COMPILE
boolean z = (3 == "Kangaroo"); // DOES NOT COMPILE
```

## Conditional Statements

```
int x = 1;
if(x) { // DOES NOT COMPILE
...
}
```

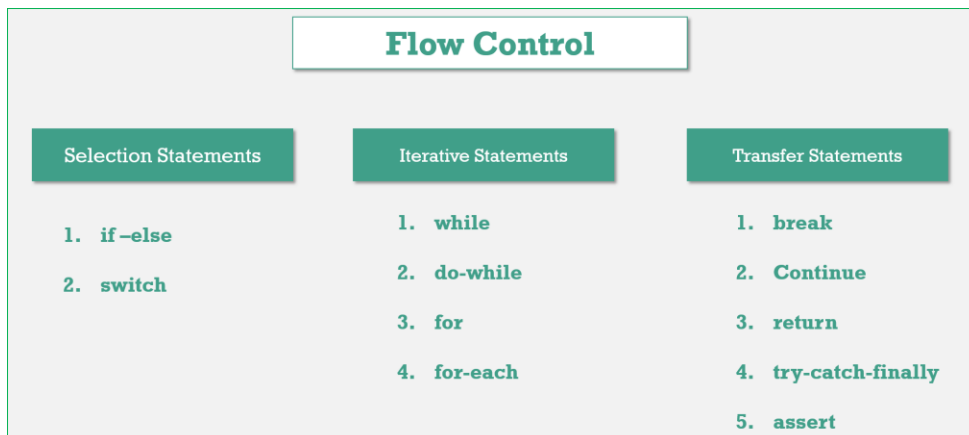
```
int x = 1;
if(x = 5) { // DOES NOT COMPILE
...
}
```

```
System.out.println((y > 5) ? 21 : "Zebra");
int animal = (y < 91) ? 9 : "Horse"; // DOES NOT COMPILE
```

^ (X-OR) - Homogeneous are FALSE(T,T F,F), Heterogeneous are TRUE(T,F F,T)

## Flow Control

Flow control describes the order in which all the statements will execute at run time



## Selection Statements

### 1.if-else :

- The argument in the if statement Should be 'boolean'. If we provide other datatype, it will generate **Compile time Error**

<pre>int x=0; if (x) {     System.out.println("Hello"); } else {     System.out.println("Hi"); } incompatible types: int cannot be converted to boolean</pre>	<pre>boolean b=false; if (b=true) {     System.out.println("Hello"); } else {     System.out.println("Hi"); } //o/p Hello</pre>
---	---

- In the case of if-else statements else part & **curly braces are optional**. Without curly braces we are allowed to take 1 statement under 'if', that statement never be declarative statement otherwise compile time error.

<pre>if(true) System.out.println("hi");</pre>	✔	<pre>if(true) int i = 10</pre>	✘	<pre>if(true) {     int i = 10; }</pre>	✔
---	---	--------------------------------	---	---	---

### 2.Switch:

Syntax:

```
int i = 0;
switch (i) {
case 1:
    ACTION 1;
case 2:
    ACTION 2;
case n:
    ACTION N;
default:
    Def. Action
}
```

- Curly bases are mandatory**
- Inside a Switch **both case and default are optional**



```
int i = 10;
switch (i)
{
    // CORRECT
}
```

- Within switch every statement should be under some **case** or **default** i.e independent statements are not allowed inside switch.

```
int i = 10;
switch(i){
    System.out.println("Hello");
}
//CE : error: case, default, or '}' expected
```

- All **Integer Datatypes (int, short except floating point datatypes), Wrapper Classes, enums (1.5v), Strings(1.7v)** are allowed in switch statements.
- Case labels **must be compiled time constants (final variables)**, variables are not allowed.

```
int i = 10; //final int i=10 - No Error
switch (i) {
case i:
    System.out.println("Hello");
    break;
}
error: constant expression required : case i:
```

- The 'case' labels must be in the range supported by switch argument.

```
byte b = 100;
switch (b) //accepts byte datatype only
{
case 10 :System.out.println("10");
        break;
case 100 :System.out.println("100");
        break;
case 1000 :System.out.println("1000");
        break;
}
error:incompatible types: possible lossy conversion from int to byte(1000 is out of rang)
```

- The case labels & switch arguments can be expressions also, but case label must be constant expression

```
int x=10, y=10;
byte b = 100;
switch (b + 1) {
case 10:
    System.out.println("10");
case 20:
    System.out.println("20");
case 30 + 40:
    System.out.println("30+40 = 70");
case x + y:
    System.out.println("30+40 = 70"); //ERROR
}
Test.java:16: error: constant expression required
case x + y:
```

- Duplicate case labels are not allowed.

**default Case:** In the switch statement, we can place **default** case anywhere. but it is convention to take default case always at last.

Inside switch once we got matched case, then from that statement on words all the statements will execute from top to bottom until break or end of switch

```
switch (x) {
    default:
        System.out.println("default");
    case 0:
        System.out.println("0");
        break;
    case 1:
        System.out.println("1");
    case 2:
        System.out.println("2");
}
```

Here

if 'x' is 0 then	output is 0.
if 'x' is 1 then	output is 1,2.
if 'x' is 2 then	output is 2.
if 'x' is 3 then	output is default, 0.

## Iterative Statements

### 1. While:

```
while(boolean){
//statements...
}
```

- The argument in the while Statement Should be Boolean, otherwise we will get C.E
- **curly braces are optional.** Without curly braces we are allowed to take 1 statement under 'while', that statement never be declarative statement otherwise compile time error.
- If we wrote **boolean argument as Constant value**, it leads to **"UnReachable Statement"**

<pre>while (true) {     System.out.println("hi"); } System.out.println("Hello"); Test.java:12: error: unreachable statement System.out.println("Hello");                 ^</pre>	<pre>while (false) {     System.out.println("hi"); } System.out.println("Hello"); Test.java:4: error: unreachable tatement System.out.println("hi");                 ^</pre>
--	--

- If we wrote constant expression which is never change, it leads to **infinite loop**. If **any final Constant expression**, then the next statement after while will be **"UnReachable Statement"**

<pre>int a = 10; int b = 20; while(a&lt;b) {     System.out.println("Hi"); } System.out.println("Hello"); Output : Hi Hi Hi ... Infinite loop, no error</pre>	<pre>final int a = 10; final int b = 20; while(a&lt;b) {     System.out.println("Hi"); } System.out.println("Hello"); Test.java:4: error: unreachable tatement System.out.println("Hello");                 ^</pre>
---	---

## 2.do-while:

In the loop body has to execute at least once then we should go for do-while loop.

Syntax: **Here ';' is mandatory.**

```
do
{
}while (boolean);
```

- Curly braces are optional; without curly braces we should take only 1 statement between do-while, that statement never be declarative statement

```
do
System.out.println("hi");
while (a>b);
```

```
do
while(a>b);
//Error: at least one statement required
```

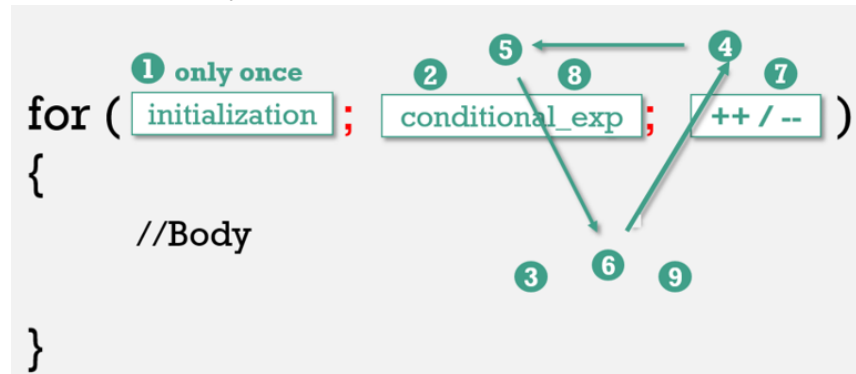
```
do;
while(a>b);
// ; is valid statement
```

- Like while, we will get **Unreachable statement** error, in following case

```
final int a = 10;
final int b = 20;
do
{
    System.out.println("Hi");
}
while (a<b);
System.out.println("Hello");
```

## 3.for:

The most used loop



All the 3 parts of for loop are independent of each other & optional.

```
for(;;); //valid
```

Curly braces are optional, without curly braces we should take only 1 statement, that statement never be declarative statement.

### a. Initialization Section

- This will be executed only once
- Here we can declare multiple variables of same type, but multiple variables of different types are not allowed. Because **break between the variables leads to C.E**

```
for (int x=10, y=20 ; ; ) // No Error
for (int x=10, byte y=20 ; ; ) // ERROR : error: <identifier> expected
```

- In the initialization section we can **take any valid java statement**, including **s.o.p** also.

```
int i = 0;
for(System.out.println("Hi"); i<5; i++)
{
    System.out.print("Hello");
}
//O/p : Hi, Hello ...(5 times)
```

## **b. Conditional Expression**

- Here We can **take any valid Conditional expression**, but result should be boolean type.
- Conditional statement is Optional; if wont specifies anything, **default** should be **'TRUE'**

## **c. Increment/Decrement**

- This statement also Optional.
- In this section we can **take any valid java statement**, including **s.o.p** also.

Unreachable Statement

<pre>for (int i = 0; ; i++) {     System.out.println("Hello"); } System.out.println("Hi");  // unreachable statement System.out.println("Hi");</pre>	<pre>for (int i = 0;false;i++) {     System.out.println("Hello"); } System.out.println("Hi");  // unreachable statement System.out.println("Hello");</pre>	<pre>final int a = 10, b = 20; for(int i = 0;a&lt;b; i++) {     System.out.println("Hello"); } System.out.println("Hi");  // unreachable statement System.out.println("hi");</pre>
--	--	--

## Transfer Statements

All Transfer statements should use

- **inside loops**
- **not in if statements**

### **1.break:**

It can be used in the following places.

- **within the loops** to come out of the loop.
- **inside switch** statement to come out of the switch.
- **If we are using break anywhere else, we will get a compile time error.**

```
int x = 0;
if(x!=5)
break;
System.out.println("if");
C.E: error: break should not outside switch or loop
```

### **2.continue:**

- we should use **'continue'** **only in the loops to skip current iteration** & continue for the next iteration.
- If we are using 'continue' anywhere except loops we will get compile time error saying, **"continue outside of loop"**.

```
for(int i=0;i<10;i++)
{
    if((i%2) == 0)
    continue;
    System.out.print(i);
}
O/P:- 13579
```

## Fundamentals – Interview Questions

### Difference between interpreter and JIT compiler?

The interpreter interprets the bytecode line by line and executes it sequentially. It results in poor performance. JIT compiler add optimization to this process by analyzing the code in blocks and then prepare more optimized machine code.

### Difference between JRE and JVM?

JVM is the specification for runtime environment which executes the Java applications. Hotspot JVM is such one implementation of the specification. It loads the class files and uses interpreter and JIT compiler to convert bytecode into machine code and execute it.

### Difference Between JVM & HotSpot VM

**JVM:** is a Specification, **HotSpot** : is a implementation of JVM.

**HotSpot** is an implementation of the JVM concept, originally developed by Sun and now owned by Oracle.

There are other implementations of the JVM specification, like

- **Open JDK**
- **IBM JVM**
- **SUN JVM**
- **JRockit**
- **Blackdown**
- **Kaffe**

JVM implementations can differ in the **way they implement JIT compiling, optimizations, garbage collection, platforms supported, version of Java supported**, etc.

### How does WeakHashMap work?

WeakHashMap operates like a normal HashMap but uses WeakReference for keys. Meaning if the key object does not hold any reference then both key/value mapping will become appropriate for garbage collection.

### How do you locate memory usage from a Java program?

You can use memory related methods from **java.lang.Runtime** class to get the free memory, total memory and maximum heap memory in Java.

<code>public static Runtime getRuntime()</code>	returns the instance of Runtime class.
<code>public void exit(int status)</code>	terminates the current virtual machine.
<code>public void addShutdownHook(Thread hook)</code>	registers new hook thread.
<code>public Process exec(String command)</code>	executes given command in a separate process.
<code>public int availableProcessors()</code>	returns no. of available processors.
<code>public long freeMemory()</code>	returns amount of free memory in JVM.
<code>public long totalMemory()</code>	returns amount of total memory in JVM.

```

public class TestApp {
    public static void main(String[] args) {
        Runtime r = Runtime.getRuntime();
        System.out.println(r.totalMemory()); //16252928
        System.out.println(r.freeMemory()); //15709576
        System.out.println(r.availableProcessors()); //24
        r.gc();
    }
}

```

### What is ClassLoader in Java?

once Java program is converted into **.class** file, it contains byte code. **ClassLoader** is responsible to load that class file from file system, network or any other location

- Bootstrap ClassLoader - **JRE/lib/rt.jar**
- Extension ClassLoader - **JRE/lib/ext** or any directory denoted by java.ext.dirs
- Application ClassLoader - **CLASSPATH environment variable, -classpath or -cp option, Class-Path attribute of Manifest inside JAR file.**

### Java heap memory

When a Java program started Java Virtual Machine gets some memory from Operating System. whenever we create an object using new operator or by any another means the object is allocated memory from Heap and When object dies or garbage collected, memory goes back to Heap space.

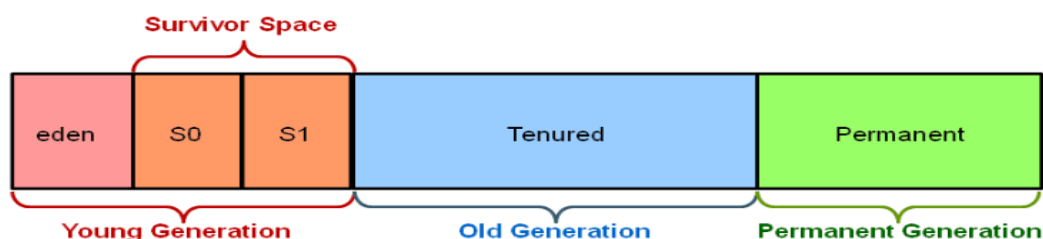
### How to increase heap size in Java

Default size of Heap space in Java is **128MB** on most of 32-bit Sun's JVM. but its highly varies from JVM to JVM. change size of heap space by using JVM **options -Xms and -Xmx**. Xms denotes starting size of Heap while -Xmx denotes maximum size of Heap in Java.

### Java Heap and Garbage Collection

As we know objects are created inside heap memory and Garbage Collection is a process which removes dead objects from Java Heap space and returns memory back to Heap in Java.

For the sake of Garbage collection Heap is divided into three main regions named as **New Generation, Old Generation, and Perm space**



- **New/Young Generation** of Java Heap is part of Java Heap memory where a newly created object is stored,
- **Old Generation** During the course of application many objects created and died but those remain live they got moved to Old Generation by Java Garbage collector thread
- **Perm space** of Java Heap is where JVM stores Metadata about classes and methods, String pool and Class level details.
- Perm Gen stands for permanent generation which holds the meta-data information about the classes.

- Suppose if you create a class name A, its instance variable will be stored in heap memory and class A along with static ClassLoader will be stored in permanent generation.
- Garbage collectors will find it difficult to clear or free the memory space stored in permanent generation memory. Hence it is always recommended to keep the permgen memory settings to the advisable limit.
- JAVA8 has introduced the concept called meta-space generation, hence permgen is no longer needed when you use jdk 1.8 versions.

- Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the `finalize()` method before object is garbage collected.
- The Garbage collector of JVM collects only those objects that are created by `new` keyword. So, if you have created any object without `new`, you can use `finalize` method to perform cleanup processing (destroying remaining objects).
- Neither finalization nor garbage collection is guaranteed.

## Data Types

### How do you convert bytes to String?

you can convert bytes to the string using string constructor which accepts `byte[]`, just make sure that right character encoding otherwise platform's default character encoding will be used which may or may not be same.

```
String str = new String(bytes, "UTF-8");
```

### How do you convert bytes to long in Java

The byte takes 1 byte of memory and long takes 8 bytes of memory? Assignment 1 byte value to 8 bytes is done implicitly by the JVM.

**byte -> short -> int -> long -> float -> double**

The left-side value can be assigned to any right-side value and is done implicitly. The reverse requires explicit casting.

```
byte b1 = 10;           // 1 byte
long l1 = b1;          // one byte to 8 bytes, assigned implicitly
```

### Is ++ operator is thread-safe in Java?

No, it's not a thread safe operator because its involve multiple instructions like reading a value, incrementing it and storing it back into memory which can be overlapped between multiple threads.

### What will this return 3\*0.1 == 0.3? true or false?

Both are not equal, because floating point arithmetic has a certain precision. Check the difference (a-b) it should be really small.

In computer memory, floats and doubles are stored using [IEEE 754](#) standard format.

- $f1 = (0.1+0.1+0.1\dots 11 \text{ times}) = 1.0999999999999999$
- $f2 = 0.1*11 = 1.1$

In **BigDecimal** class, you can specify the rounding mode and exact precision which you want to use. Using the exact precision limit, rounding errors are mostly solved. Best part is that BigDecimal numbers are

immutable i.e. if you create a BigDecimal BD with value "1.23", that object will remain "1.23" and can never be changed. You can use its .compareTo() method to compare to BigDecimal numbers

```
private static void testBdEquality()
{
    BigDecimal a = new BigDecimal("2.00");
    BigDecimal b = new BigDecimal("2.0");

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

    System.out.println(a.compareTo(b) == 0);  // true
}
```

### Which one will take more memory, an int or Integer? (answer)

An Integer object will take more memory an Integer is the an object and it store meta data overhead about the object and int is primitive type so its takes less space.

### How to convert Primitives to Wrapper & Wrapper to Primitive ??

```
// 1. using constructor
Integer i = new Integer(10);

// 2. using static factory method
Integer i = Integer.valueOf(10);

//3.wrapper to primitive
int val = i.intValue();
```

### Autoboxing and Unboxing?

If a **method (remember only method – not direct)** requires Integer Object value, we can directly pass primitive value without issue. Autoboxing will take care about these.

We can also do direct initializations (1.8 V)

```
Integer i = 10; // it will create Integer value of 10 using Autoboxing
int j = i; // ;// it will convert Integer to int using Autoboxing
```

Previously versions of Java (<1.4) shows

```
Integer i = 10; // it will create Integer value of 10 using Autoboxing
int j = i; // But we cant assign int to Integer Type mismatch: cannot convert from Integer to int
```

### what if I make main() private/protected ?

if you do not make **main()** method **public**, there is no compilation error. You will **runtime error** because matching **main()** method is not present. Remember that whole syntax should match to execute **main()** method.

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



## 2. Class Declaration & Access Modifiers

### Java Source File Structure

- A java Source file can contain any no of classes but at most one class can be declared as **public**.
- if there is any **public** class then compulsory the name of the source file and the name of the public class must be matched otherwise, we will get compile time error.
- If there is no public class, then any name we can use for the source file

### Java Access Modifiers

1. **public** – accessible everywhere
2. **protected** – accessible in the same package and in sub-classes
3. **default** – accessible only in the same package
4. **private** – accessible only in the same class

**(Topmost) Classes and interfaces cannot be private. private members are accessible within the same class only.**

There are two levels of access control.

- **Class level** — Allowed modifiers are **public**, **default** only
- **Method level** — Allowed modifiers are **public**, **private**, **protected**, or package-private (default)

### Class Modifiers – Applicable only for classes

For Top – Level Class	For Inner classes
<b>public</b> <b>(default)</b> <b>final</b> <b>abstract</b> <b>strictfp</b> If we are using any other modifier we will get <b>C.E : error: modifier private not allowed here</b>	<ul style="list-style-type: none"><li>• <b>public</b></li><li>• <b>(default)</b></li><li>• <b>protected</b></li><li>• <b>private</b></li><li>• <b>final</b></li><li>• <b>abstract</b></li><li>• <b>strictfp</b></li><li>• <b>static</b></li></ul>

#### final:

**final** is the modifier applicable for **classes, methods, and variables**.

### 1.final at Class level

If a class declared as final, **inheritance is not allowed**.

```
final class P {  
}  
class C extends P {  
}  
Test.java:18: error: cannot inherit from final P  
class C extends P {
```

### 2.final at Method level

If a method declared as final, **we are not allowed to override** that method in child classes

```
class P {  
    public final void marry() {  
        System.out.println("Bujji");  
    }  
}  
class C extends P {  
    public void marry() {  
        System.out.println("Preeti");  
    }  
}  
Test.java:21: error: marry() in C cannot override marry() in P  
    public void marry() {
```

### 3.final at variable level

- If a variable declared as final, **we are not allowed to change it's value**.
- For final instance variables JVM won't provide any default values, compulsory we should perform initialization **before completion of constructor**. The following are the places to perform this

- At the time of declaration:

```
final int i = 0;
```

- Inside instance initialization block

```
final int i;  
{  
    i = 0;  
}
```

- Inside constructor

```
final int i;  
test()  
{  
    i = 0;  
}
```

- Inside static blocks, for static final variables

```
static  
{  
    i = 0;  
}
```

- For the local variables the only applicable modifier is final.
- Before using a local variable (whether it is final or non-final) we should perform initialization. If we are not using local variable, then no need of perform initialization even though it is final.
- Every method presents in final class by default final, but variables are not final.

## abstract:

- abstract modifier is applicable only for **classes and methods**, but not for variables.
- abstract method should have only declaration but not implementation. hence abstract method declaration should end with ;(semicolon)

```
public abstract void m1(); //CORRECT
public abstract void m1(){ //WRONG
```

- If a class contain at least one abstract method, then the corresponding class should be declared as **abstract** otherwise we will get C.E.
- Even though class doesn't contain any abstract method still we can declare that class with abstract modifier. i.e abstract class can contain zero no of abstract methods.

<pre>class Test {     public abstract void m1()     {     } } C_E: abstract methods can't have a body</pre>	<pre>class Test {     public abstract void m1(); } C.E: Test is not abstract and doesn't override abstract method m1 in Test()</pre>	<pre>abstract class Test {     public abstract void m1(); } class SubTest extends Test { } C.E: SubTest is not abstract and doesn't override abstract method m1() in Test.</pre>
---	--	--

## strictfp:

- **strictfp** modifier is applicable only for **methods and classes** but not for variables.
- If a method declared as a strictfp **all floating-point calculations in that method will follows IEEE standard** so that we can **get platform independent results**.
- strictfp and abstract is always illegal combination for methods, but allowed for classes
- If a class declared as strictfp all concrete methods in that class will follow IEEE standard for floating point arithmetic.

## Member modifiers – Applicable for methods & variables

<ol style="list-style-type: none"><li>1. <b>public</b></li><li>2. <b>protected</b></li><li>3. <b>&lt;default&gt;</b></li><li>4. <b>private</b></li></ol>	<ol style="list-style-type: none"><li>5. <b>final</b></li><li>6. <b>static</b></li><li>7. <b>native</b></li><li>8. <b>synchronized</b></li><li>9. <b>transient</b></li><li>10. <b>volatile</b></li></ol>
--	--

## 1. public members

we can access public members from anywhere, but the corresponding class must be visible

## 2. protected members

If a member declared as protected, then we can access that member from anywhere within the current package and only in child classes from outside package.

## 3. <default> members

If a member declared as a default, we can access that member only in the current package.

## 4. private members

If a member declared as private, we can access that member only in the current class.

## 5. final members

- A **final class** cannot be inherited. You cannot create subclasses of final classes.
- If a method declared as final, **we are not allowed to override** that method in child classes
- We cannot change the value of a final variable once it is initialized.

## 6. Static

static is the modifier is applicable for methods and variables but not classes (inner classes allowed).

- **Overloading** is possible for static methods.
- **Inheritance** concept is applicable for static methods, including main(). for example, while executing child class, if child class main() method is not present then parent class main() will execute.

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

```
C:\Users\kaveti_S\Downloads\JUnitHelloWorld\src\main\java>java B
A
```

- It seems Overriding concept is applicable for static methods, **but it's not Overriding, it is "method hiding"**

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

```
C:\Users\kaveti_S\Downloads\JUnitHelloWorld\src\main\java>java A
Parent class
```

```
C:\Users\kaveti_S\Downloads\JUnitHelloWorld\src\main\java>java B
Child class
```

## 7. native modifier

- The methods which are implemented in non-java (like C, C++) are called “**native methods**”. The main objectives of native keyword are
  - To improve performance of the system.
  - To communicate with already existing legacy systems.
- native is the modifier applicable only for methods, **but not classes and variables**.
- native method should end with ; (semicolon). because we are not responsible to provide implementation, it is already available. so *abstract* and *native* is *illegal combination* of modifier
- For the native methods **overloading**, **Inheritance** and **overriding** concepts are applicable.
- The use of native keyword breaks the platform independent nature of java.

```
class Native {
    static {
        System.LoadLibrary("Path of native library") ;// Loading the native library.
    }
    public native void m1(); // Declaring a native method.
}
class client
{
    Native n = new Native();
    n.m1(); // Invoking a native method.
}
```

## 8. Synchronized

- It is a keyword applicable only for **methods** and **blocks**. We can't declare variables and classes with synchronized keyword.
- If a method declared as synchronized at a time only one thread is allowed to execute on the given object. Hence the main advantage of synchronized keyword is we can overcome data inconsistency problem.
- Synchronized methods are implemented methods, so abstract combination is illegal for the methods.

## 9. Transient Modifier

- Transient is the keyword applicable only for variables, but not methods and classes.
- While performing serialization if u don't want to save the value of a particular variable, that variable we have declared with transient keyword.
- At the time of serialization, JVM ignores the value of transient variable and saves its default value.

## 10. Volatile

- Volatile keyword is applicable only for variables.
- it guarantees that value of volatile variable will always be read from main memory and not from Thread's local cache.
- So, we can use volatile to achieve synchronization because it's guaranteed that all reader thread will see updated value of volatile variable once write operation completed.
- volatile provides the guarantee, changes made in one thread is visible to others.

### What is the difference between the volatile and atomic variable in Java?

For example **count++** operation will not become atomic just by declaring count variable as volatile. On the other hand **AtomicInteger** class provides atomic method to perform such compound operation atomically e.g. `getAndIncrement()` is atomic replacement of increment operator. It can be used to atomically increment current value by one. Similarly, you have atomic version for other data type and reference variable as well.

Conclusion						
Modifier	Variables	Method	Top level class	Inner class	Blocks	constructor
public	✓	✓	✓	✓	✗	✓
<default>	✓	✓	✓	✓	✗	✓
protected	✓	✓	✗	✓	✗	✓
private	✓	✓	✗	✓	✗	✓
abstract	✗	✓	✓	✓	✗	✗
final	✓	✓	✓	✓	✗	✗
strictfp	✗	✓	✓	✓	✗	✗
synchronized	✗	✓	✗	✗	✓	✗
native	✗	✓	✗	✗	✗	✗
transient	✓	✗	✗	✗	✗	✗
volatile	✓	✗	✗	✗	✗	✗

## Nested Classes

<b>Inner Class</b>	A class created within class and outside method.
<b>Static Nested Class</b>	A static class created within class and outside method.
<b>Local Inner Class</b>	A class created within method.
<b>Anonymous Inner Class</b>	A class created for implementing interface or extending class. Its name is decided by the java compiler.
<b>Nested Interface</b>	An interface created within class or interface.

## 1. 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

```
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 need to create Outer class Object.
  - we need to add OuterClass class & Outer Class Object before Inner class

```
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. Static Nested Classes (Nested Classes)

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

- **Non-Static Data Members/Methods** : it **Cannot** access 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
```

## 3. 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
```



## 4. 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 returning instance, we can directly implement and will get the object
- **Interface:** Same way, if 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 AnonymousDemo {
    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) {
        AnonymousDemo demo = new AnonymousDemo();
        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 compile generates Anonymous inner class as below

### Class A : Inner class

```
class AnonymousDemo$1 implements A
{
    AnonymousDemo$1(AnonymousDemo paramAnonymousDemo) {}
    public void aShow()
    {
        System.out.println("A show()");
    }
}
```

## Class B : Inner Class

```
class AnonymousDemo$2 extends B
{
    AnonymousDemo$2(AnonymousDemo paramAnonymousDemo) {}
    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

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

## Nested Interface

We can declare an interface in another **interface** or **class**. Such an interface is termed as a nested interface.

The following are the rules governing a nested interface.

- A nested interface declared within an interface must be **public**.
- A nested interface declared within a class can have any access modifier.
- A nested interface is by default **static**.

```
interface Showable {
    void show();
    interface Message {
        void msg();
    }
}

class TestNestedInterface1 implements Showable.Message {
    public void msg() {
        System.out.println("Hello nested interface");
    }

    public static void main(String args[]) {
        Showable.Message message = new TestNestedInterface1();// upcasting here
        message.msg();
    }
}
```

```
class Animal {
    interface Activity {
        void move();
    }
}

class Dog implements Animal.Activity {
    public void move() {
        System.out.println("Dogs can walk and run");
    }
}

public class Tester {
    public static void main(String args[]) {
        Dog dog = new Dog();
        dog.move();
    }
}
```

### 3. Interfaces

- We can declare interface by `interface` keyword & implementing using `implements` keyword. The allowed modifiers for interface are

```
public
abstract
strictfp
<default>
```

```
interface sum1{
}

abstract interface sum2{
}

abstract strictfp interface sum3{
}
```

- Whenever a class implementing an interface, we should provide the implementation for all the interface methods. Otherwise, the class must be declared that class as `abstract`. Violation leads to compile time error.
- By default, all interface methods are `public abstract` & variables are `public static final`.
- Whenever we are implementing an interface method, compulsory we should declare that method as `public`, otherwise we will get compile time error.

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

class Test implements sample {
    void m1() {
    }
}

Test.java:13: error: m1() in Test cannot implement m1() in sample
    void m1() {
        ^
    attempting to assign weaker access privileges; was public 1 error
```

- Every interface variable is by default `public static final`. Hence the following declarations are equal inside interface.

```
int i = 10;
public int i = 10;
public static int i = 10;
public static final int i = 10;
```

- For interface variables we should perform initialization at the time of Declaration only. **Because they are final by default.** For final variables we must provide value at the time of initialization.

```
interface inter
{
int i; C.E = expected.
}
```

- interface variables are by default available in the implemented classes. From the implementation classes we are allowed to access but not allowed to change their values i.e reassignment is not possible because these are final.

```

interface inter {
    int i = 10;
}

class test implements inter {
    public static void main(String arg[]) {
        i=20;
        System.out.println(inter.i);
    }
}

```

B.java:7: error: cannot assign a value to final variable i  
i=20;

- we can re-declare interface variable in implemented class with same variable name, there is no Error, because both are created two different memory areas

```

interface inter {
    int i = 100;
}

class Demo implements inter {
    static int i = 200;

    public static void main(String arg[]) {
        System.out.println(i);
        System.out.println(inter.i);
    }
}

```

E:\Users\Kaveti\_S\Desktop\Codes\NotepadExamples>java Demo  
200  
100

### Naming conflicts in interfaces

- If two interfaces contain a method with same signature and same return type in the implementation class, only one method implementation is enough

```

interface Left {
    void m1();
}

interface Right {
    void m1();
}

class Test implements Left, Right {
    public void m1() {
        System.out.println("method");
    }

    public static void main(String[] args) {
        Left l = new Test();
        l.m1();

        Right r = new Test();
        r.m1();
    }
}

```

method  
method

- If two interfaces contain a method with **same signature but different return type**, then we can't implement those two interfaces simultaneously.

```
interface Left {
    void m1();
}
interface Right {
    int m1();
}
class Test implements Left, Right {
    public void m1() {
        System.out.println("void");
    }
    public int m1() {
        System.out.println("void");
    }
}
Test.java:10: error: m1() in Test cannot implement m1() in Right
    ^ return type void is not compatible with int
```

- If a class contains methods with **same signature but different return type**, it will throw C.E:  
Duplicate method m1() in type Outer

### Marker Interface

- an interface which doesn't contain any methods, treated as 'Marker' interface
- By implementing marker interface, our object will get some special ability(features), such type of interfaces are called "marker" or "taginterface".
- Ex: `Serializable`, `Cloneable` interfaces are marked for some ability.

## Interface Enhancements

1. **default** methods – Java 8
2. **static** methods – Java 8
3. **private** methods – Java 9 (Static & Non-Static also)

### interface Default Methods: Java 8

- Java 8 allows you to add non-abstract methods in interfaces. These methods must be declared **default** keyword.
- Default methods enable you to add new functionality to the interfaces of your libraries and ensure binary compatibility with code written for older versions of those interfaces.

```
interface Vehicle {
    default void move() {
        System.out.println("Def. move");
    }
}
class Car implements Vehicle {
}
class Bus implements Vehicle {
    public void move() {
        System.out.println("Bus. Move");
    }
}
public class Test {
    public static void main(String[] args) {
        new Car().move();
        new Bus().move();
    }
}
Def. move
Bus. Move
```

- **Vehicle** interface defines a method `move()` and provided a default implementation as well. If any class implements this interface then it need not to implement its own version of `move()` method. It can directly call `instance.move()`.
- If class willingly wants to customize the behavior of `move()` method then it can provide its own custom implementation and override the method by removing '**default**' keyword

### interface Static Methods: Java 8

Java interface static method is similar to **default** method except that **we can't override them in the implementation classes**. This feature helps us in avoiding inconsistent results in case of poor implementation in implementation classes

```
interface Vehicle {
    default void move() {
        System.out.println("Def. move");
    }
    static void year(){
        System.out.println("Def. 1998");
    }
}
class Test implements Vehicle{
    @Override
    public void move() {
        System.out.println("Bus. Move");
    }
    static void year(){
        System.out.println("2018");
    }
    public static void main(String[] args) {
        Vehicle.year();
        year();
    }
}
```

```
Def. 1998
2018
```

Note that `year()` is a simple class method, it's not overriding the interface method. For example, if we will add `@Override` annotation to the `year()` method, it will result in compiler error.

### Interface Private Methods – Java 9

private methods will improve code re-usability inside interfaces. For example, if two default methods needed to share code, a private interface method would allow them to do so, but without exposing that private method to its implementing classes.

Using private methods in interfaces have four rules:

- Private interface method **cannot** be **abstract**.
- Private method can be used only inside interface.
- Private static method can be used inside other static and non-static interface methods.
- Private non-static methods cannot be used inside private static methods.

```
public interface Calculator
{
    default int addEvenNumbers(int... nums) {
        return add(n -> n % 2 == 0, nums);
    }

    default int addOddNumbers(int... nums) {
        return add(n -> n % 2 != 0, nums);
    }
}
```

```
private int add(IntPredicate predicate, int... nums) {  
    return IntStream.of(nums)  
        .filter(predicate)  
        .sum();  
}
```

## Functional Interfaces

Functional interfaces are also called *Single Abstract Method interfaces (SAM Interfaces)*. As name suggest, they permit exactly one abstract method inside them.

Java 8 introduces an annotation i.e. `@FunctionalInterface` which can be used for compiler level errors, when the interface you have annotated violates the contracts of Functional Interface.

```
@FunctionalInterface  
public interface Test {  
    public void firstWork();  
}
```

Please note that a functional interface is valid even if the `@FunctionalInterface` annotation would be omitted. It is only for informing the compiler to enforce single abstract method inside interface.

# 4.OOPS

## 1.Data Hiding

Hiding of data, so that outside person can't access our data. The main advantage of data hiding is we can achieve security. Using 'private' modifier we can achieve data hiding.

```
class datademo
{
private double amount;
.....
}
```

## 2.Abstraction

Hiding implementation details is nothing but abstraction. The main advantages of abstraction are we can achieve **security** as we are not highlighting internal implementation. using **interfaces & abstract classes** we can achieve data Abstraction.

## 3. Encapsulation

Encapsulation means the 'encapsulating' or 'wrapping up' of data. This is actually a mechanism that binds data together.

When encapsulation is implemented, only the **variables** inside the class can access it. No class outside the current class can access the variables inside it. This is similar to the fact that no one except you can access the clothes inside your travel bag. You can also declare a method as abstract and add a private access specifier to limit its access outside the class. This is an example of Encapsulation and Abstraction.

Wrapping data and methods within classes in combination with implementation **hiding** (through access control) is often called encapsulation.

If a class follows **Data Hiding**(private) and **Abstraction** (Interfaces) such type of class is said to be 'Encapsulated' class. **Encapsulation = Data Hiding + Abstraction**

### Abstraction VS Encapsulation

- Abstraction is more about '**What**' a class can do. [**Idea**]
- Encapsulation is more about '**How**' to achieve that functionality. [**Implementation**]

```
class Account {
private int balance;

public void setBalance(int balance) {
this.balance = balance; // validating the user & his permissions.
}

public int getBalance() {
return balance; // validating the user and his permissions.
}
}
```

**Encapsulation essentially has both i.e. information hiding and implementation hiding.**



## Tightly Encapsulated Class

A class is said to be tightly encapsulated iff **all the data members declared as private**.

- Only **data members should private**. getters, setters, other methods are **not required to be private**.
- if the parent class is not tightly encapsulated then no child class is tightly encapsulated.

```
class x {
    int i = 0;
}
class y extends x {
    //int i -> hides as public
    private int j = 20;
}
class z extends y {
    //int i -> hides as public
    private int k = 30;
}
```

## 4. Inheritance

### IS-A Relationship

- Also known as '**Inheritance**'.
- By using **extends** keyword we can implement inheritance.
- The main advantage is reusability.

```
class P {
    public void m1() {
        System.out.println("Parent method");
    }
}
class C extends P {
    public void m1() {
        System.out.println("Child method");
    }
}
public class Test {
    public static void main(String[] args) {
        C p = new C();
        p.m1();
    }
}
```

Child method

- Every java class is a direct Child class of 'Object' class

```
Object
|
Class A
```

- If our java class extends any other class, then it is indirect child class of Object

```
Object
|
Class B
|
Class A extends B
```

- Cyclic inheritance is not allowed in java

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

```
Test.java:3: error: cyclic inheritance involving A
class A extends B{
^ 1 error
```

## Has-A Relationship

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

```
class Student
{
    int sno;
    String name;
    Address address;
}
```

## Association, Aggregation and Composition?

### Association

Association in Java is one of the building blocks and the most basic concept of object-oriented programming. **Association is a connection or relationship between two separate classes.**

It shows how objects of two classes are associated with each other. The Association defines the multiplicity between objects. We can describe the Association as a **has-a** relationship between the classes. Has-a relationship is one in which an **object of one class is created as a data member in another class.**

```
class Student
{
    int sno;
    String name;
    Address address;
}
```

Association is a kind of relationship between classes whose objects have an **independent lifecycle** and there is **no ownership** between the objects. It can be **one-to-one, one-to-many, many-to-one, many-to-many.**

#### 1. One-to-one

The best example of a one-to-one association is that one person or one individual can have only one passport. This is a one-to-one relationship between the person and the passport.

#### 2. One-to-many

Suppose there is a Doctor and his patients. So, one doctor is associated with many patients. So this is an example of a one-to-many Association between a doctor and patients.

#### 3. Many-to-one

For example, there can be many books in one library, each book is associated with that library, and it can't be a part of another library. So, many books are related to one library. This is an example of a many-to-one Association between books and a library.

#### 4. Many-to-many

If we talk about a teacher and student, there can be many students associated with one teacher, and also, the teacher can be related to many students. So, the relationship between a teacher and student can be many-to-many.

There are two special forms of Association in Java. They are:

1. Aggregation
2. Composition

## Aggregation

Aggregation in Java is a special kind of association. It represents the Has-A relationship between classes. Java Aggregation allows only **one-to-one** relationships.

If an object is destroyed, it will not affect the other object, i.e., both objects can work independently. Let's take an example. There is an Employee in a company who belongs to a particular Department. If the **Employee** object gets destroyed still the **Department** can work independently

## Composition

The composition is another form of aggregation. In this type of association, the entities are completely dependent on each other. One entity cannot exist without the other. Composition in Java represents a **one-to-many** relationship.

Suppose there is a **House** and inside the house, there are many **rooms**. A single house can have multiple rooms, but a single room cannot have multiple houses. And, if we delete the house, the rooms will automatically be deleted.

## Uses-A Relationship

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

```
class Student {
    int sno;
    String name;

    public static void main(String[] args) {
        Address address = new Address();
    }
}
```

## Overloading

Two methods are said to be overloaded, iff the method names are same, but arguments are different.

```
class Test {
    public void m1() {
        System.out.println("no-args");
    }
    public void m1(int i) {
        System.out.println("int-args");
    }
    public void m1(double d) {
        System.out.println("double-args");
    }
    public static void main(String[] args) {
        Test t = new Test();
        t.m1();
        t.m1(10);
        t.m1(10.5f); // Promoted to next level
    }
}
```

```
no-args
int-args
double-args
```

The overloading method resolution is the responsibility of compiler based on reference type and method arguments. Hence overloading is considered as **compile-time polymorphism**

- In the case of overloading if there is no method with the required argument then the compiler won't raise immediately compile time error. First it will promote arguments datatype to next level and checks is there any matched method with promoted arguments, if there is no such method compiler will promote the argument to the next level and checks for the matched method. After all possible promotions still the compiler unable to find the matched method then it raises compile time error.
- In the case of overloading the more specific version will get the chance first. If Specific version is not available, **Child version will get more priority than Parent version**

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

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

    public static void main(String arg[]) {
        Test t = new Test();
        t.m1("raju"); // String Version
        t.m1(new Object()); // Object Version
        t.m1(null); // String(child) → Object(parent)
    }
}
```

String Version  
Object Version  
String Version

- In case of same level of child classes are available, it will **throws Ambiguity error**

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

    public void m1(StringBuffer o) {
        System.out.println("StringBuffer Version");
    }

    public static void main(String arg[]) {
        Test t = new Test();
        t.m1("raju"); // String Version
        t.m1(null); // Ambiguity Version
    }
}
```

Test.java:13: error: reference to m1 is ambiguous  
t.m1(null); // Ambiguity Version  
both method m1(String) in Test and method m1(StringBuffer) in Test match: 1 error

- **var-arg** method will always **get least priority** i.e if no other method matched then only var-arg method will get chance for execution

## Overriding

If the child class is not satisfied with the parent class implementation, then the child can overwrite that parent class method with its own specific implementation. this concept is nothing but **"overriding"**.

## Rules

**#1: Only inherited methods can be overridden.**

**#2: final and static methods cannot be overridden.**

private, static and final method cannot be overridden in Java. By the way, **you can hide private and static method** but trying to override final method will result in compile time error "Cannot override the final method from a class"

<pre>class P {     public final void m1(){         System.out.println("m1");     } } class Test extends P {     // final void m1 - hidden     public void m1() {     } }</pre>	<pre>class P {     public static void m1(){         System.out.println("m1");     } } class Test extends P {     public void m1(){     } }</pre>	<pre>class P {     private void show(){         System.out.println("show");     } } class Test extends P {     public static void main(String a[]){         Test ob = new Test();         Ob.show();     } }</pre>
<pre>Test.java:8: error: m1() in Test cannot override m1() in P     public void m1(){ overridden method is final 1 error</pre>	<pre>Test.java:8: error: m1() in Test cannot override m1() in P     public void m1(){                 ^ overridden method is static 1 error</pre>	<pre>Demo.java:14: error: cannot find symbol ob.show(); symbol:   method show() show() method not visible to Test class</pre>

**#3: The overriding method must have same return type (or Child type/subtype/Covariant).**

```
class P{
    public Object m1(){
        return null;
    }
}
class Test extends P {
    public String m1(){
        return null;
    }
}
```

**#4: The overriding method must not have lesser access modifier.**

```
class P{
    public String m1(){
        return null;
    }
}
class Test extends P{
    protected String m1(){
        return null;
    }
}
```

```
Test.java:8: error: m1() in Test cannot override m1() in P
protected String m1(){
                ^
attempting to assign weaker access privileges; was public
1 error
```

**#5: The overriding method must not throw new or broader CheckedExceptions. It can have allowed to throw Child Exceptions or remove throws keyword from method signature.**

```

class P{
    public String m1() throws IOException{
        return null;
    }
}
class Test extends P{
    public String m1() throws Exception{
        return null;
    }
}

```

```

Test.java:10: error: m1() in Test cannot override m1() in P
    public String m1() throws Exception{
                ^
    overridden method does not throw Exception : 1 error

```

## Confusing Cases

- **final** method can't be overridden in child classes. **private** methods are not visible in the child classes. Hence, they won't participate in overriding. Based on our requirement we can take exactly same declaration in child class, But It is not overriding.
- A **static** method can't be overridden as non-static, and a non-static method can't be overridden as static method
- If both parent and child class methods are **static**, then there is no compile time error or run time error it seems that overriding is happened, but it is not overriding this concept is called "**method hiding**". All the rules of method hiding are exactly similar to overriding, except both methods declared as static.
- In the case of method hiding method resolution will take care by compiler based on reference type (But not runtime object).
- Overriding concept is not applicable for variables. And it is applicable only for methods. Variable resolution always takes care by compiler based on reference type

```

class P {
    int i = 888;
}
class C extends P {
    int i = 999;
}
class Test {
    public static void main(String arg[]) {
        // Case1:
        P p = new P();
        System.out.println(p.i); // 888

        // Case2:
        C c = new C();
        System.out.println(c.i); // 999

        // Case3:
        P p1 = new C();
        System.out.println(p1.i); // 888
    }
}

```

## 4. Static & Instance Control flows

### Static Blocks

If we want to perform some activity at the time of class loading, Then we should define that activity at static blocks because these (static blocks) will execute at the time of class loading only.

If we want to load native libraries at the time of class loading, then we can place that activity inside "static block".

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

### Static Control Flow

whenever child class is being loaded – automatically its parent classes will be loaded

1. **Identifying the static members** from parent to child (top to bottom).
2. Execution of **static variable assignments & static blocks** from parent to child.
3. Execution of child class main method

During these phases there is one such state called RIWO(Read Indirectly Write Only) for a static variable.

### Read Indirectly Write Only(Read should Indirect & Write is allowed anyway)

If a variable is RIWO state, we can't perform Read operation Directly, if we try to do that it will throw

Compile time Error: **illegal forward reference**

<pre>public class Test{     static int x = 10;     static {         System.out.println(x);     } }</pre>	<pre>public class Test{     static {         System.out.println(x);     }     static int x = 10; }</pre>
Output : 10	Test.java:5: error: illegal forward reference System.out.println(x);

<pre>public class Test{     [2] static {         System.out.println(x);     }     [1] static int x = 10; [3] }</pre>	<p>1. Identifying the static members static int x x = <input type="text"/></p> <p>2. Execution of Static variable assignments &amp; static blocks</p> <pre>static {     System.out.println(x); //C.E }</pre> <p>ce:error: illegal forward reference</p> <p>x = <input type="text" value="10"/></p>
--	--

To resolve this, instead of direct read, we should go for indirect read. Line use method in-between.

```
public class Test{
    static {
        m1();
    }
    static void m1(){
        System.out.println(x);
    }
    static int x = 10;
}
```

Output : 0

class's static default initialization normally happens immediately before the first time one of the following events occur:

- an instance of the class is created,
- a static method of the class is invoked, (this is in above case)
- a static field of the class is assigned,
- a non-constant static field is used

### Static Control flow example

```
class Base {
    static int x = 10;

    static {
        m1(); // a static method of the class is invoked so, y=[0]
        System.out.println("Base Class : static block");
    }

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

    public static void m1() {
        {
            System.out.println("y ==>" + y);
        }
    }
    static int y = 20;
}

class Derived extends Base {
    static int i = 100;

    static {
        m2();
        System.out.println("Derived Class : Static block");
    }

    public static void main(String[] args) {
        m2();
        System.out.println("Derived Class : Main method");
    }

    public static void m2() {
        System.out.println("j ==> " + j);
    }

    static {
        System.out.println("Derived Class : Static block at the end");
    }
    static int j = 200;
}
```

```
C:\Users\src\main\java>java Base
```

```
y ==>0
```

```
Base Class : static block
```

```
y ==>20
```

```
Base Class : Main method
```

```
C:\Users\src\main\java>java Derived
```

```
y ==>0
```

```
Base Class : static block
```

```
j ==> 0
```

```
Derived Class : Static block
```

```
Derived Class : Static block at the end
```

```
j ==> 200
```

```
Derived Class : Main method
```



## Instance Control flow

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

Whenever we are trying to create child class object, the following events will be performed automatically.

**(Remember, here we are creating Child class Object in the main)**

### 1. Identification of instance members from parent to child (top to bottom).

#### 2. Parent Class

- Execution of instance variables assignments and instance blocks only in parent class.
- Execution of parent class constructor.

#### 3. Child Class

- Execution of instance variables assignments and instance blocks only in child class.
- Execution of child class constructor.

All **static initializers** are executed in textual order in which they appear and execute before any instance initializers.

```
class Parent {
    int i = 10;
    {
        System.out.println("First parent Instance block");
    }
    Parent() {
        m1();
        System.out.println("Parent Constructor");
    }
    public static void main(String[] args) {
        Parent p = new Parent();
        System.out.println("parent main");
    }
    public void m1() {
        System.out.println(j);
    }
    static {
        System.out.println("parent static block");
    }
    int j = 20;
}

class Child extends Parent {
    int x = 20;
    {
        m2();
        System.out.println("First child Instance block");
    }
    Child() {
        System.out.println("Child Constructor");
    }
    public static void main(String[] args) {
        Child c = new Child();
        System.out.println(" Child main");
    }
    public void m2() {
        System.out.println(y);
    }
    {
        System.out.println("Second child instance block");
    }
    int y = 200;
}
```

```
C:\Users\kaveti_S\Downloads\JUnitHelloWorld\src\main\java>java Parent
```

```

parent static block
First parent Instance block
20
Parent Constructor
parent main

C:\Users\kaveti_S\Downloads\JUnitHelloWorld\src\main\java>java Child
parent static block
First parent Instance block
20
Parent Constructor
0
First child Instance block
Second child instance block
Child Constructor
Child main

```

### If child class object is not created, then Output is

```

E:\Users\Kaveti_S\Desktop\Codes\NotepadExamples>java Child
parent static block
Child main

```

## Combining Both

### First Static – Only once

1. Identification of static & instance members from parent to child (top to bottom).
2. Execution of static variable assignments & static blocks from parent to child.

### Second - Instance

#### 3. Parent Class

- Execution of instance variables assignments and instance blocks only in parent class.
- Execution of parent class constructor.

#### 4. Child Class

- Execution of instance variables assignments and instance blocks only in child class.
- Execution of child class constructor.

## 5. Constructor

At the time of Object Creation some piece of code will execute automatically to perform initialization. that piece of code is nothing but "**Constructor**". Hence the main Objective of constructor is to perform initialization.

### Rules for writing Constructor

- The name of the constructor and name of the class must be same.
- The only allowed modifiers for the constructors are **public, private, protected, <default>**. If we are using any other modifier, we will get **C.E(Compiler Error)**.

```

class Test
{
static Test(){
----
}
}
C.E:- modifier static not allowed here.

```

- return type is not allowed for the constructors, even `void` also. If we are declaring return type, then the compiler treats it as a method and hence there is no C.E and R.E(RuntimeError).

```

class Test
{
void Test(){
System.out.println("Hai .....");
}
public static void main(String arg[]){
Test t = new Test();
}
}

```

- If we are not writing any constructor, then the compiler always **generates default constructor**.
- If we are writing at least one constructor, then **the compiler won't generate any constructor**. Hence every class contains either programmer written constructor or compiler generated default constructor but not both simultaneously.

Programmer Code	Compiler Code
<pre> class Test{ } </pre>	<pre> class Test {     Test() {         super();     } } </pre>
<pre> public class Test { } </pre>	<pre> public class Test {     public Test() {         super();     } } </pre>
<pre> class Test {     private Test() {     } } </pre>	<pre> class Test {     private Test() {         super();     } } </pre>
<pre> class Test {     void Test() {     } } </pre>	<pre> class Test {     void Test() {     }     Test() {         super();     } } </pre>
<pre> class Test {     Test() {         this(10);     }     Test(int i) {     } } </pre>	<pre> class Test {     Test() {         this(10);     }     Test(int i) {         super();     } } </pre>
<pre> class Test {     Test(int i) {     } } </pre>	<pre> class Test {     Test(int i) {         super();     } } </pre>

- super() & this() in constructor**
  - we should use as first statement in constructor.
  - We can use either `super` or `this` but not both simultaneously.
  - we can invoke a constructor directly from another constructor only

- Inheritance concept is not applicable for constructor, so overriding is also not applicable
- Recursive Constructor invocation leads to Compile-time Exception.

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

    Test(int i) {
        this();
    }
}
Test.java:6: error: recursive constructor invocation
    Test(int i) {
    ^
1 error
```

- whenever we are writing parameterized constructor, it is recommended to provide no-argument constructor as well. If parent class contains parameterized constructor, then while writing child class constructor we should take a bit care

<pre>class p {     p(int i) {     } }  class c extends p {     //no-arg not defiend }</pre>	<pre>class p {     p(int i) {     } }  class c extends p { //Generated Code     c(){         super();     } }</pre>
<pre>Test.java:6: error: constructor p in class p cannot be applied to given types; class c extends p ^</pre>	

- If the parent class constructor throws **checked exception**, Compulsory the child class constructor should throw the same checked exception or its parent otherwise we will get compile time error

```
class p {
    p() throws IOException {
    }
}

class c extends p {
    c() {
        super();
    }
}
Test.java:10: error: unreported exception IOException; must be caught or declared to be thrown
    super();
    ^
```

**Don't confuse in Overriding must throw same or Child class Exception, but in Constructor we must Throw same or Parent class Exception**

- If the parent class constructor throws **unchecked exception**, then child class constructor **not required to throw that exception.**

## 5.Exception Handling

The unexpected unwanted event which disturbs entire flow of the program is known as "Exception"

- If we are not handling exception, the program may terminate abnormally without releasing allocated resources.
- Exception handling means it is not repairing an exception, just providing alternative way to continue the program execution normally.

### Common scenarios where exceptions may occur

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

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

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

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

The wrong formatting of any value may occur `NumberFormatException`.

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

If you are inserting any value in the wrong index, it would result `ArrayIndexOutOfBoundsException`

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

### Default Exception Handling

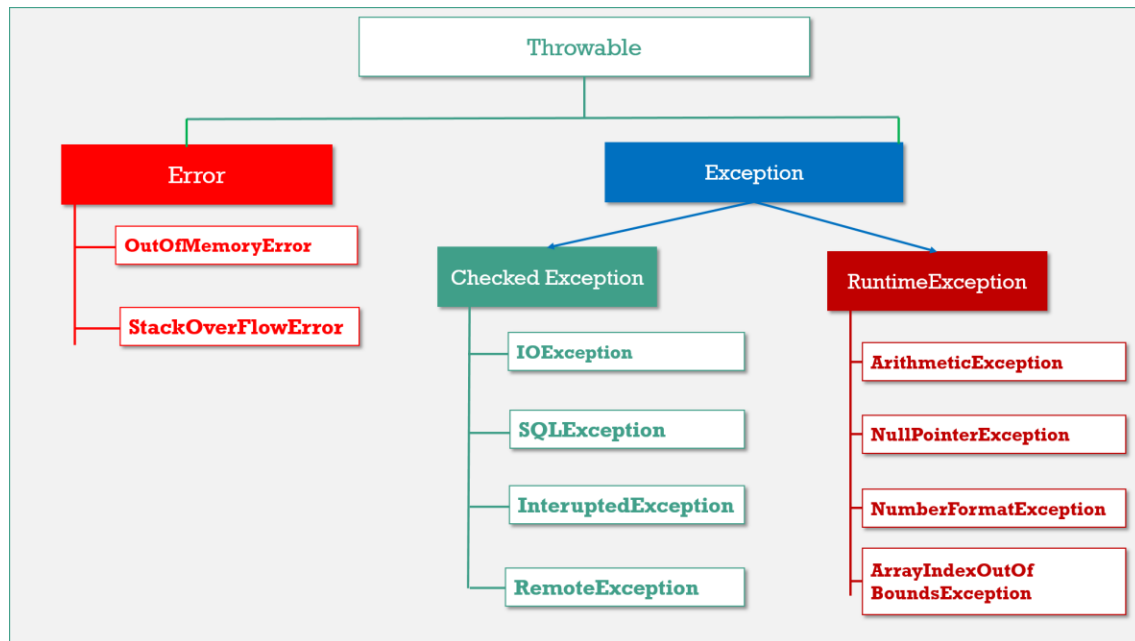
- Whenever an exception raised, the method in which it is raised is responsible for the preparation of exception object by including the following information

```
Name of Exception: Description.  
Location of Exception(Line Numebr)
```

- After preparation of Exception Object, the method handovers the object to the **JVM**, JVM will check for Exception handling code in that method.
- If the method doesn't contain any exception handling code, then JVM terminates that method abnormally and removes corresponding entry from the stack.
- JVM will check for exception handling code in the caller and if the caller method also doesn't contain exception handling code then JVM terminates that caller method abnormally and removes corresponding entry from the stack.
- This process will be continued until main method, if the main method also doesn't contain any exception handling code then JVM terminates main method abnormally.
- Just before terminating the program JVM handovers the responsibilities of exception handling to default exception handler. Default exception handler prints the error in the following format.

```
Name of Exception: Description  
stackTrace
```

# Exception Hierarchy



**Throwable** is the parent of entire java exception hierarchy. It has 2 child classes

- 1) **Exception.**
- 2) **Error.**

## 1.Exception

These are recoverable. Most of the cases exceptions are raised due to bad code.

- **Checked Exceptions:** They Checked by Compiler, they will check that the given resource is existed or not, they are usually occur interacting with outside resources/ network resources e.g., database problems, network connection errors, missing files etc. **Java forces you to handle these error scenarios in some manner in your application code**
- **Unchecked Exceptions:** occurrences of which are not checked by the compiler like coding, initialization, Primitive data errors. They usually result of bad code in your system. **RuntimeException** and its child classes, **Error** and it's child classes are considered as unchecked exceptions and all the remaining considered as checked.

## 2.Error

Errors are non-recoverable. Most of the cases errors are due to lack of system resources but not due to our programs.

**JVM +Memory+ OS level issues.** *OutOfMemory, StatckOverFlow*

### Partially Checked Vs Fully Checked

- **Fully Checked:** A checked exception is said to be **fully checked** iff all its child classes also checked. **Ex: - IOException.**
- **Partially Checked:** A checked exception is said to be partially checked if some of it's child classes are not checked. **Ex: - Exception, Throwable.**

## 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
```

## Using try, catch, finally

- We have to place the risky code inside the try block and the corresponding exception handling code inside catch block.

Without try-catch	With try-catch
<pre>class Test {     public static void main(String arg[]) {          System.out.println("Statement 1");         System.out.println(10 / 0);         System.out.println("Statement 2");      } }</pre>	<pre>class Test {     public static void main(String arg[])     {         System.out.println("Statement 1");         try {             System.out.println(10 / 0);         }         catch (ArithmeticException e)         {             System.out.println(10 / 2);         }         System.out.println("Statement 2");     } }</pre>
Statement 1 Exception in thread "main" java.lang.ArithmeticException: / by zero	Statement 1 5 Statement 2

- In the case of try with multiple catch blocks the order of catch blocks is important. And it should be from child to parent, otherwise Compiler Error. Saying **Exception xxx has already been caught**.
- If there is no chance of raising an exception in try statement, then we are not allowed to maintain catch block for that exception. If we do so, violation leads to compile time error. but this rule is applicable only for fully checked exceptions.

```
class Test {
    public static void main(String arg[]) {
        try {
            System.out.println("Hi");
        } catch (IOException e) {
        }
    }
}
```

```
Test.java:7: error: exception IOException is never thrown in body of corresponding try statement
    } catch (IOException e) {
      ^
```

- It is not recommended to maintain cleanup code within the catch block. because there is no guaranty of execution of particular catch block.
- **finally** block should always execute irrespective of whether the exception is raised or not and handled or not handled.
- The finally block won't be executed, if the system itself exists (JVM shutdown) i.e in the case of **System.exit()** finally block won't be executed.

**Possible combinations of try, catch, finally**

<pre>try { } catch (X e) { } finally { } CORRECT</pre>	<pre>try { } finally { } CORRECT</pre>	<pre>try { } CE: error: 'try' without 'catch', 'finally' or resource declarations</pre>
<pre>try { } System.out.println("Hello"); catch (X e) { } CE: error: 'try' without 'catch', 'finally' or resource declarations</pre>	<pre>try { } catch (X e) { } System.out.println("Hello"); finally { } CE: error: 'try' without 'catch', 'finally' or resource declarations</pre>	<pre>try { } finally { } catch (X e) { } CE: error: 'try' without 'catch', 'finally' or resource declarations</pre>

The following program will demonstrate the control flow in different cases.

```
class Demo {
    public static void main(String arg[]) {

        try{
            statement1;
            statement2;
            statement3;

        } catch (Exception e)
        {
            statement4;
        } finally
        {
            statement5;
        }
        statement6;
    }
}
```



- if there is no exception, then the statements **1, 2, 3, 5, 6** will execute with normal termination.
- if an exception raised at **statement-2** and the corresponding catch block matched, then the statements **1, 4, 5, 6** will execute with normal termination.
- if an exception raised at **statement-2** but the corresponding catch block not matched then the statements **1, 5** will execute with abnormal termination.
- if an exception raised at **statement-2** and while executing the corresponding catch block at **statmnt-4** an exception raised then the statements **1, 5** will execute with abnormal termination.
- if an exception raised at **statement-5** or at **statement-6** then it is always abnormal condition.

### What happens if we put return statement on try/catch? Will finally block execute.?

Yes, finally block will execute even if you put a return statement in the try or catch block.

```
try {
    //try block
    ...
    return success;
}
catch (Exception ex) {
    //catch block
    ....
    return failure;
}
finally {
    System.out.println("Inside finally");
}
```

The answer is yes. **finally** block will execute. The only case where it will not execute encounters **System.exit()**.

<pre>public class Main {     public static void main(String[] args) {         try {             System.out.println("Start...");             System.out.println(10 / 0);         } catch (Exception e) {             System.out.println("Exception..");         } finally {             System.out.println("Finally...");         }     } }</pre> <p>Start... Exception... Finally...</p>	<pre>public class Main {     public static void main(String[] args) {         try {             System.out.println("Start...");             System.out.println(10 / 0);         } catch (Exception e) {             System.exit(0);         } finally {             System.out.println("Finally...");         }     } }</pre> <p>Start... (abnormal Termination)</p>
--	--

### What happens when a finally block has a return statement?

Finally block overrides the value returned by try and catch blocks.

```
public static int myTestingFuncn(){
    try{
        ....
        return 5;
    }
    finally {
        ....
        return 19;
    }
}
```

This program would return value 19 since the value returned by try has been overridden by finally.

## Throws

In our code, if there is a chance of raising **checked exception**, then compulsory we should handle that checked exception either by using try, catch or we have to delegate that responsibility to the caller using **throws** keyword otherwise **C.E : must be caught or declared to be thrown**

Throws will give 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

## 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.

```
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)
```

## 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)

```
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)
```

In Real time CSW Id Start with A1 & Contains 8-digit number, so if given ID not met the requirement Throws CSWException

## Exception Handling with Method Overriding in Java

### If the superclass method does not declare an exception

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

### 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 (delete exception) **but cannot declare parent exception.**

### If the parent class constructor throws checked exception,

If the parent class constructor throws checked exception, Compulsory the child class constructor should throw the **same checked exception or it's parent otherwise** we will get compile time error

**Don't confuse in Overriding must throw same or Child class Exception, but in Constructor we must Throw same or Parent class Exception**

## Java 1.7 Exception handling Enhancements

### Java try-with-resources

- Before java 7, to clean up the resources we have to use **finally** blocks to write closing statements manually.
- With Java 7, no need to explicit resource cleanup required, It's done automatically.
- Automatic resource cleanup done when we are initializing resource in try block. **try(resource).**
- Cleanup happens because of new interface **AutoCloseable**. Its **close()** method is invoked by JVM as soon as try block finishes.
- In java 7, we have a new super interface **java.lang.AutoCloseable**. This interface has one method:  

```
void close() throws Exception;
```

**All File.IO.\* Streams(InputStream,OutputStream,) & ResultSet** by default implements AutoClosable interface.
- When we open any such AutoCloseable resource in special try-with-resource block, immediately after finishing the try block, **JVM calls this close() method on all resources initialized in "try()" block.**
- If you want to use this in custom resources, then implementing AutoCloseable interface is mandatory. otherwise, program will not compile.

## Example

## Case 1 : Try-with- Single resource

```
public class Test {
    public static void main(String args[]) {
        try (FileInputStream input = new FileInputStream("file.txt")) {

            int data = input.read();
            while (data != -1) {
                System.out.print((char) data);
                data = input.read();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
Satya
shiva
rma
hanu
vinay
govind
ramesh
```

The opened resource will be limited to inner try block, so not available for **catch/final**

```
try (FileInputStream input = new FileInputStream("file.txt"))
{
    input.read(); // Available, No Issue
}
catch (Exception e) {
    e.printStackTrace();
    input.close(); // Not Available, No Issue
}
finally {
    input.close();
}
```

```
Test.java:18: error: cannot find symbol      input.close();
                        ^
```

## Case 2: Try-with- Multiple Resources

we can declare it by line by line separated by semicolons.

```
public class Test {
    public static void main(String args[]) {
        try (
            FileInputStream input = new FileInputStream("file.txt");
            FileOutputStream output = new FileOutputStream("file.txt")
        ) {
            // Write to file
            for (int i = 0; i < 10; i++) {
                output.write(i);
            }

            // Read from File
            int data = input.read();
            while (data != -1) {
                System.out.print(data);
                data = input.read();
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
        }
    }
}
```

### Case 3: Try-with- Resources – Exception in Try block

If a try block throws an exception and one or more exceptions are thrown by the try-with-resources, the exceptions thrown by try-with-resources are **suppressed**.

You can get these **suppressed exceptions** by using the **Throwable.getSuppressed()** method of Throwable.

```
public class Test {
    public static void main(String args[]) {
        try (FileInputStream input = new FileInputStream("nofile.txt")); {

            } catch (Exception e) {
                System.out.println("Catch block");
                e.printStackTrace();
            } finally {
                System.out.println("finally");
            }
        }
    }
}

```

Catch block  
[java.io.FileNotFoundException](#): nofile.txt (The system cannot find the path specified)  
at core.Test.main([Test.java:9](#))

finally

### Case 4: Java9 Enhancement - Resource declared outside the try block

In Java 7, try-with-resources has a limitation that requires resource should declare within try block only, otherwise compiler generates an error: **<identifier> expected**.

```
FileInputStream input = new FileInputStream("file.txt");
try (input) {
}

```

error: <identifier> expected  
try(input)

To deal with this error, try-with-resource is improved in **Java 9** and now we can use reference of the resource that is not declared locally.

```
public class Test {
    public static void main(String args[]) throws IOException {
        FileInputStream input = new FileInputStream("file.txt");
        try (input) {
            int data = input.read();
            while (data != -1) {
                System.out.print((char) data);
                data = input.read();
                input.close();
            }
        } catch (Exception e) {
            e.printStackTrace();
            input.close();
        }finally {
            input.close();
        }
    }
}

```

Now we can access input variable in catch and finally block

### Catch with Multiple Exception classes

From Java 7 onwards, you can catch multiple exceptions in single catch block using (|) symbol.

```
catch (NullPointerException | IndexOutOfBoundsException ex)
{
    Ex.printStackTrace();
}

```

- If a **catch** block handles more than one exception type, then the **catch** parameter is implicitly **final**. In this example, the **catch** parameter **ex** is **final** and therefore you cannot assign any values to it within the **catch** block.
- The Exceptions must be in same level of Hierarchy.

```
catch(NullPointerException | Exception ex) {
    throw ex;
} The exception NullPointerException is already caught by the alternative
```

## Exception Handling Interview Questions

### What will happen if you put System.exit(0) on try or catch block?

In normal Finally block will always execute. The only case finally block is not executed is **System.exit(0)**. In advanced case it will execute in following case.

By Calling System.exit(0) in try or catch block, its stops execution & throws **SecurityException**.

- If **System.exit(0)** **NOT** throws SecurityException, then **finally block Won't be executed**
- if **System.exit(0)** throws SecurityException then **finally block will be executed**.

**java.lang.System.exit()** will terminates the currently executing program by JVM.

- **exit(0)** : Generally used to indicate **successful** termination.
- **exit(1) or exit(-1) or any other non-zero value** –indicates **unsuccessful** termination.

### What happens if we put return statement on try/catch? Will finally block execute?

Yes, finally block will execute even if you put a return statement in the try block or catch block.

```
public class Demo {
    public static int div(int a, int b) {
        try {
            return a / b;
        } catch (ArithmeticException e) {
            System.out.println(" In Catch : 0");
            return 0;
        } finally {
            System.out.println(" In Finally : 1");
            return 1;
        }
    }
    public static void main(String[] args) {
        System.out.println("Return Value is : " + div(10, 0));
    }
}
```

```
In Catch: 0
In Finally: 1
Return Value is: 1
```

**finally** block will execute. The only case where it will not execute is when it encounters **System.exit()**.

### What happens when a finally block has a return statement?

Finally block overrides the value returned by try and catch blocks.

```
public static int myTestingFuncn(){
    try{
        ....
        return 5;
    }
    finally {
        ....
        return 19;
    }
}
```

This program would return value 19 since the value returned by try has been overridden by finally.

## Why do you think Checked Exception exists in Java, since we can also convey error using RuntimeException?

**Most of checked exceptions are in java.io package**, which make sense because if you request any system resource and it is not available, then a robust program must be able to handle that situation gracefully.

By declaring **IOException** as checked Exception, Java ensures that you should provide that gracefully exception handling. Another possible reason could be to ensuring that system resources like file descriptors, which are limited in numbers, should be released as soon as you are done with that using catch or finally block

## Have you faced OutOfMemoryError in Java? How did you solve that?

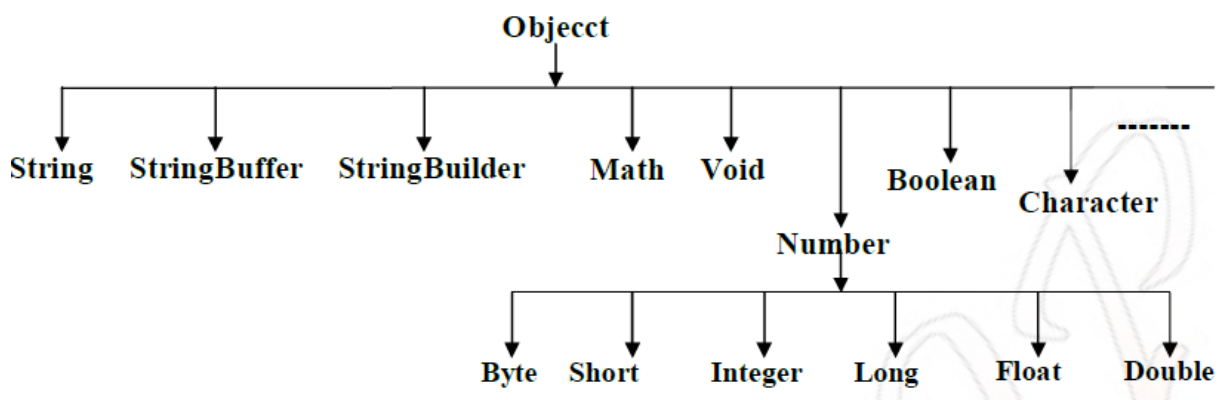
**OutOfMemoryError** in Java is a subclass of `java.lang.VirtualMachineError` and JVM throws `java.lang.OutOfMemoryError` when it **ran out of memory in the heap**.

An easy way to solve `OutOfMemoryError` in java is to increase the maximum heap size by using JVM options `"-Xmx512M"`, this will immediately solve your `OutOfMemoryError`.

## 6. java. lang package

The most commonly used and general-purpose classes which are required for any java program are grouped into a package which is nothing but a `"java.lang.package"`.

All the classes and interfaces which are available in this package are by default available to any java program. There is no need to import this class.



1. **Object** Class
2. **String** class
3. **StringBuffer** Class
4. **StringBuilder** Class
5. **Wrapper** Classes



# 1.Object Class

The most common general methods which can be applicable on any java object are defined in object class. Object class is the parent class of any java class, whether it is predefined, or programmer defined. Hence all the object class methods are by default available to any java class.

## Object class define the following 11 methods

1. `toString()`
2. `equals(Object otherObject)`
3. `hashCode()`
  
4. `clone()`
5. `finalize()`
6. `Class getClass()`
  
7. `void wait()`
8. `void wait(long ms)`
9. `void wait(long ms, int nano)`
10. `void notify()`
11. `void notifyAll()`

### 1. **String toString():** Returns a string representation of the object.

- Whenever we are passing object reference as argument to **System.out.println()** internally JVM will call `toString()` on that object.
- If we are not providing implementation to `toString()` method, then Object class `toString()` will be executed which is implemented as follows

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

### 2. **boolean equals(Object otherObject)** – It is used to simply verify the equality of two objects. Its default implementation is simply checking the references of two objects, to verify their equality. **By default, two objects are equal if and only if they are stored in the same memory address.**

In **String** class (**not StringBuilder, not StringBuffer**) & **All Wrapper classes** `equals()` method is overridden for Content Comparison

#### **a. equals() at Object comparison level**

- If we are comparing **non-String Objects** `equals()` method it **compares references of Objects**.
- **It is same as "==" Operator**

#### **b. equals() at String Comparison Level**

- If we are comparing **String data on** `.equals()` method **compares only content**.
- **References are doesn't matter.**

**3. `int hashCode()`** – Returns a unique integer value for the object in runtime. By default, integer value is mostly derived from memory address of the object in heap (but it's not mandatory always).

### Relation between `equals()` & `hashCode()` methods

- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode()` method on each of the two objects must produce the same integer result.

```
String s1 = new String("Satya");
String s2 = new String("Satya");
System.out.println("s1: "+s1.hashCode());
System.out.println("s2: "+s2.hashCode());
System.out.println(s1.equals(s2));
```

```
s1: 79657294
s2: 79657294
true
```

- **Whenever we override the `equals()` method, we should override `hashCode()` method.**
- In **String class(not `StringBuilder`, not `StringBuffer`) & All Wrapper classes `equals()` method is overridden for Content Comparison.**

### Compare two employee Objects based on Their Id?

```
public class Employee {
    int id;
    String name;
    //Setters & Getters
    @Override
    public boolean equals(Object obj) {
        Employee e = (Employee) obj;
        boolean flag = false;
        if (this.getId() == e.getId()) {
            flag = true;
        }
        return flag;
    }
    public static void main(String[] args) {
        Employee e1 = new Employee();
        Employee e2 = new Employee();
        e1.setId(101);
        e2.setId(101);
        System.out.println(e1.equals(e2)); //true
    }
}
```

So, are we done? Not yet. Let's test again above modified `Employee` class in different way.

```
public static void main(String[] args) {
    Employee e1 = new Employee();
    Employee e2 = new Employee();
    e1.setId(101);
    e2.setId(101);

    Set<Employee> set = new HashSet<>();
    set.add(e1);
    set.add(e2);
    System.out.println(set); //[basic.Employee@15db9742, basic.Employee@6d06d69c]
}
```

Above class prints two objects in print statement. If both employee objects have been equal, in a `Set` which stores only unique objects, there must be only one instance inside `HashSet`.

We are missing the second important method `hashCode()`. As java docs say, if you override `equals()` method then you **must** override `hashCode()` method

```

public class Employee {
    int id;
    String name;
    @Override
    public boolean equals(Object obj) {
        Employee e = (Employee) obj;
        boolean flag = false;
        if (this.getId() == e.getId()) {
            flag = true;
        }
        return flag;
    }
    @Override
    public int hashCode() {
        return getId();
    }
    public static void main(String[] args) {
        Employee e1 = new Employee();
        Employee e2 = new Employee();
        e1.setId(101);
        e2.setId(101);
        Set<Employee> set = new HashSet<>();
        set.add(e1);
        set.add(e2);
        System.out.println(set); //[basic.Employee@65]
    }
}

```

Apache commons provide two excellent utility classes [HashCodeBuilder](#) & [EqualsBuilder](#) for generating hash code and equals methods. [Ref.Example](#)

### HashCodeBuilder

create an instance of HashCodeBuilder. Append the fields we're gonna use to calculate the hashCode. The final result of the actual hashCode can be obtained by calling the toHashCode() from the instance of HashCodeBuilder.

### EqualsBuilder

On the first line you can check to see if the passed object is an instance of the same object, we use the instanceof operator. We then compare the values stored in both objects using the EqualsBuilder class. To get the equality result you must remember to call the isEqual() method.

```

public class Emp {
    int id;
    String name; // Setters,Getters,Constructors

    @Override //Implement the hashCode method using HashCodeBuilder.
    public int hashCode() {
        return new HashCodeBuilder().append(id).append(name).toHashCode();
    }

    @Override //Implement the equals method using the EqualsBuilder.
    public boolean equals(Object obj) {
        if (!(obj instanceof Emp)) {
            return false;
        }
        Emp e = (Emp) obj;
        return new EqualsBuilder().append(this.id, e.id).append(this.name, e.name).isEqual();
    }
    public static void main(String[] args) {
        Emp e1 = new Emp(101, "Satya");
        Emp e2 = new Emp(101, "Satya");
        System.out.println(e1);
        System.out.println(e2);
        System.out.println(e1.equals(e2));
    }
}

```

core.Emp@4bfe2d0  
core.Emp@4bfe2d0  
true

#### 4.Object clone():

Cloning is the process of creating a copy of an Object.

```
Test t1 = new Test();
Test t2 = (Test)t1.clone();
```

An Object is said to be cloneable iff the corresponding class has to implement **java.lang.cloneable** interface. It doesn't contain any methods it is a marker interface.

#### Rules

- To clone an Object, it must implement **java.lang.Cloneable** 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 - **Shallow Cloning & Deeply Cloning**

#### 1. Shallow copy Cloning – Default Implementation

- 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**.
- Shallow copy is not 100% independent of original object.

```
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) {
        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 – Override clone method

- 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 addition, 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.

```

class Address implements Cloneable{
    String dno;
    String city;
    public Address(String dno, String city) {
        this.dno = dno;
        this.city = city;
    }
    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}

class Student implements Cloneable {
    int sno;
    String name;
    Address address;
    public Student(int sno, String name, Address address) {
        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 : HYDERABAD
        //S1,S2 are independent to each other
    }
}

```

HYDERABAD : HYDERABAD  
KANURU : HYDERABAD

**5. void finalize():** Called by the garbage collector on an object, when garbage collection determines that there are no more references to the object.

**6. Class getClass():** Returns the runtime class of an `object.getClass()`, or the class-literal

- `Foo.class` return a Class object, which contains some metadata about the class:

- name
- package
- methods
- fields
- constructors
- annotations

we can create Class object by following ways

```
Class c = Class.forName("StudentB0")
```

```
Class c = StudentB0.class
```

```
Class c = ob.getClass();
```

```
public static void main(String[] args) throws Exception {
    TestApp a = new TestApp();
    Class c1 = a.getClass();
    Class c = Class.forName("java.lang.String");
    System.out.println("Class represented by c : " + c.toString());

    Object obj = c.newInstance();
}
```

```
public class Test {
    public static void main(String args[]) throws IOException, ClassNotFoundException {
        Class c = Class.forName("java.lang.Object");
        System.out.println("Name : "+c.getName());
        System.out.println("getConstructors : "+c.getConstructors());
        System.out.println("getFields : "+c.getFields());
        System.out.println("getMethods : "+c.getMethods());

        java.lang.reflect.Method[] methods = c.getMethods();
        for (java.lang.reflect.Method m : methods) {
            System.out.println("====> "+m);
        }
    }
}
```

```
====> public final void java.lang.Object.wait(long,int)
====> public native int java.lang.Object.hashCode()
====> public final native void java.lang.Object.notify()
====> public final native void java.lang.Object.notifyAll()... all 9 methods will come
```

**7. void wait():** current thread will wait, until another thread notifies

**8. void wait(long ms):** current thread will wait for the specified milliseconds, until another thread notifies

**9. void wait(long ms, int nano):** current thread will wait for the specified milliseconds and nanoseconds, until another thread notifies.

**10. void notify():** Wakes up a single thread that is waiting on this object's monitor.

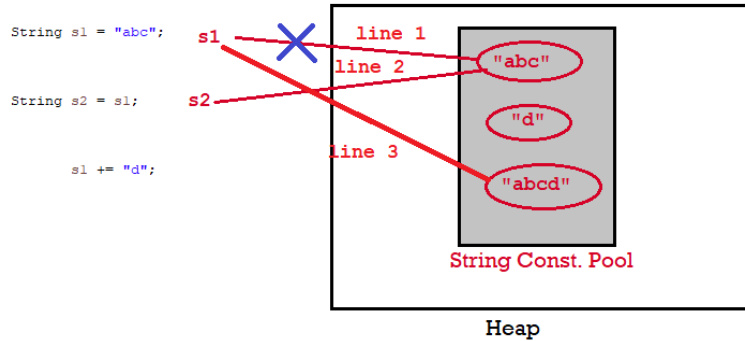
**11. void notifyAll():** Wakes up all threads that are waiting on this object's monitor.



### Case 2:

```
String s1 = "abc";  
String s2 = s1;  
s1 += "d";  
System.out.println(s1+", "+s2+", "+(s1==s2));
```

abcd, abc, false

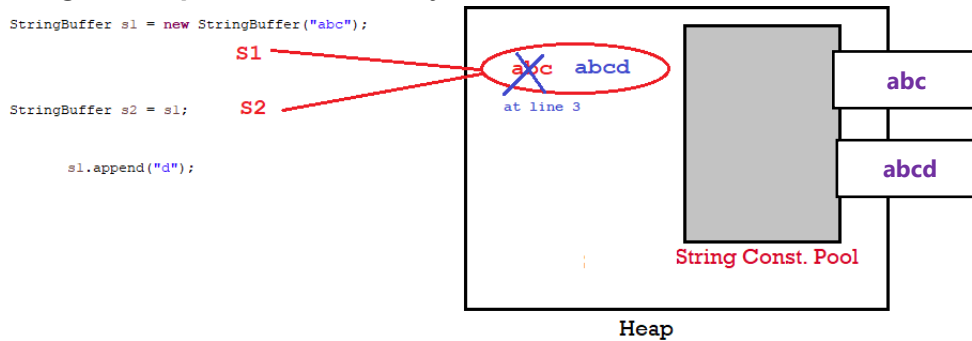


### Case 3:

```
StringBuffer s1 = new StringBuffer("abc");  
StringBuffer s2 = s1;  
s1.append("d");  
System.out.println(s1+", "+s2+", "+(s1==s2));
```

abcd, abcd, true

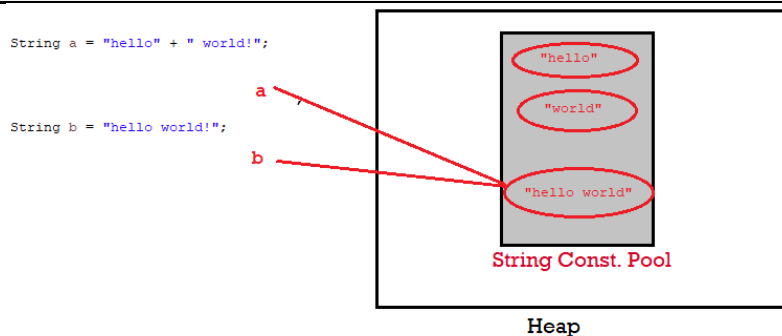
#### StringBuffer operates on Same Object.



Whenever we created String of Object type, first it will store in SCP & then String of Object type will be created. This is applicable for `StringBuffer` & `StringBuilder` also

### Case 4:

```
String a = "hello" + " world!";  
String b = "hello world!";  
System.out.println(a==b); //TRUE
```





When concatenating two string literals "a"+"b" , JVM joins the two values & then check the string pool, then it realizes the value already exists in the pool so it just simply assigns this reference to the String.

### Case 5 : (+= uses StringBuilder Inside to Create & Append String)

```
String a = "Bye";
a += " bye!";

String b = "Bye bye!";

System.out.println(a == b);//FALSE
```

This case is kind of different tho, because you're using the += operator which when compiled to bytecode **it uses StringBuilder to concatenate the strings**, so this creates a new instance of StringBuilder Object thus pointing to a different reference. (string pool vs Object). Oracle Says, to improve performance, instead of using string concatenation, use `StringBuffer.append()`. String objects are immutable

### Performance

It's better to use `StringBuilder` these days, in almost every case, but here's what happens:

When you use + with two strings, it compiles code like this:

```
String third = first + second;
```

To something like this

```
StringBuilder builder = new StringBuilder( first );
builder.append( second );
third = builder.toString();
```

for example, you might be using many different appending statements, or a loop like this:

```
for( String str : strings ) {
    out += str;
}
```

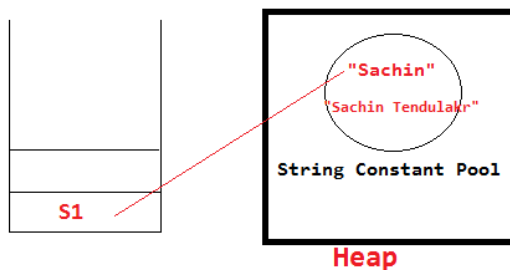
Here, a new `StringBuilder` instance, & a new `String` is required in each iteration (Strings are immutable. For each + operation it will create new String object). This is very wasteful, Replacing this with `StringBuilder` means you can just produce a single `String` and not fill up the heap with unused Strings.

To get the String literal which is created in SCP, we use `intern()` method

```
String s=new String("Welcome");
String s2=s.intern();
System.out.println(s2);// Welcome
```

In java, **string objects are immutable**. Immutable simply means unmodifiable or unchangeable.

```
class Testimmutablestring{
    public static void main(String args[]){
        String s1="Sachin";
        s.concat(" Tendulkar");//concat() method appends the string at the end
        System.out.println(s);//Sachin, because strings are immutable objects
    }
}
```



### 3,4.StringBuffer, StringBuilder Classes

String (think like it is Object type of String)	StringBuffer
String class is <b>immutable</b> .	StringBuffer class is <b>mutable</b> .
<b>String is slow and consumes more memory</b> when you concat too many strings because every time it creates new instance.	StringBuffer is fast and consumes less memory when you concat strings.
<b>String class overrides the equals()</b> method of Object class. So you can compare the contents of two strings by equals() method.	<b>StringBuffer class doesn't override the equals()</b> method of Object class.
String is <b>synchronized</b> i.e. thread safe	StringBuffer is <b>synchronized</b> i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously.

```
String a = new String("Cat");
String b = new String("Cat");
System.out.println("String : " + a.equals(b)); //true

StringBuffer b1 = new StringBuffer("Cat");
StringBuffer b2 = new StringBuffer("Cat");
System.out.println("StringBuffer : " + b1.equals(b2)); //false
```

Creates an empty StringBuffer object with default initial capacity 16. If it reaches max capacity then a new StringBuffer object will be created with new **capacity = (currentcapacity + 1) \* 2**

```
StringBuffer sb = new StringBuffer();
```

1st	--> 16	=16
2nd	--> (16+1)*2	=34
3rd	--> (34+1)*2	=70

StringBuffer	StringBuilder
StringBuffer is <b>synchronized</b> i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously.	StringBuilder is <b>non-synchronized</b> i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously.
StringBuffer is less efficient than StringBuilder.	StringBuilder is more efficient than StringBuffer.

	String	StringBuffer	StringBuilder
<b>Storage Area</b>	Constant String Pool	Heap	Heap
<b>Modifiable</b>	No (immutable)	Yes( mutable )	Yes( mutable )
<b>Thread Safe</b>	Yes	Yes	No
<b>Performance</b>	Fast	Very slow	Fast

### Examples on String, StringBuffer, StringBuilder

## 1. What is immutable object? Can you write immutable object?

### **Don't confuse over Singleton class**

Immutable classes are Java classes whose objects cannot be modified once created.

1. Declare **the class as final** so it can't be extended.
2. Make all **fields private & final** so that direct access is not allowed & its values can be assigned only once.
3. **Initialize** all the fields via a **constructor**
4. Write getters only, **not setters**.

```
// An immutable class
final class Student {
    final int sno;
    final String name;

    public Student(int sno, String name) {
        this.name = name;
        this.sno = sno;
    }
    public String getName() {
        return name;
    }
    public int getsno() {
        return sno;
    }
}

public class Test {
    public static void main(String args[]) {
        Student s1 = new Student(101, "Satya");
        s1.name = "Vijay";
    }
}
```

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
The final field Student.name cannot be assigned
```

All String & Wrapper class objects are Immutable

## 2. What is Singleton? Can you write critical section code for singleton?

A Singleton class is one which allows us to create only one object for JVM.

Rules:

- **Create Singleton class Object make it as private**, so no other classes will access it.
- **Create private constructor**, so no other classes won't create Object using new Student(), because it is not visible to out the classes.
- Every Singleton class contains **at least one static factory method**, make it public so, it is visible other classes

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

        Student s1 = Student.getObject();
        System.out.println("s1 : " + s1.hashCode());

        Student s2 = Student.getObject();
        System.out.println("s2 : " + s2.hashCode());

        System.out.println("s1==s2 : " + (s1 == s2));

    }
}

```

```

OBJECET Created FIRST TIME
s1 : 1829164700
OBJECET ALREDAY CREATED
s2 : 1829164700
s1==s2 : true

```

In above code, it will create multiple instances of Singleton class if called by more than one thread in parallel. **Double checked locking of Singleton** is a way to ensure only one instance of Singleton class is created through application life cycle.

In **double checked locking pattern**, where only critical section of code is locked. Programmers call it double checked locking because there are two checks for `_instance == null`. one is without locking and other with locking (inside synchronized) block. Here are how double-checked locking looks like in Java

```

public static Singleton getInstanceDC() {
    if (_instance == null) { // Single Checked
        synchronized (Singleton.class) {
            if (_instance == null) { // Double checked
                _instance = new Singleton();
            }
        }
    }
    return _instance;
}

```

### How do you reverse a String in Java without using StringBuffer?

The Java library provides `StringBuffer` and `StringBuilder` class with **reverse()** method, which can be used to reverse String in Java. Following code without using those classes.

```

String reverse = "";
String source= "My Name is Khan";

for(int i = source.length() -1; i>=0; i--){
    reverse = reverse + source.charAt(i);
}

```

### How to Print duplicate characters from String?

```

public class RepeatedChar {
    public static void main(String[] args) {
        String a = "success";

        // 1.convert into char array. ['s', 'u', 'c', 'c', 'e', 's', 's',]
        char[] c = a.toCharArray();

        // 2.create Hashmap store key as character, count as value
        HashMap map = new HashMap<>();
        for (char ch : c) {
            // 3.Check if Map contains given Char as <key> or not
            if (map.containsKey(ch)) {
                // if their, get the value & increment it
                int i = (int) map.get(ch);
                i++;
                // add updated value to it
                map.put(ch, i);
            }
        }
    }
}

```

```

        } else {
            // if not their , add key & value as 1
            map.put(ch, 1);
        }
    }
    Set set = map.entrySet();
    Iterator iterator = set.iterator() ;
    while (iterator.hasNext()) {
        Map.Entry entry = (Entry) iterator.next();
        System.out.println(entry.getKey()+" : "+entry.getValue());
    }
}
}
}
}
s : 3
c : 2
u : 1
e : 1

```

### How to Check given String contains Number or not

```

public class RegEx {
    public static void main(String[] args) {
        // Regular expression in Java to check if String is number or not
        Pattern pattern = Pattern.compile(".*[^\0-9].*");
        String[] inputs = { "123", "-123", "123.12", "abcd123" };
        /* Matches m = pattern.match(input);
        * boolean ch = m.match(); */
        for (String input : inputs) {
            System.out.println("does " + input + " is number : " + !pattern.matcher(input).matches());
        }

        // Regular expression in java to check if String is 6 digit number or not
        String[] numbers = { "123", "1234", "123.12", "abcd123", "123456" };
        Pattern digitPattern = Pattern.compile("\\d{6}");
        // Pattern digitPattern = Pattern.compile("\\d\\d\\d\\d\\d\\d");
        for (String number : numbers) {
            SOP("does " + number + " is 6 digit number : " + digitPattern.matcher(number).matches());
        }
    }
}

```

### Reverse Words in a String

```

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

        // using s.split("\\s");
        String s = "My name is Satya";
        String words[] = s.split("\\s");
        String rev = "";
        int len = words.length;
        for (int i = (len - 1); i >= 0; i--) {
            rev = rev + words[i];
        }
        System.out.println(rev);

        // using Collections.reverse(str)
        List<String> word = Arrays.asList(s.split("\\s"));
        Collections.reverse(word);
        System.out.println(word);
    }
}

```

## 5. Wrapper classes

Collections in Java deal only with objects; if you want to store primitive type values into any of these Collection classes, you need to Convert the primitive type to Object Type.

The main objectives of wrapper classes are:

- To Wrap primitives into object form. So that we can handle primitives also just like objects.
- To Define several utility functions for the primitives (converting primitive to the string etc.)

Each primitive type has a corresponding wrapper class.

Primitive Type	Wrapper Class
byte	Byte
short	Short
int	Int
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

### Primitive type to wrapper class

1.using **constructors**

```
// 1. using constructor
Integer object = new Integer(10);
```

2.using **static factory methods** such as `valueOf()` (except `Character`)

```
// 2. using static factory method
Integer anotherObject = Integer.valueOf(10);
```

Similarly, we can convert other primitive types like `boolean` to `Boolean`, `char` to `Character`, etc.

### Wrapper class to primitive type

Converting from wrapper to primitive type is simple. We can use the corresponding methods to get the primitive type. e.g. `intValue()`, `doubleValue()`, `shortValue()` etc.

```
Integer object = new Integer(10);
int val = object.intValue(); //wrapper to primitive
```

### Autoboxing and Unboxing

Beginning with JDK 5, Java added two important features: *autoboxing* and *auto-unboxing*.

**Autoboxing** is the automatic conversion of the primitive types into their corresponding object wrapper classes. For example, converting an `int` to an `Integer`, a `char` to a `Character`, and so on. We can simply pass or assign a primitive type to an argument or reference accepting wrapper class object.

```
List<Integer> integerList = new ArrayList<>();

for (int i = 1; i < 50; i += 2)
{
    integerList.add(i); //autoboxing - Automatically converts int to Integer
}
}
```

**Unboxing** happens when the conversion happens from wrapper class to its corresponding primitive type. It means we can pass or assign a wrapper object to an argument or reference accepting primitive type.

```
public static int sumOfEven(List<Integer> integerList)
{
    int sum = 0;
    for (Integer i: integerList) {
        if (i % 2 == 0)
            sum += i;           //Integer to int
    }
    return sum;
}
```

In above example, the remainder (%) and unary plus (+=) operators do not apply on Integer objects. The compiler automatically converts an Integer to an int at runtime by invoking the `intValue()` method.

```
Integer i = 10; // it will create Integer value of 10 using Autoboxing
int j = i; // it will convert Integer to int using Unboxing
```

### Wrapper Classes Internal Caching

**Wrapper classes are immutable in java, Right? "YES"**. So, like string pool, they can also have their pool, right? "Great Idea". Well, it's already there. JDK provided wrapper classes also provide this in form of instance pooling i.e. each wrapper class store a list of commonly used instances of own type in form of cache and whenever required, you can use them in your code. It saves memory on your program runtime.

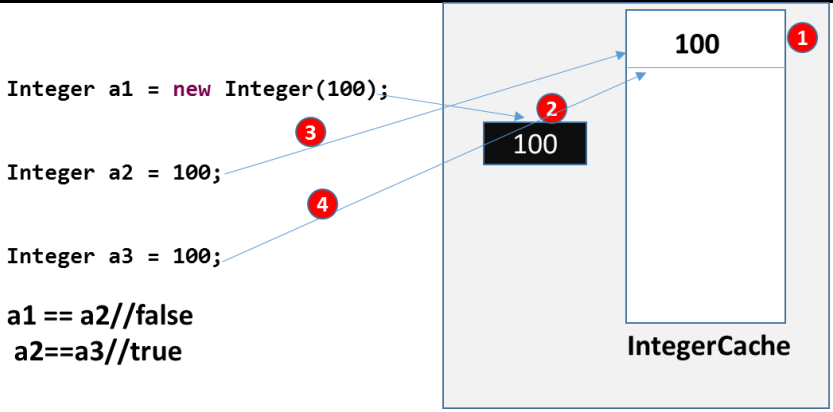
In **Integer.java** class, there is an inner class i.e. **IntegerCache**. When you assign a new int to Integer type like below

```
Integer i = 10; //OR
Integer i = Integer.valueOf(10);
```

An already created Integer instance is returned and reference is stored in `i`. Please note that if you use `new Integer(10)`; then a new instance of Integer class will be created, and caching will not come into picture. Its only available when you use `Integer.valueOf()` OR directly primitive assignment (which ultimately uses `valueOf()` function)

```
Integer a1 = new Integer(100);
Integer a2 = 100;
Integer a3 = 100;
System.out.println(a1==a2); //False
System.out.println(a2==a3); //True

int a4 = 100;
int a5 = 100;
System.out.println(a4==a5); //True
```



If you want to store a bigger number of instances, you can use runtime parameter as below:

```
-Djava.lang.Integer.IntegerCache.high=2000
```

# Garbage collection

- In C/C++, programmer is responsible for both creation and destruction of objects. Usually programmer neglects destruction of useless objects. Due to this negligence, at certain point enough memory may not be available for creation of new objects and entire program will terminate abnormally causing **OutOfMemoryErrors**.
- But in Java, the programmer need not to care for all those objects which are no longer in use. Garbage collector destroys these objects.
- Garbage collector is best example of Daemon thread as it is always running in background.
- Main objective of Garbage Collector is to free heap memory by destroying **unreachable objects**

## The ways to make an object eligible for Garbage Collector

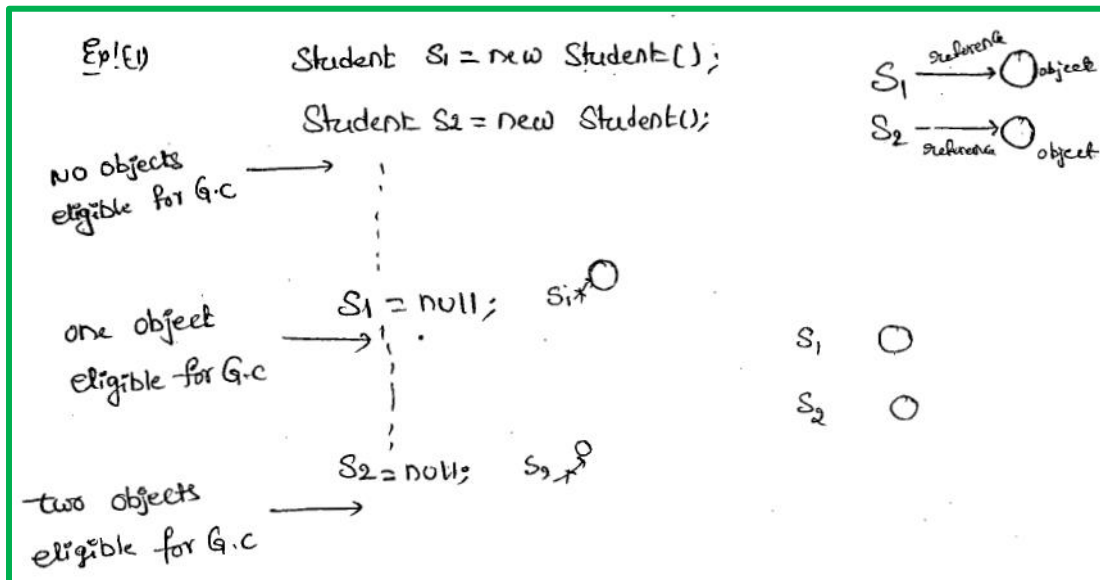
Even though the programmer is not responsible for destruction of objects it's good programming practice to make an object eligible for Garbage Collector if it is no longer required.

The following are different ways for this

1. Nullifying the reference Variable
2. Reassigning the reference Variable
3. The Objects Created inside a method
4. The Island of Isolation
5. Static variables Garbage Collection

### 1.Nullifying the reference Variable

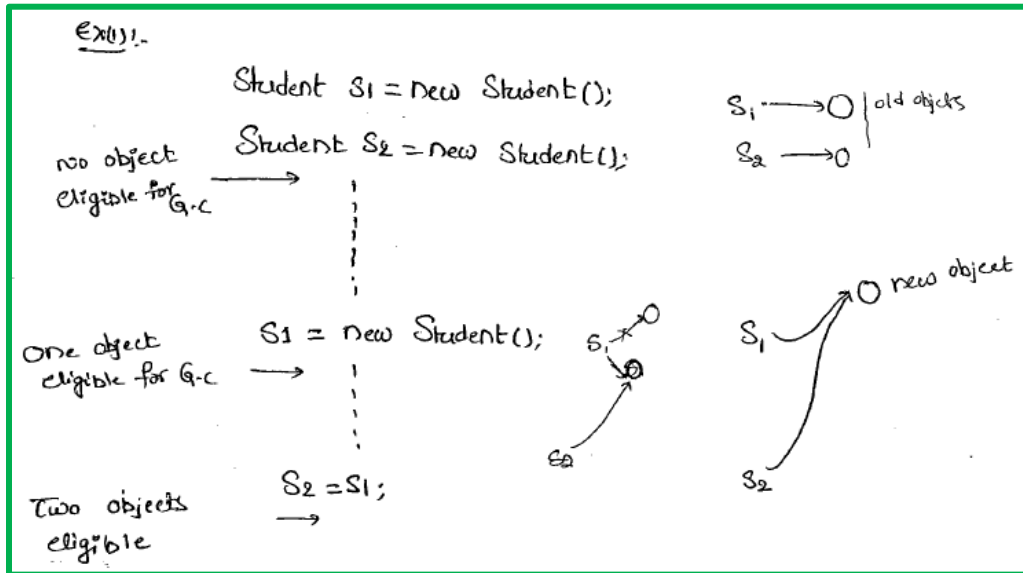
If an object is no longer required assign **null** to all its reference variables.





## 2. Reassigning the reference Variable

If an object is no longer required then, assigning its reference variables to some other objects then, that old object automatically eligible for garbage collection.



## 3.The Objects Created inside a method

The objects which are created in a **method** are by default eligible for Garbage Collector, once the method execution completes.

### Case 1:

```
class Test
{
    public static void main(String arg[])
    {
        m1();
        //Two Objects s1,s2 eligible for gc
    }

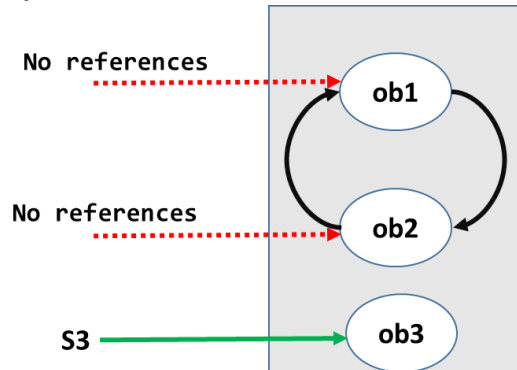
    public static void m1()
    {
        Student s1 = new Student();
        Student s2 = new Student();
        //No Objects eligible for gc
    }
}
```

### Case 2:

```
class Test {
    public static void main(String arg[]) {
        Student s = m1();
        //One Object- s2 eligible for gc
    }
    public static Student m1() {
        Student s1 = new Student();
        Student s2 = new Student();
        return s1; // returning s1
    }
}
```

#### 4. The Island of Isolation

**Obj1** references **Obj2** and **Obj2** references **Obj1**. Neither Object 1 nor Object 2 is referenced by any other object. That's an island of isolation.



Basically, an island of isolation is a group of objects that reference each other but they are not referenced by any active object in the application. Strictly speaking, even a single unreferenced object is an island of isolation too.

```
class Test
{
    Test i;

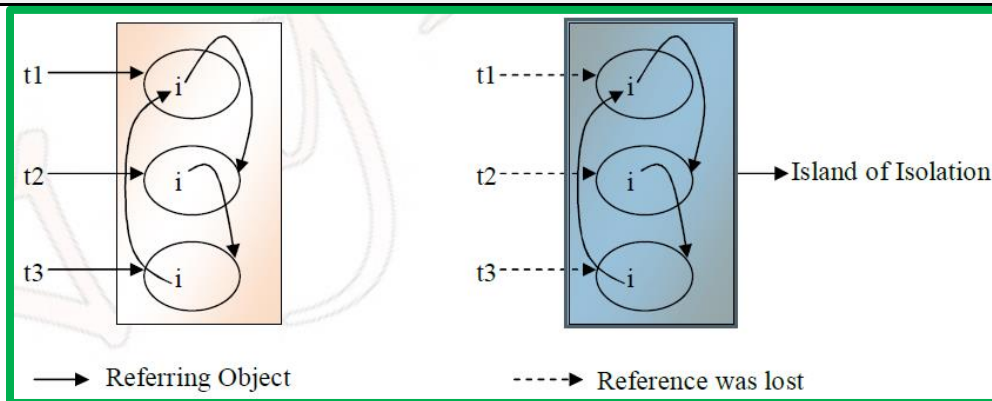
    public static void main(String[] args)
    {
        Test t1 = new Test();
        Test t2 = new Test();
        Test t3 = new Test();

        t1.i = t2;
        t2.i = t3;
        t3.i = t1;

        t1 = null;
        //No Object eligible for Garbage Collector

        t2 = null;
        //No Object eligible for Garbage Collector

        t3 = null;
        //All Objects eligible for Garbage Collector
    }
}
```



#### 5. Static variables Garbage Collection

Static variables cannot be eligible for garbage collection. They can be garbage collected when the respective class loader drops the class or is itself collected for garbage.

## What are the methods to request JVM to run Garbage Collector?

We can request JVM to run Garbage Collector but **there is no guarantee** whether JVM accepts our request or not. We can do this by using the following ways.

### 1. By System class

'System' class contains a static 'gc' method for requesting JVM to run Garbage Collector.

```
System.gc();
```

### 2. By Using Runtime Class

A java application can communicate with JVM by using Runtime class Object. We can get Runtime Object as follows.

```
Runtime runtime = Runtime.getRuntime();
runtime.gc();
```

Once we get Runtime Object, we can apply the following methods on that object.

- **freeMemory():** returns the free memory available in the loop
- **totalMemory():** returns heap size
- **gc():** for requesting JVM to run Garbage Collector

**gc() method available in the System class is static method, but gc() method available in Runtime class is an instance method.**

## finalize()

- Just before destroying any object, Garbage Collector always calls **finalize()** to perform cleanup activities.
- **finalize()** is available in the **Object** class which is declared as follows.

```
protected void finalize() throws Throwable
{
}
```

**case1:** Garbage Collector always calls **finalize()** on the Object which is eligible for Garbage Collection and the corresponding class finalize method will be executed.

```
class Test {
    public static void main(String arg[]) {
        String s = new String("raju");
        //Test s = new Test();
        s = null;
        System.gc();
        System.out.println("end of main method");
    }

    public void finalize() {
        System.out.println("finalize method called");
    }
}
```

O/P:- end of main method.

In this case String Object is eligible for G.C and hence String class **finalize()** method has been executed. In the above program if we are replacing String Object with Test Object then Test class **finalize()** will be executed. In this case **O/P** is end of main method.

```
finalize method called
end of main method
```

**case2:** we can call **finalize()** explicitly in that case it will execute just like a normal method and object won't be destroyed.

While executing `finalize()` method if any exception is uncaught, it is simply ignored by the JVM but if we are calling finalize method explicitly and if an exception is uncaught, then the program will be terminated abnormally.

```

class Test {
    public static void main(String arg[]) {
        Test s = new Test();
        //s.finalize();
        s = null;
        System.gc();
        System.out.println("End of main method");
    }

    public void finalize() {
        System.out.println("finalize method");
        System.out.println(10 / 0);
    }
}

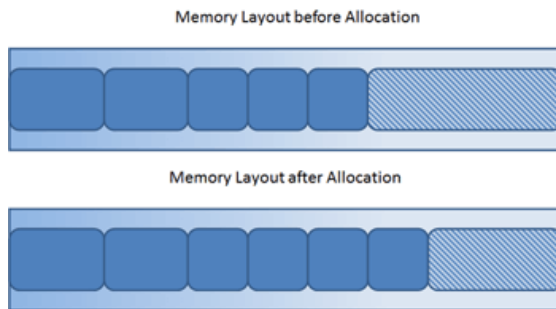
```

O/P:- finalize method  
end of main method

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

### Types of Garbage Collectors

When an object is no longer used, the garbage collector reclaims the underlying memory and reuses it for future object allocation. This means there is no explicit deletion, and no memory is given back to the operating system.



Java has **four types of garbage collectors**,

- **Serial Garbage Collector**
- **Parallel Garbage Collector**
- **CMS Garbage Collector** (Concurrent Mark & Sweep)
- **G1 Garbage Collector**

Each of these four types has its own advantages and disadvantages. Most importantly, we the programmers can choose the type of garbage collector to be used by the JVM. We can choose them by passing the choice as **JVM argument**

## 1. Serial Garbage Collector

- It is designed for **the single-threaded environments**.
- It **uses just a single thread for garbage collection**.
- It freezes(stops) all the application threads while performing garbage collection.
- it may not be suitable for a server environment.
- It is best suited for simple command-line programs.

Turn on the **-XX:+UseSerialGC** JVM argument to use the serial garbage collector.

## 2. Parallel Garbage Collector

- It is the **default garbage** collector of the JVM.
- It uses **multiple threads for garbage collection**.
- Similar to serial garbage collector this also freezes(stops) all the application threads while performing garbage collection.

## 3. CMS Garbage Collector

Concurrent Mark & Sweep (CMS) garbage collector uses **multiple threads to scan the heap memory** and **mark instances** which are eligible for garbage collection and **then sweep the marked instances**. Turn on the **XX:+UseParNewGC** JVM argument to use the CMS garbage collector.

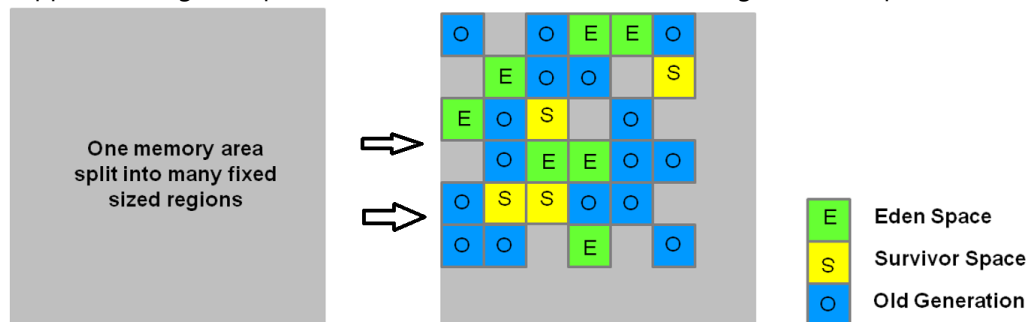
## 4. G1 Garbage Collector

- G1 garbage collector is used if we have a large (more than 4GB) heap space.
- It divides the heap into equal-sized (usually 1MB to 32MB) **regions** & will **prioritize** them.
- It performs the parallel garbage collection on that **region** based on the **priority**.

### The G1 Garbage Collector Step by Step

#### G1 Heap Structure

The heap is one memory area split into many fixed sized regions. Region size is chosen by the JVM at startup. The JVM generally targets around 2000 regions varying in size from 1 to 32Mb. These regions are mapped into logical representations of Eden, Survivor, and old generation spaces.



Before Split

After Split

Live objects are evacuated (i.e., copied or moved) from one region to another.

Regions are designed to be collected in parallel with or without stopping all other application threads.

As shown regions can be allocated into **Eden, survivor, and old generation regions**. In addition, there is a fourth type of object known as Humongous regions. These regions are designed to hold objects that are 50% the size of a standard region or larger. They are stored as a set of contiguous regions. Finally, the last type of regions would be the unused areas of the heap.

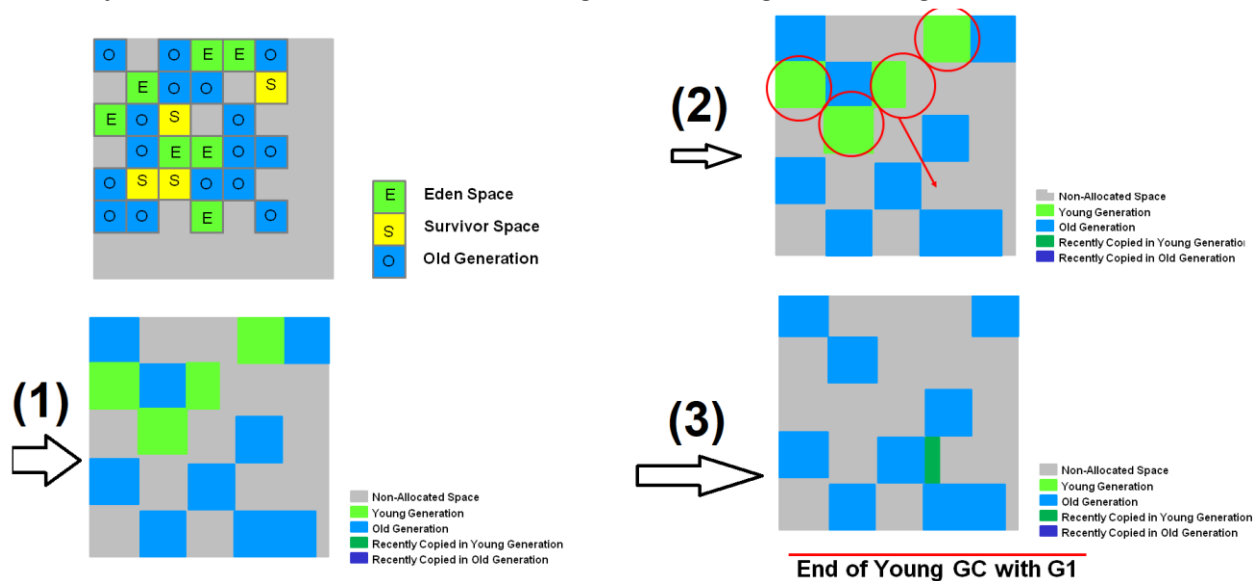
### 1.Young Generation in G1

1.The heap is split into approximately 2000 regions. Minimum size is 1Mb and maximum size is 32Mb.

Blue regions hold old generation objects and green regions hold young generation objects.

2.Live objects are evacuated (i.e., copied or moved) to one or more survivor regions. If the aging threshold is met, some of the objects are promoted to old generation regions.

3.Live objects have been evacuated to survivor regions or to old generation regions.



Recently promoted objects are shown in dark blue. Survivor regions in green.

In summary, the following can be said about the young generation in G1:

- The heap is a single memory space split into regions.
- Young generation memory is composed of a set of non-contiguous regions. This makes it easy to resize when needed.
- Young generation garbage collections, or young GCs, are stop the world events. All application threads are stopped for the operation.
- The young GC is done in parallel using multiple threads.
- Live objects are copied to new survivor or old generation regions.

### Old Generation Collection with G1

Like the CMS collector, the G1 collector is designed to be a low pause collector for old generation objects. The following table describes the G1 collection phases on old generation.

#### G1 Collection Phases - Concurrent Marking Cycle Phases

The G1 collector performs the following phases on the old generation of the heap. Note that some phases are part of a young generation collection.

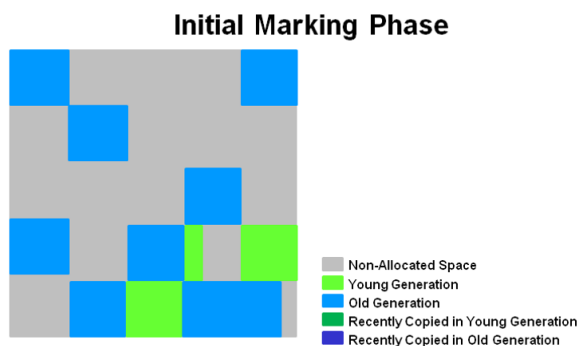
Phase	Description
(1) Initial Mark <i>(Stop the World Event)</i>	This is a stop the world event. With G1, it is piggybacked on a normal young GC. Mark survivor regions (root regions) which may have references to objects in old generation.
(2) Root Region Scanning	Scan survivor regions for references into the old generation. This happens while the application continues to run. The phase must be completed before a young GC can occur.
(3) Concurrent Marking	Find live objects over the entire heap. This happens while the application is running. This phase can be interrupted by young generation garbage collections.
(4) Remark <i>(Stop the World Event)</i>	Completes the marking of live object in the heap. Uses an algorithm called snapshot-at-the-beginning (SATB) which is much faster than what was used in the CMS collector.
(5) Cleanup <i>(Stop the World Event and Concurrent)</i>	<ul style="list-style-type: none"> <li>○ Performs accounting on live objects and completely free regions. (Stop the world)</li> <li>○ Scrubs the Remembered Sets. (Stop the world)</li> <li>○ Reset the empty regions and return them to the free list. (Concurrent)</li> </ul>
(*) Copying <i>(Stop the World Event)</i>	These are the stop the world pauses to evacuate or copy live objects to new unused regions. This can be done with young generation regions which are logged as [GC pause (young)]. Or both young and old generation regions which are logged as [GC Pause (mixed)].

## G1 Old Generation Collection Step by Step

With the phases defined, let's look at how they interact with the old generation in the G1 collector.

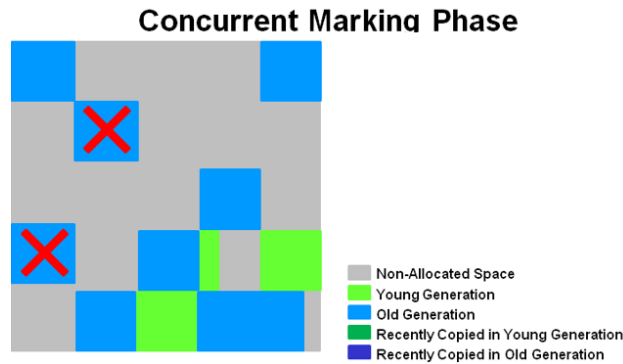
### Initial Marking Phase

Initial marking of live object is piggybacked on a young generation garbage collection. In the logs this is noted as GC pause (young)(initial-mark).



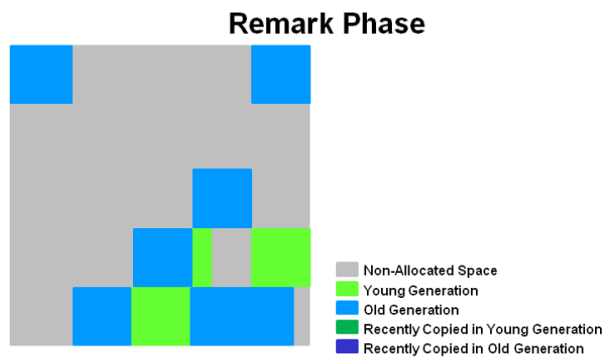
### Concurrent Marking Phase

If empty regions are found (as denoted by the "X"), they are removed immediately in the Remark phase. Also, "accounting" information that determines liveness is calculated.



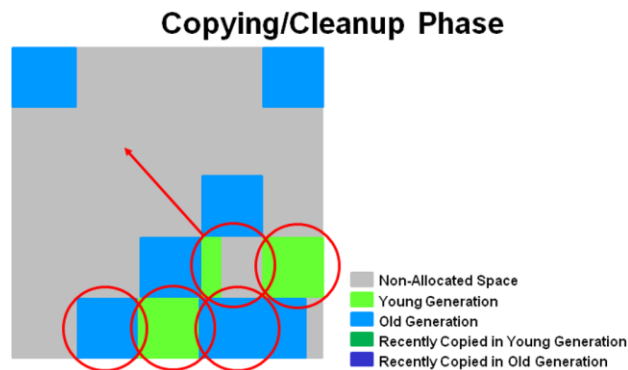
### Remark Phase

Empty regions are removed and reclaimed. Region liveness is now calculated for all regions.



### Copying/Cleanup Phase

G1 selects the regions with the lowest "liveness", those regions which can be collected the fastest. Then those regions are collected at the same time as a young GC. This is denoted in the logs as [GC pause (mixed)]. So both young and old generations are collected at the same time.

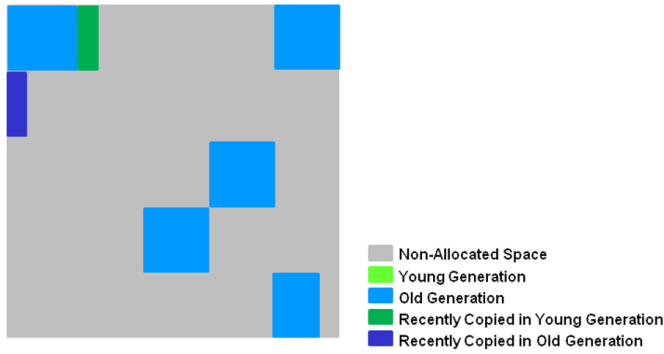


### After Copying/Cleanup Phase

The regions selected have been collected and compacted into the dark blue region and the dark green region shown in the diagram.



## After Copying/Cleanup Phase



### Summary of Old Generation GC

In summary, there are a few key points we can make about the G1 garbage collection on the old generation.

- **Concurrent Marking Phase**
  - Liveness information is calculated concurrently while the application is running.
  - This liveness information identifies which regions will be best to reclaim during an evacuation pause.
  - There is no sweeping phase like in CMS.
- **Remark Phase**
  - Uses the Snapshot-at-the-Beginning (SATB) algorithm which is much faster than what was used with CMS.
  - Completely empty regions are reclaimed.
- **Copying/Cleanup Phase**
  - Young generation and old generation are reclaimed at the same time.
  - Old generation regions are selected based on their liveness.

### Java 8 Improvement

Turn on the `-XX:+UseStringDeduplication` JVM argument while using G1 garbage collector. This optimizes the heap memory by removing duplicate String values to a single char[] array. This option is introduced in [Java 8 u 20](#).

Given all the above four types of Java garbage collectors, which one to use depends on the application scenario, hardware available and the throughput requirements.

## Garbage Collection JVM Options

### Type of Garbage Collector to run

Option	Description
<code>-XX:+UseSerialGC</code>	Serial Garbage Collector
<code>-XX:+UseParallelGC</code>	Parallel Garbage Collector
<code>-XX:+UseConcMarkSweepGC</code>	CMS Garbage Collector
<code>-XX:ParallelCMSThreads=</code>	CMS Collector – number of threads to use
<code>-XX:+UseG1GC</code>	G1 Garbage Collector

## GC Optimization Options

Option	Description
-Xms	Initial heap memory size
-Xmx	Maximum heap memory size
-Xmn	Size of Young Generation
-XX:PermSize	Initial Permanent Generation size
-XX:MaxPermSize	Maximum Permanent Generation size

## 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**.

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

## Interview Questions

### Can a top-level class be private or protected?

**No** - Top level classes in java can't be private or protected, but inner classes in java can. The reason for not making a top-level class as private is very obvious, because nobody can see a private class and thus, they cannot use it.

### What Happens if we compile Empty java file?

Compiles but Runtime Error.

```
C:\Users\kaveti_S\Downloads>javac Demo.java
C:\Users\kaveti_S\Downloads>java Demo
Error: Could not find or load main class Demo
```

### Is it possible to make array volatile in Java?

Yes, it is possible to make an array volatile in Java, but only the reference, which is pointing to an array, by reassigning it.

### What is a.hashCode() used for? How is it related to a.equals(b)?

HashCode is derived from memory location. According to the Java specification, two objects which are identical to each other using equals() method needs to have the same hash code

### Explain Liskov Substitution Principle.

According to the Liskov Substitution Principle, methods or functions which use super class type must be able to work with object of subclass without issues. **Co-Variant** return types are implemented based on this principle.

### What is a compile time constant in Java? What is the risk of using it?

**public static final variables** are also known as the compile time constants, the **public** is optional there. **They are substituted with actual values at compile time because compiler recognizes their value up-front, and also recognize that it cannot be altered during runtime.**

One of the issues is that if you choose to use a **public static final** variable from in-house or a third-party library is, and their value changed later, then your client will still be using the old value even after you deploy a new version of JARs.

### What is double checked locking in Singleton?

**Singleton** means we can create only one instance of that class

#### Rules:

- Create Singleton class Object make it as **private**
- Create **private** constructor
- 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("OBJECT ALREADY 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());
    }
}

```

**Double checked locking in Singleton means**, at any cost only one instance is created in multi-threaded environment. In this case at **null** checking make Block as Synchronized.

```

public static Singleton getInstanceDC() {
    if (_instance == null) { // Single Checked
        synchronized (Singleton.class) {
            if (_instance == null) { // Double checked
                _instance = new Singleton();
            }
        }
    }
    return _instance;
}
}

```

### When to use volatile variable in Java?

- Volatile keyword is used with only variable in Java
- it guarantees that value of volatile variable will always be read from main memory and not from Thread's local cache.
- So, we can use volatile to achieve synchronization because it's guaranteed that all reader thread will see updated value of volatile variable once write operation completed

### Difference between static and dynamic binding in Java? (detailed answer)

**static binding is related to overloaded method** and **dynamic binding is related to overridden method**. Method like private, final and static are resolved using static binding at compile time but virtual methods which can be overridden are resolved using dynamic binding at runtime.

### Which design pattern have you used in your production code?

- **Dependency injection** -in Spring
- **Factory pattern** - Connection Object
- **Adapter Design pattern** - DAO Interface Implementations
- **Singleton** - in Spring
- **Template Design Pattern** - JDBCTemplate, HibernateTemplate

**Decorator design pattern** is used to modify the functionality of an object at runtime.

### How to create an instance of any class without using new keyword

Using `newInstance()` method of Class class

```

Class c = Class.forName("StudentBo");
StudentBo bo = (StudentBo) c.newInstance();

```

Using `clone()` of `java.lang.Object`

```

NewClass obj = new NewClass();
NewClass obj2 = (NewClass) obj.clone();

```

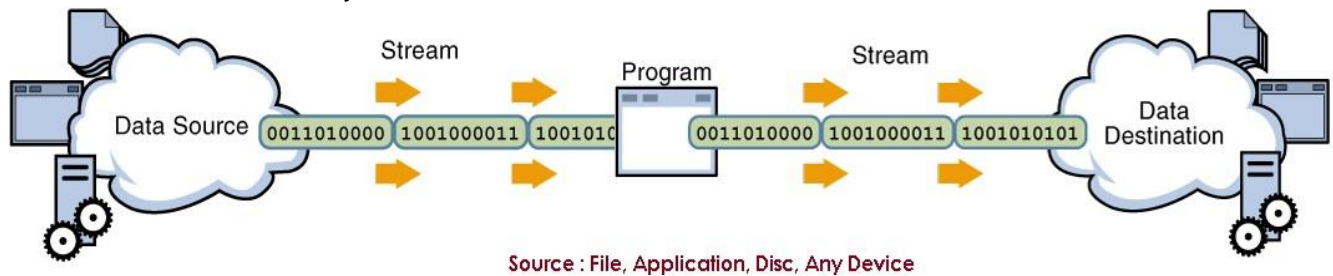
## 7. java.io

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

In java we will write two types of programs

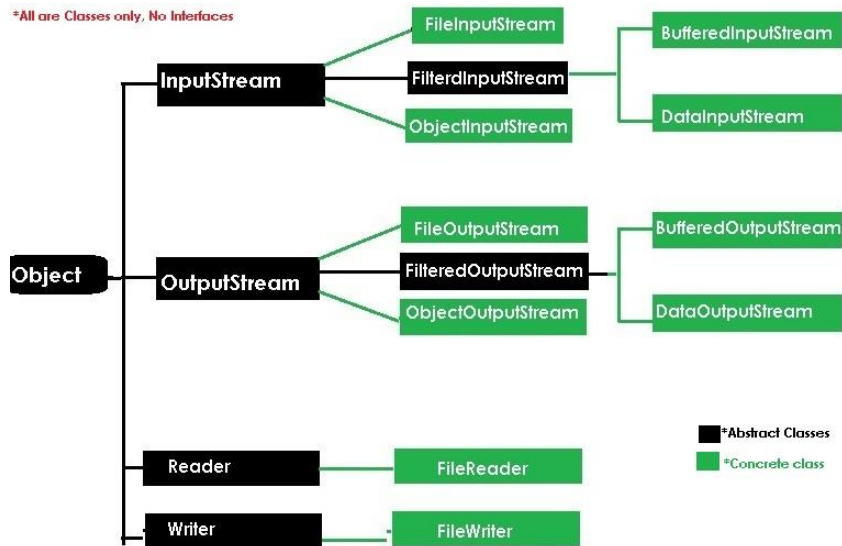
1. **volatile programs:** whose result are stored in main Memory (RAM), Temporally (Ex. Console Applications)
2. **Non-Volatile programs:** whose results are saved permanently in secondary memory like Drives, Hard disks, Databases & files.

**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

- `public int read();`
- `public int length();` // total size of the file
- `public int available();` // available number of bytes only
- `public void close();`

### 2. OutputStream: Write data to file/Destination

- `public void write (int);`
- `public int length ();`
- `public void available ();`
- `public void close ();`

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

## Byte Streams

- Data transfer is **one byte at a time** from source to destination
- used to read byte-oriented data for example **to read image, audio, video etc.**

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

Constructors	Methods
<pre>FileInputSteam(File file) FileInputSteam(String FilePath)</pre>	<pre>Int read(byte b[]) Int read(byte[] b, int off, int len)</pre>

### 2. FileOutputStream

Constructors	Methods
<pre>FileOutputStream(File file) FileOutputStream(String filepath) FileOutputStream(File file, boolean append) //true →append, false→ overwrite FileOutputStream(String name, boolean append) name.</pre>	<pre>void write(byte b[]) void write(byte[] b, int off, int len) //only byte type is allowed. (Sure checked)</pre>

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); //using write(int) of OutputStream Interface
        }

        FileInputStream inputStream = new FileInputStream(filepath);
        int i;
        while ((i = inputStream.read()) != -1) { //read() returns int
            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 

## 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 use 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); //accepts char type only
        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 

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

## 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

```
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 operations. When we create an object of **BufferedInputStream**, we get a temporary memory space whose **default size is 1024 bytes**, and it can be increased by multiple of 2.

### 2. BufferedOutputStream:

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

Constructors	Methods
<b>BufferedInputStream(InputStream is)</b> <b>BufferedInputStream(InputStream is, int bufsize)</b>	Int read(byte b[]) Int read(byte[] b, int off, int len)
<b>BufferedOutputStream(OutputStream os)</b> <b>BufferedOutputStream(OutputStream os, int bufsize)</b>	void write(byte b[]) void write(byte[] b, int off, int len) //only byte type is allowed. (Sure checked)

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 -Programming Simplified";
        byte[] b = s.getBytes();
        bos.write(b);
        bos.flush();

        // 1.Create Stream Object
        FileInputStream fis = new FileInputStream(filepath);
    }
}
```



```

// 2.pass Stream object to BufferedStream 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 → Allows **byte(int)** datatype only
- Character streams will transfer 2 bytes of data at a time → Allows **char(int)** type only
- Buffered Streams will transfer 1024 bytes of data at a time → Allows **Byte/Char** type

## Data Streams

In Previous InputStream allowed only **int type**, Byte Stream allowed only **byte[]** & CharacterStream allowed only **char[]** for writing data.

To work with Other Datatypes, Data streams introduced to 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. **DataOutputStream**: Used for write primitive Java data types to Output stream. (**writeXXX()** method)  
here XXX = primitive data types

Constructors	Methods
<b>DataInputStream (InputStream is)</b>	Int read(byte b[]) Int read(byte[] b, int off, int len) Byte readByte() Int readInt() Char readchar()
<b>DataOutputStream (OutputStream os)</b>	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

```

## Object Streams

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

<code>ObjectOutputStream</code> ( <code>OutputStream out</code> ) void <code>writeObject</code> ( <code>Object obj</code> )	<code>ObjectInputStream</code> ( <code>InputStream in</code> ) Object <code>readObject</code> ()
--	---

### 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 if it won't implement
    Serializable
    private int sno;
    private String name;
    private String addr;
    //Setters & getters 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);
    }
}

//data saved in student.txt
-i sr
io.Student0p@(!°| _____ I _____ snoL _____
addrL java/lang/String;L _____
nameq ~ xp et
VIJAYAWADAT
```

## 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

```
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.

### Transient Keyword

To avoid the variable from the serialization process, make that variable declaration as transient i.e., **transient variables never participate in serialization process.** Default values will be initialized for transient variables.

```
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.

### 3. Externalization

The default serialize object is **heavy weight** & having lots of attributes and properties, that you do want to serialize for any reason (e.g. they always been assigned default values even if we use **transient** keyword), you get heavy object to process and send more bytes over network which may be costly on some cases.

To **customize your serialization mechanism**, we can use Externalization. **Externalizable interface extends Serializable interface**. If you implement this interface, then you need to override following methods with the fields which you want to serialize.

```
public void readExternal(ObjectInput arg0) throws IOException,
public void writeExternal(ObjectOutput arg0) throws IOException
```

**Example: I'm a Student , I don't want to save my GF data.**

```
class Student implements Externalizable {

    private int sno;
    private String name;
    // I don't want save my GF data
    private String girlFriend;
    // getters & setters

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

    @Override
    public void readExternal(ObjectInput input) throws IOException, ClassNotFoundException {
        sno = input.readInt();
        name = input.readUTF();// String
    }

    @Override
    public void writeExternal(ObjectOutput output) throws IOException {
        output.writeInt(sno);
        output.writeUTF(name);
    }

    @Override
    public String toString() {
        return "Student [sno=" + sno + ", name=" + name + ", girlFriend=" + girlFriend + "];"
    }
}

public class Test {
    public static void main(String args[]) throws Exception {
        // Writing data
        FileOutputStream fos = new FileOutputStream("student.txt");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(new Student(101, "Satya"));

        // Reading data
        FileInputStream fis = new FileInputStream("student.txt");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Student s = (Student) ois.readObject();
        System.out.println(s.toString());
    }
}

Student [sno=101, name=Satya, girlFriend=null]
```

<https://howtodoinjava.com/java/serialization/java-externalizable-example/>

**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();
```

## printf and format Methods

The java.io package includes a PrintStream class that has two formatting methods. **format** and **printf**

**public PrintStream format (String format, Object... args)**

```
System.out.format("The value of " + "the float variable is " +
    "%f, while the value of the " + "integer variable is %d, " +
    "and the string is %s", floatVar, intVar, stringVar);
```

## Java NIO(**Non-blocking I/O**)-1.4

The `java.nio.file` provide support for file I/O and for accessing the default file system.

### 1) IO streams versus NIO blocks

- NIO provides high-speed, block-oriented I/O. original I/O deals with **data in streams**, whereas NIO deals with **data in blocks**.
- A block-oriented I/O system deals with data in blocks. Each operation produces or consumes a block of data in one step. Processing data by the block can be much faster than processing it by the (streamed) byte

### 2) Synchronous vs. Asynchronous IO

- Java IO's various streams are blocking or synchronous. That means, when a thread invokes a read() or write(), that thread is blocked until there is some data to read, or the data is fully written.
- In asynchronous IO, a thread can request that some data be written to a channel, but not wait for it to be fully written. The thread can then go on and do something else in the meantime. Usually, these threads spend their idle time on when not blocked in IO calls, is usually performing IO on other channels in the meantime. That is, a single thread can now manage multiple channels of input and output

```
public class ReadFileWithFixedSizeBuffer
{
    public static void main(String[] args) throws IOException
```

```

{
    RandomAccessFile aFile = new RandomAccessFile("test.txt", "r");
    FileChannel inChannel = aFile.getChannel();

    ByteBuffer buffer = ByteBuffer.allocate(1024);
    while(inChannel.read(buffer) > 0)
    {
        buffer.flip();
        for (int i = 0; i < buffer.limit(); i++)
        {
            System.out.print((char) buffer.get());
        }
        buffer.clear(); // do something with the data and clear/compact it.
    }

    inChannel.close();
    aFile.close();
}
}

```

IO	NIO
It is based on the Blocking I/O operation	It is based on the Non-blocking I/O operation
It is Stream-oriented	It is Buffer-oriented
Channels are not available	Channels are available for Non-blocking I/O operation
Selectors are not available	Selectors are available for Non-blocking I/O operation

### Blocking vs. Non-blocking I/O

**Blocking I/O**

Blocking IO wait for the data to be write or read before returning. Java IO's various streams are blocking. It means when the thread invoke a write() or read(), then the thread is blocked until there is some data available for read, or the data is fully written.

**Non-blocking I/O**

Non-blocking IO does not wait for the data to be read or write before returning. Java NIO non- blocking mode allows the thread to request writing data to a channel, but not wait for it to be fully written. The thread is allowed to go on and do something else in a mean time

# 8. Threads

## 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 exists single flow of controls whereas the languages like **JAVA, .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 threads (main Thread)** are those which are **executing user defined sub-programs**. There is a possibility of creating 'n' number of fore ground threads
2. **Background threads** are those which are **monitoring the status of fore ground thread**. And always there exist **single background thread**.

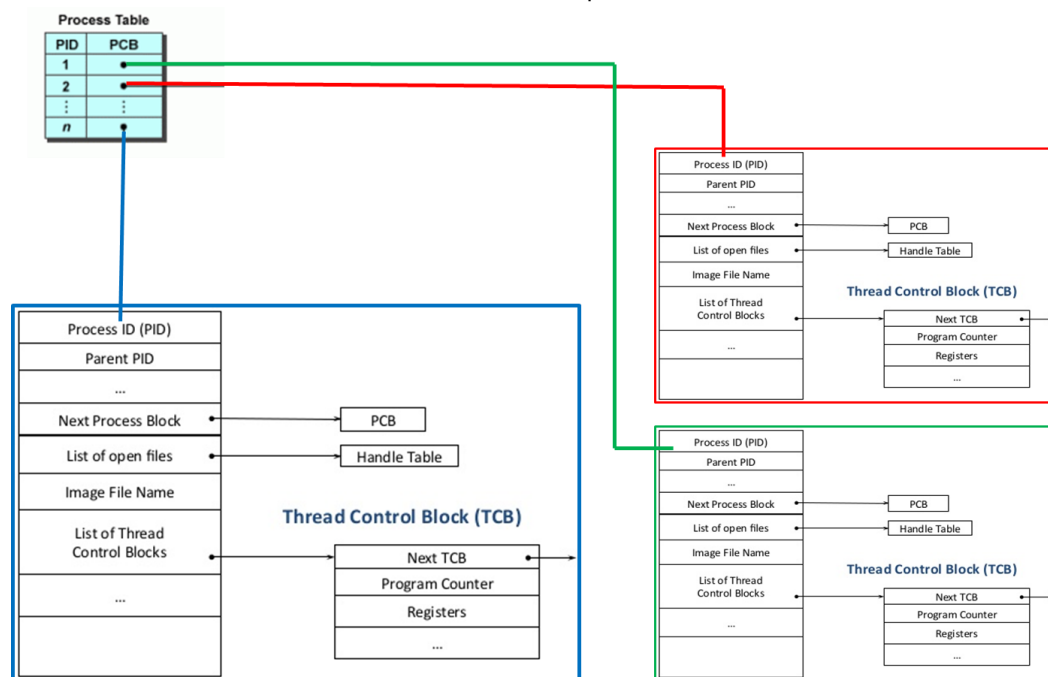
In information technology we can develop two types of applications. They are **process based applications** and **thread-based applications**.

### Context Switch

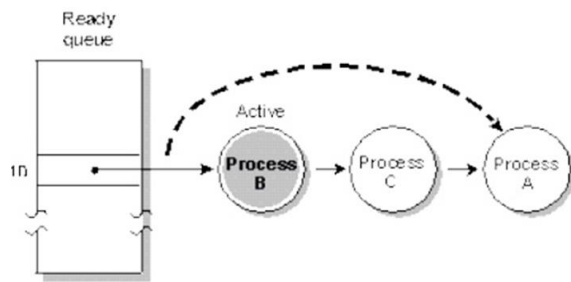
#### 1.Context Switch in Process

Single flow of Execution is process. Each process has Process Control Block (**PCB**), the state of the process is represented in PCB by the operating system.

Whenever CPU want to execute another process it will save the State of current process in **PCB (Process Control Block)** with their PID and loads the new process to execute.



Here **Context Switch time is very high** because **Process** are running two different address spaces.



## 2. Control Switch in Threads

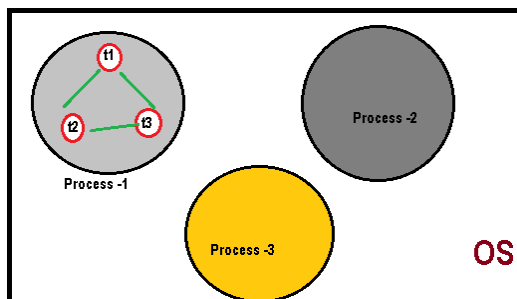
A thread is a sequential execution stream within a process. This means that a single process may be broken up into multiple threads. Each thread has its own **Program Counter, registers, and stack**, but **they all share the same address space within the process**.

When we switch between two threads all threads share the same address space cost of switching between threads is much smaller than the cost of switching between processes.

Process Based Applications	Thread Based Applications
<ol style="list-style-type: none"> <li>1. Exist <b>single flow of control</b>.</li> <li>2. All <b>C, C++</b> applications comes under it.</li> <li>3. <b>Context switch is more</b>.</li> <li>4. Each process have <b>its own address in memory</b> i.e. each process allocates separate memory area.</li> <li>5. These are treated as <b>heavy weight</b> components.</li> <li>6. In this we can achieve only sequential execution and they <b>are not recommending for developing internet applications</b>.</li> </ol>	<ol style="list-style-type: none"> <li>1. Exist <b>Multiple flow of controls</b>.</li> <li>2. All <b>JAVA, .NET</b> applications come under it.</li> <li>3. <b>Context switch is very less</b>.</li> <li>4. Threads share the <b>same address space</b></li> <li>5. These are treated as <b>light weight</b> components.</li> <li>6. In thread-based applications we can achieve both sequential and concurrent execution and they are always <b>recommended for developing internet applications</b>.</li> </ol>

## 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, shares a common memory area. if there occurs exception in one thread, it doesn't affect other threads.



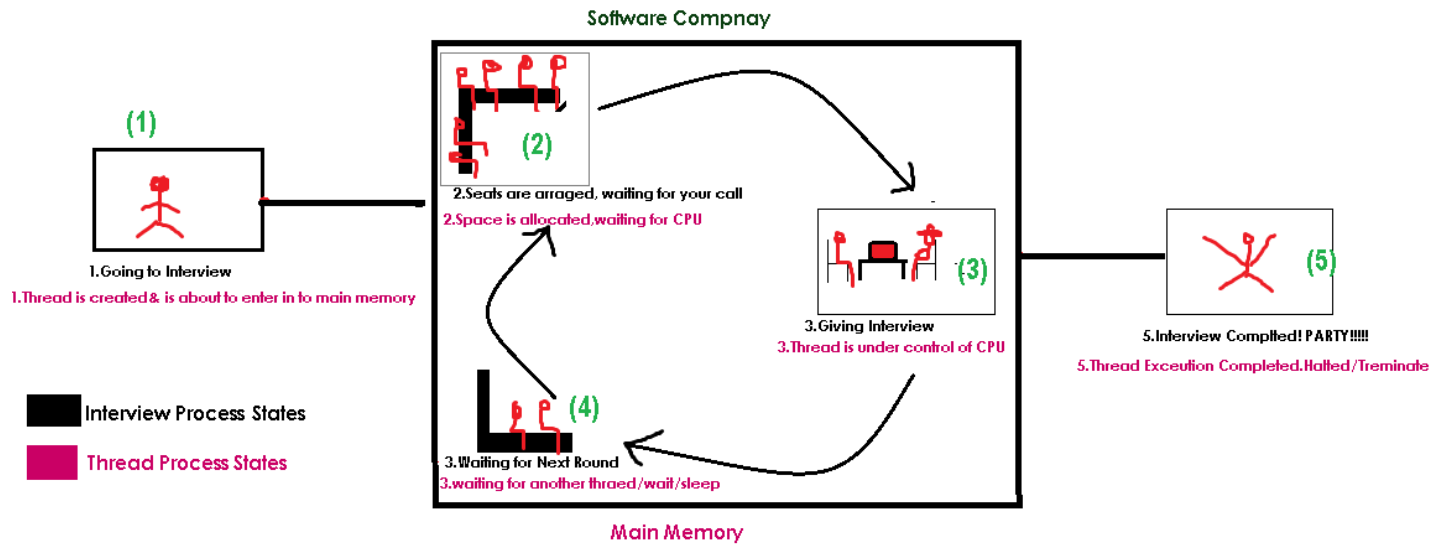
As shown in the figure, thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS and one process can have multiple threads.

**At a time one thread is executed only.**



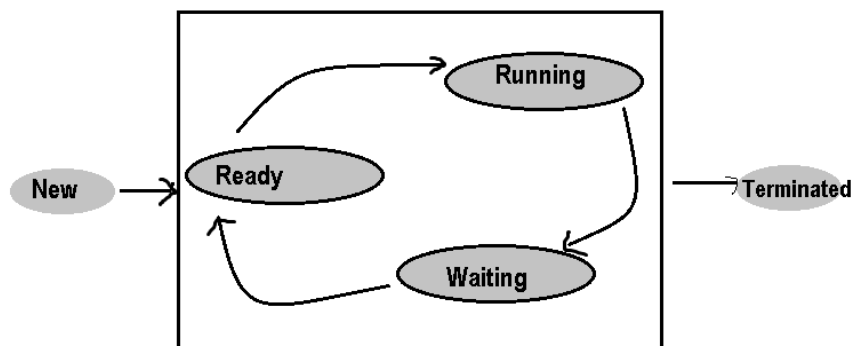
# Thread Life Cycles (Thread States)

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. Seated, waiting for your call	2. Space is allocated, waiting for CPU
3. giving interview	3. Thread is under control of CPU
4. waiting 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 – before execution of `run()`**, 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 it 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 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`.

## 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
<code>Thread()</code>	Creates new Thread, whose default thread name is <b>Thread-0</b>
<code>Thread(String name)</code>	Creates new Thread, with user defined thread name
<code>Thread(Runnable r)</code>	Used for converting Runnable Object to Thread Object for accessing start() method with default thread name
<code>Thread(Runnable r, String name)</code>	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():** Thread 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 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) & Thread control block (TCB).** (deprecated)

**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 Sleeping/Waiting thread

**6. void join()** -Waits for this thread to die.

**7. void join(long mil)** -Waits at most given milliseconds for this **thread to die**

**8. void stop()** -is used to stop the thread(deprecated).

### Static Methods

static void <code>sleep(long ms)</code>	- sleeps/temporary block the thread for specified amount of time
static void <code>yield()</code>	- pause current thread and allow other threads to execute
static <code>Thread currentThread()</code>	- Get currently running thread Object. mainly used in <b>run()</b>
static int <code>activeCount()</code>	- Counts the no.of active threads in current thread group& subgroups.

### Object class methods used in Multithreading

void <code>wait()</code>	- waits the current thread until another thread invokes the notify()
void <code>wait(long ms)</code>	- waits the current thread until another thread invokes the notify()/specified amount of time
void <code>notify()</code>	-Wakes up a single thread that is waiting on this object's monitor.
void <code>notifyAll()</code>	-Wakes up all threads that are waiting on this object's monitor.

### Other useful instance methods of Thread Class

- **void setName(String name)** -set thread's name
- **String getName()** -Returns this thread's name.
- **long getTid()** -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.

1. **public static final int MIN\_PRIORITY (1);**
2. **public static final int NORM\_PRIORITY (5);**
3. **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.**

## java.lang.Runnable Interface

**Runnable** interface has only one abstract method `run()`. Thread class implemented Runnable interface `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
<ol style="list-style-type: none"><li>1.write a class <b>extending Thread class</b></li><li>2.write <b>execution logic in run()</b> method</li><li>3.Create Object of thread <code>ThreadDemo ob = new ThreadDemo();</code></li><li>4.call <b>start()</b> method, it internally calls run() method <code>ob.start();</code></li></ol>	<ol style="list-style-type: none"><li>1.write a class <b>implements Runnable Interface</b></li><li>2.write <b>execution logic in run()</b> method</li><li>3.Create Object of implemented thread class &amp; create Thread Object by passing it <code>RunnableDemo r = new RunnableDemo();</code> <code>Thread ob = new Thread(r);</code></li><li>4.call <b>start()</b> method, it internally calls run() method <code>ob.start();</code></li></ol>

### 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 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

```

#### Example 4: What happens if we start 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

## Interrupting a Thread

An *interrupt* is an indication to a thread that it should stop what it is doing and do something else. It's up to the programmer to decide exactly how a thread responds to an interrupt.

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() - Interrupting a Thread
```

### To Test the Status

- public boolean **isInterrupted()** : It is a **instance method** and tests whether the thread instance on which the method is invoked is interrupted or not.
- public static boolean **interrupted()** : It is a **Static method**, tests whether the CURRENTLY running thread 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); //here Error Re-Thrown
        }
        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, you are re-throwing InterruptedException. So thread is interrupted & also stops its execution.

## Thread is Interrupted & doesn't Stop 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);
        }
        System.out.println("isInterrupted :: "+t1.isInterrupted());
    }
}

```

```

Thread interrupted...java.lang.InterruptedException: sleep interrupted
Thread is Running
isInterrupted :: true

```

In above, Exception is handled. It is only interrupted sleeping thread.remaining 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);
        }
        System.out.println("isInterrupted :: "+t1.isInterrupted());
    }
}

```

```

*** No Sleep is Here ****
Thread is Running ...
isInterrupted :: true

```

The interrupt mechanism is implemented using an internal **flag** known as the `interrupt status`.

Invoking `interrupt()` sets this flag. When a thread checks for an interrupt by invoking the static method `Thread.interrupted`, interrupt status is cleared. The non-static `isInterrupted` method, which is used by one thread to query the interrupt status of another, does not change the interrupt status flag.

By convention, any method that exits by throwing an `InterruptedException` **clears interrupt status** when it does so. However, it's always possible that interrupt status will immediately be set again, by another thread invoking `interrupt`.

## Joining a Thread (join () method)

`java.lang.Thread` class provides the `join()` method which allows one thread to wait until another thread completes its execution.

### 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 next instruction will be execute.

```
public class JoinExample extends Thread {
    public void run() {
        for (int i = 1; i <= 10; i++) {
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                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) {
            e.printStackTrace();
        }
        t2.start();
        t3.start();
    }
}
```

```
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
```

t2, t3 threads waits for t1 thread to die. After completion of t1 thread execution t2, 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.

```
public class JoinExample extends Thread {
    public void run() {
        for (int i = 1; i <= 10; i++) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                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(5000);
        } catch (InterruptedException e) {
            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 5000 milliseconds. But in given time t1 did not die. So, t2, t3 threads start their execution parallel with t1 thread.



## Thread Priority

Each thread has 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 pre-emptive scheduling). **But it is not guaranteed because it depends on JVM specification** that which scheduling it chooses.

- 1) public static final int **MIN\_PRIORITY (1)**;
- 2) public static final int **NORM\_PRIORITY (5)**;
- 3) public static final 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 Priority:"+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    Priority:10
Name :Thread-1    Priority:5
Name :Thread-0    Priority:1
```

Even though t1 starts first, it has **MIN\_PRIORITY** so, it executes last that to depends on JVM Specification

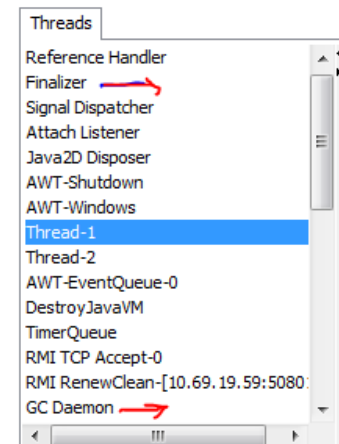
## Daemon Thread

**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 threads automatically.

We can see all daemon threads using **JConsole**

(C:\ProgramFiles\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's current thread as daemon thread.
2. **public boolean isDaemon()** : is used to check that current is daemon.

## Example

```
public class DemonEx extends Thread {
    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)
```

## Thread Group

Thread Group is a process of grouping multiple threads into a single object. We can suspend, interrupt & resume in a single method call.

### Constructors

`ThreadGroup(String name)` : creates a thread group with given name.

`ThreadGroup(ThreadGroup parent, String name)`:creates a thread group with given parent group & name.

### Methods

<code>int activeCount()</code>	returns no. of threads running in current group.
<code>int activeGroupCount()</code>	returns a no. of active group in this thread group.
<code>void destroy()</code>	destroys this thread group and all its subgroups.
<code>String getName()</code>	returns the name of this group.
<code>ThreadGroup getParent()</code>	returns the parent of this group.
<code>void interrupt()</code>	interrupts all threads of this group.
<code>void list()</code>	Prints information of this group to standard console.

```
class ThreadEx extends Thread {
    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("Sm1Codes Group");
        ThreadEx thread = new ThreadEx();
        // adding threads to Thread 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();
    }
}

```

```

java.lang.ThreadGroup[name=Sm1Codes Group,maxpri=10]
Thread Name: Thread-3      Thread Group Name: java.lang.ThreadGroup[name=Sm1Codes Group,maxpri=10]
Thread Name: Thread-1      Thread Group Name: java.lang.ThreadGroup[name=Sm1Codes Group,maxpri=10]
Thread Name: Thread-2      Thread Group Name: java.lang.ThreadGroup[name=Sm1Codes Group,maxpri=10]
Thread[Thread-1,5,Sm1Codes Group]
Thread[Thread-2,5,Sm1Codes Group]
Thread[Thread-3,5,Sm1Codes Group]

```

## 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. If a thread that needs consistent access to an object's fields must 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;

    public int getCount() {
        return this.count;
    }

    public void run() {
        for (int i = 1; i <= 5; i++) {
            waitCounter(i);
            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 in **3 ways**

1. **Synchronized Instance Methods**
2. **Synchronized Static Methods.**
3. **Synchronized Blocks**

**Note:** All the Synchronized **methods, blocks** in the part of class which extends **Thread/Runnable**

### 1. Synchronized Instance methods:

If an ordinary instance method is made 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**

## Inter Thread Communication

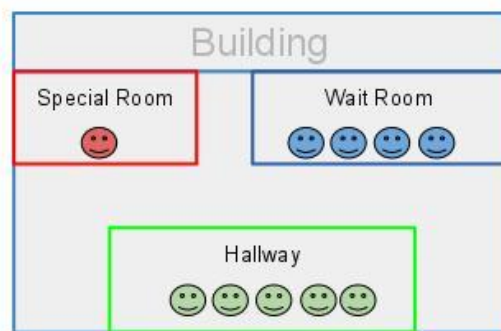
If a Thread is synchronized only one thread should be access at a time. To access multiple threads on synchronized resource there should be some communication between them. Inter-thread communication or Co-operation is all about allowing synchronized threads to communicate with each other.

**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.

## 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.

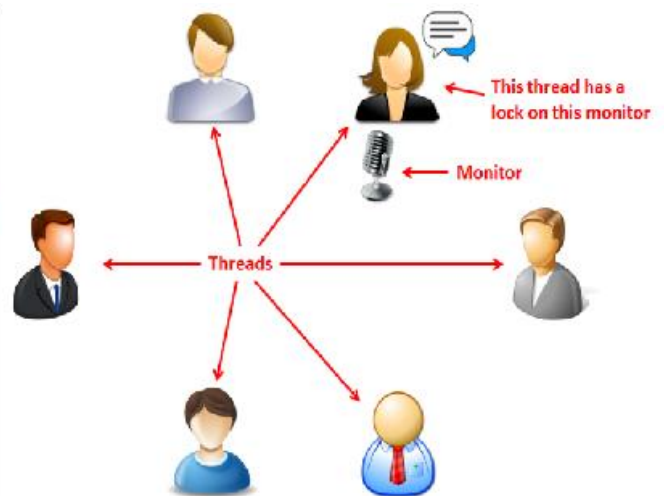
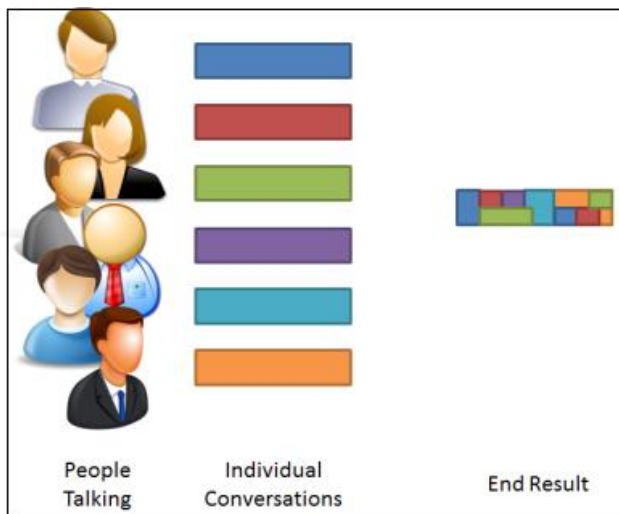
### Thread 1:

```
public void a()
{
    synchronized(someObject) {
        // do something (1)
    }
}
```

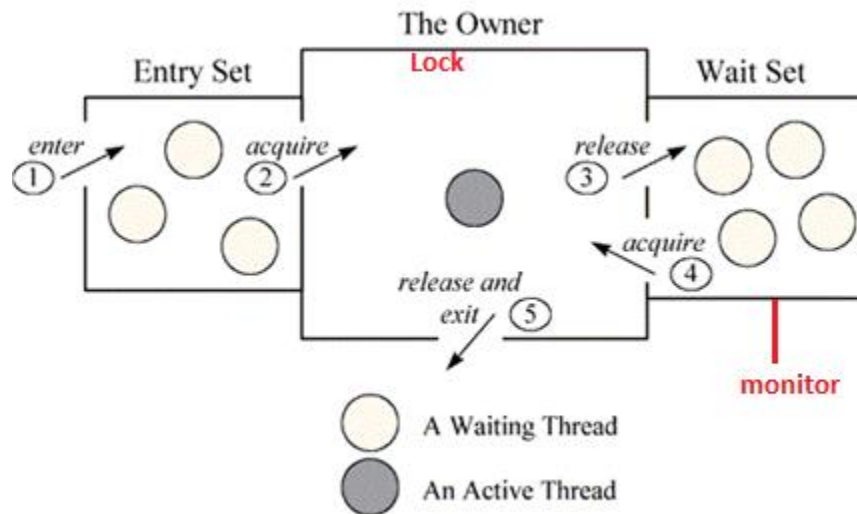
### Thread 2:

```
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.



## 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.

Lock's 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 until deposit() done & 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 notify 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 until deposit() done & notify

DEPOSITING
*****
DEPOSIT COMPLETED
calling notify on withdraw()
***** WITHDRAW COMPLETED *****

```

### 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
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

## Callable Interface

The Callable interface is similar to Runnable, contains `call()` method which returns a **Value** & throws **CheckedException**.

```
public interface Callable<V> {  
    V call() throws Exception;  
}
```

```
public class Demo implements Callable<String>{  
  
    @Override  
    public String call() throws Exception {  
        return "Hello";  
    }  
  
    public static void main(String[] args) throws Exception {  
        String msg = new Demo().call();  
        System.out.println(msg);  
    }  
}
```

Hello



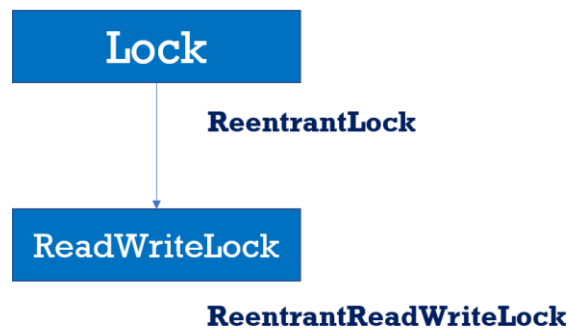
## 9. java.util.Concurrency

**java.util.concurrent** package introduced in version 5.0 with following features

1. **Lock objects** support locking idioms that simplify many concurrent applications.
2. **Executors** define a high-level API for launching and managing threads. Executor implementations provided by **java.util.concurrent** package provides **thread pool management** suitable for large-scale applications.
3. **Atomic variables** have features that minimize synchronization and help avoid memory consistency errors.
4. **Concurrent collections** make it easier to manage large collections of data, and can greatly reduce the need for synchronization.
5. **ThreadLocalRandom** (in JDK 7) provides efficient generation of pseudorandom numbers from multiple threads.

### Lock Interface

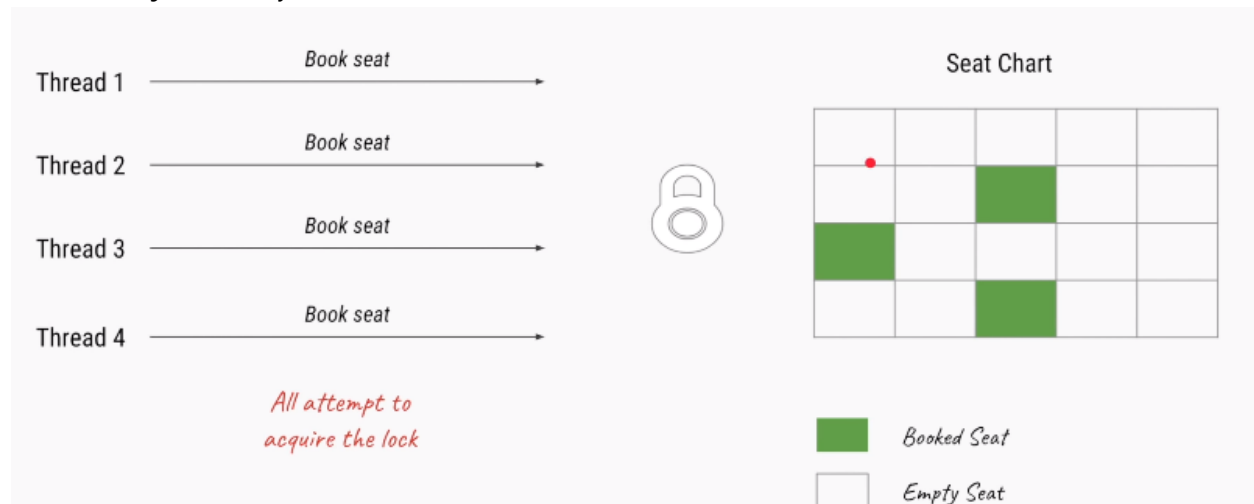
**java.util.concurrent.locks.Lock** interface is used to as a **thread synchronization mechanism similar to synchronized blocks**.



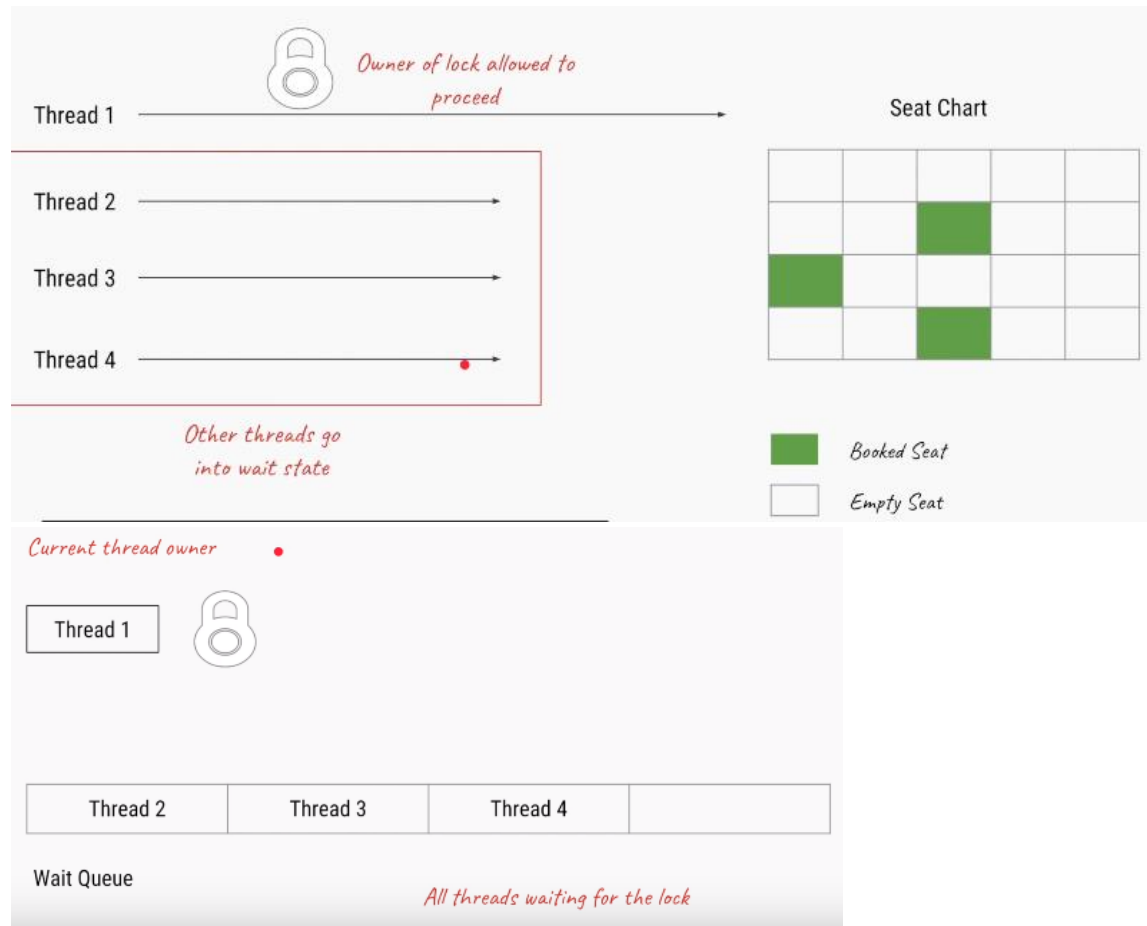
[https://www.youtube.com/watch?v=ahBC69\\_ijk4&index=13&t=1s&list=PLhfHPmPYPPRk6yMrcbfafFGSbE2EPK\\_A6](https://www.youtube.com/watch?v=ahBC69_ijk4&index=13&t=1s&list=PLhfHPmPYPPRk6yMrcbfafFGSbE2EPK_A6)

### Scenario

1. In **BookMyShow** only one thread is allowed to book **seat:A10**



2. Here **Thread-1** get the lock for **seat:A10**, other threads will wait in waiting queue, until t1 releases the Lock



3. Once Lock got released, remaining threads will try to get the Lock. Process will continue.



	Advantage	Disadvantage
Fair lock	Equal chance for all threads	Slower
Unfair lock	Faster (more throughput)	Possible thread starvation

## Methods

**1.void lock()** – To acquire Lock.if lock is already available current thread will get that lock.if lock is not available it waits until get the lock.it is similar to synchronized keyword.

**2.void unlock()** – Releases the lock.**if we call on thread which is not having lock it will throw runtime exception `IllegalMonitorStateException`**

**3.boolean tryLock()** – To acquire lock without waiting.if it acquires lock returns true, if not false & continues its execution without waiting.In this case thread never goes into waiting state

```
if(l.tryLock())
{
    //perform safe operations
}else{
    //perform alternative operations
}
```

**4.boolean tryLock(long time, TimeUnit unit)** – Same as above, but specifying time.TimeUnit is Enum having values as `NANOSECONDS`, `SECONDS`, `MINUTES`, `HOURS`, `DAYS`

```
if(l.tryLock(1000,TimeUnit.MINUTES)) //waiting for 1000 minutes
{
    //perform safe operations
}else{
    //perform alternative operations
}
```

**5. void lockInterruptibly()** – Acquires lock if available & returns immediately. Not available it will wait.while waiting if thread is interrupted then thread won't get the lock.

## ReentrantLock Class

It is the **implementation** class of **Lock interface** & direct child class of Object.Reentrant means **A thread can acquire same lock multiple times without any issue.**

In ReentrantLock maintains **holdcount** variable. when ever we call **lock()** it increments threads **holdcount++** & when ever thread calls **unlock()** it decrements **holdcount--** value. Lock will be released when ever count reaches 0. Below are the benefits by using ReentrantLock...

### 1.lock on a resource more than once

ReentrantLock allow threads to enter into **lock on a resource more than once**. When the thread first enters into lock, a **hold count** is set to one. Before unlocking the thread can re-enter into lock again and every time hold count is incremented by one. For every unlock request, hold count is decremented by one and when hold count is 0, the resource is unlocked

### 2.offer a fairness parameter

after a thread unlocks the resource, the lock would go to the thread which has been waiting for the longest time. This fairness mode is set up by passing true to the constructor of the lock.

```
Lock lock = new ReentrantLock(true); //Setting Fairness Policy
```

## Methods

It has all the methods which are there in Lock interface, additionally it has following methods.

- `getHoldCount()`: This method returns the count of the number of locks held on the resource
- `isHeldByCurrentThread()`: This method returns true if the lock on the resource is held by the current thread.

## NormalLock Example

```
public class NormalLock extends Thread {
    static int i = 0;
    Lock lock = new ReentrantLock();

    @Override
    public void run() {
        increment();
    }

    public void increment() {
        try {
            lock.lock();

            i++;
            S.O.P(Thread.currentThread().getName() + " Got Lock: incremented, i=" + i);
            Thread.sleep(3000);

            lock.unlock();

        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
        }
    }

    public static void main(String[] args) {

        NormalLock ob = new NormalLock();

        Thread t1 = new Thread(ob, "One");
        Thread t2 = new Thread(ob, "Two");
        Thread t3 = new Thread(ob, "Three");

        t1.start();
        t2.start();
        t3.start();
    }
}
```

```
One Got Lock: incremented, i=1
Three Got Lock: incremented, i=2
Two Got Lock: incremented, i=3
```

Here all the Threads are executed, because each thread will wait until they get the lock & in above we haven't set fairness true, so order of thread executed randomly(1, 3, 2).

If we set fairness is true output is like

```
Lock lock = new ReentrantLock(true);
```

```
One Got Lock: incremented, i=1
Two Got Lock: incremented, i=2
Three Got Lock: incremented, i=3
```

**Note:** All the Synchronized **methods, blocks, Locks** in the part of class which extends **Thread/Runnable** Remember, **Lock** is same as Synchronized block. Opening brace is – **lock**, closing brace is – **unlock**.

## Example : tryLock()

```
public class TryLockDemo extends Thread {
    static int i = 0;
    Lock lock = new ReentrantLock();

    public void run() {
        increment();
    }
    public void increment() {
        try {
            // WAIT FOR 2 Seconds to get the Lock
            if (lock.tryLock(2, TimeUnit.SECONDS)) {
                i++;
                SOP(Thread.currentThread().getName() + " Got Lock: incremented, i=" + i);
                Thread.sleep(3000);
                lock.unlock(); // Unlocks here
            } else {
                SOP(Thread.currentThread().getName() + " Iam doing Something else");
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            System.out.println("Final i : "+i);
        }
    }
    public static void main(String[] args) {
        TryLockDemo ob = new TryLockDemo();

        Thread t1 = new Thread(ob, "One");
        Thread t2 = new Thread(ob, "Two");
        Thread t3 = new Thread(ob, "Three");

        t1.start();
        t2.start();
        t3.start();
    }
}
```

```
Three Got Lock: incremented, i=1
Two Iam doing Something else
One Iam doing Something else
Final i : 1
```

Here all threads trying to get lock, but **Three** Thread got lock, remaining threads won't wait for lock, they are executing else block some kind of alternative job without waiting. So final count is 1.

## HoldCount Example

```
public class HoldCount extends Thread {
    static int i = 1;

    ReentrantLock lock = new ReentrantLock();

    public void run() {
        lock.lock();
        System.out.println("one");

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

        System.out.println("HOLD Count : " + lock.getHoldCount());
        lock.unlock();

        System.out.println("isHeldByCurrentThread : " + lock.isHeldByCurrentThread());

        System.out.println("HOLD Count : " + lock.getHoldCount());
        lock.unlock();

        System.out.println("HOLD Count : " + lock.getHoldCount());
    }
}
```

```

public static void main(String[] args) {
    HoldCount t = new HoldCount();
    t.setName("BIG-THREAD");
    t.start();
}

```

```

one
Two
HOLD Count : 2
isHeldByCurrentThread : true
HOLD Count : 1
HOLD Count : 0

```

## Synchronization VS Locks

There are few differences between the use of **Synchronized block** and using **Lock API's**

- **A synchronized block is fully contained within a method** – we can have *Lock API's* **lock()** and **unlock()** operation in separate methods.

```

public class ThreadDemo extends Thread {

    Lock lock = new ReentrantLock();
    public int sum(int a, int b) {
        //Locking in one method
        lock.lock();
        a = 10;
        b = 20;
        return a + b;
    }
    public void show() {
        System.out.println(sum(10, 20));
        //UnLocking in another method
        lock.unlock();
    }
}

```

- A *synchronized block* does not support **the fairness**, any thread can acquire the lock – there is no preference specified for getting the lock. We can achieve fairness within the Lock APIs by specifying the **fairness property**. It makes sure that longest waiting thread is given access to lock.
- A thread gets blocked if it can't get an access to the *synchronized block*. **The Lock API provides *tryLock()* method. The thread acquires lock only if it is available and not held by any other thread.** This reduces blocking time of thread waiting for the lock.

```

if(l.tryLock())
{ //perform safe operations
}else{
    //perform alternative operations
}

```

- A thread which is in "waiting" for a long time (say hours) to acquire the access to **synchronized block**, can't be interrupted. **The Lock API provides a method *lockInterruptibly()* which can be used to interrupt the thread when it is waiting for the lock**

## ReadWriteLock interface

In addition to *Lock* interface, we have a **ReadWriteLock** interface which maintains a pair of locks, one for **read-only operations**, and one for the **write operation**.

### ReentrantReadWriteLock class

**ReentrantReadWriteLock** class is implementation of it

- **Lock readLock()** – returns the lock that's used for reading
- **Lock writeLock()** – returns the lock that's used for writing. Once lock gain no write & read allowed.



```
public class ReadWriteLockDemo implements Runnable {
    ArrayList list = new ArrayList<>();
    ReadWriteLock rwlock = new ReentrantReadWriteLock(true);

    Lock readLock = rwlock.readLock();
    Lock writeLock = rwlock.writeLock();

    public void write() {
        writeLock.lock();
        int ele = 100;
        list.add(ele);
        System.out.println(Thread.currentThread().getName() + " : Write : " + ele);
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        writeLock.unlock();
    }

    public void read() {
        if (readLock.tryLock()) {
            int index = list.size();
            int ele = (int) list.get(index - 1);
            System.out.println(Thread.currentThread().getName() + " : Read : " + ele);
            readLock.lock();
        } else {
            S.O.println(Thread.currentThread().getName() + ": Read Lock Not available");
        }
    }

    @Override
    public void run() {
        String thname = Thread.currentThread().getName();
        if (thname.contains("write")) {
            write();
        } else {
            read();
        }
    }
}
```

```

public static void main(String[] args) throws InterruptedException {
    ReadWriteLockDemo ob = new ReadWriteLockDemo();

    new Thread(ob, "write1").start();
    Thread.sleep(2000);

    new Thread(ob, "read1").start();
    Thread.sleep(5000);

    new Thread(ob, "read2").start();
}
}
write1 : Write : 100
read1 : Read Lock Not availble
read2 : Read : 100

```

## ReentrantLock

One thread at a time

## ReadWriteLock

One writer thread at a time  
OR  
Multiple reader threads at a time

## Conditions

A `java.util.concurrent.locks.Condition` interface provides a thread ability to suspend its execution, until the given condition is **true**.

Condition variables are instance of `java.util.concurrent.locks.Condition` class, which provides inter thread communication methods similar to wait, notify and notifyAll e.g. **await()**, **signal()** and **signalAll()**.

### Alternative for `wait(await)` & `notify(signal)`

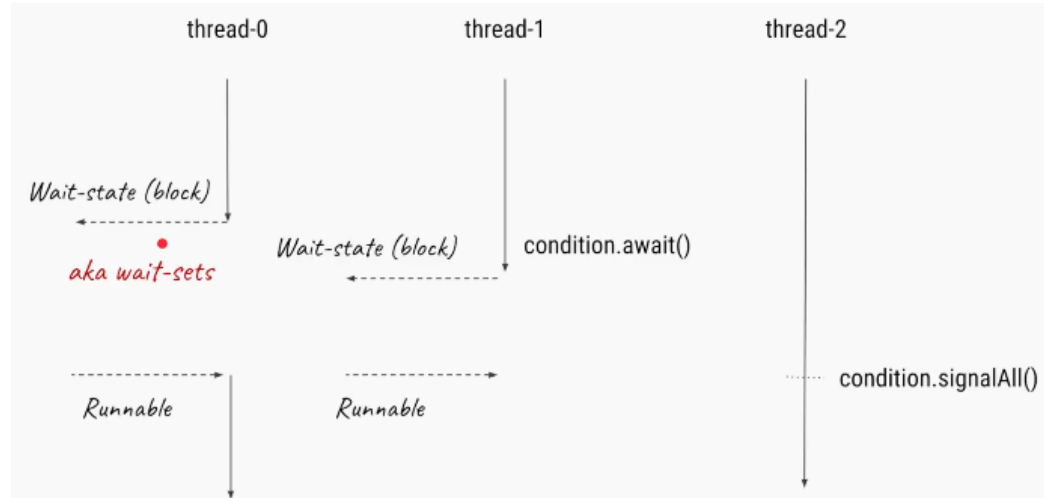
We can Create Condition object by using **ReentrantLock** and **ReentrantReadWriteLock** Which are implementation classes of Lock interface. You can create condition variable by calling **lock.newCondition()** method

### Methods

- **await()**:The current thread suspends its execution until it is signalled or interrupted.
- **await(long time, TimeUnit unit)** :The current thread suspends its execution until it is signalled, interrupted, or the specified amount of time elapses.
- **awaitNanos(long nanosTimeout)** :The current thread suspends its execution until it is signalled, interrupted, or the specified amount of time elapses.
- **awaitUninterruptibly()** :The current thread suspends its execution until it is signalled (cannot be interrupted).
- **signal()**:This method wakes a single thread which is waiting for a longtime on this condition.
- **signalAll()**:This method wakes all threads waiting on this condition.



if one thread is waiting on a condition by calling **condition.await()** then once that condition changes, second thread can call **condition.signal()** or **condition.signalAll()** method to notify that its time to wake-up.



```
private Lock lock = new ReentrantLock();
private Condition conditionMet = lock.newCondition();

public void method1() throws InterruptedException {
    lock.lock();
    try {
        conditionMet.await(); <-Suspend here
        // can now do dependant operations <- Resume here
    } finally {
        lock.unlock();
    }
}

public void method2() {
    lock.lock();
    try {
        // do some operations
        conditionMet.signal();
    } finally {
        lock.unlock();
    }
}
```

thread-1

thread-2

Locks are used for Synchronization. We will use Lock and Condition variables for solving classic Producer Consumer problem.

In the producer-consumer problem, there is one Producer that is producing something and there is one Consumer that is consuming the products produced by the Producer. The producers and consumers share the **same memory buffer that is of fixed-size**. The following are the problems that might occur in the Producer-Consumer:

- The producer should produce data only when the buffer is not full. If the buffer is full, then the producer shouldn't be allowed to put any data into the buffer.
- The consumer should consume data – only when the buffer is full. If the buffer is empty, then the consumer shouldn't be allowed to take any data from the buffer.
- The producer and consumer should not access the buffer at the same time.

```

private Lock lock = new ReentrantLock();
private Condition added = lock.newCondition();
private Condition removed = lock.newCondition();

```

```

public void produce() throws Interrupt
lock.lock();
try {
    while (count == MAX_COUNT)
        removed.await();

    addData();
    added.signal();
} finally {
    lock.unlock();
}
}

```

```

public String consume() throws Inte
lock.lock();
try {
    while (count == 0)
        added.await();

    String data = getData();
    removed.signal();

    return data;
} finally {
    lock.unlock();
}
}

```

### Example : solving classic Producer Consumer problem. //Not Imp

```

public class ProducerConsumerSolutionUsingLock {

    public static void main(String[] arg) {

        // Object on which producer and consumer thread will operate.
        ProducerConsumerImpl sharedObject = new ProducerConsumerImpl();

        // creating producer and consumer threads.
        Producer p = new Producer(sharedObject);
        Consumer c = new Consumer(sharedObject);

        // starting producer and consumer threads.
        p.start();
        c.start();
    }
}

class ProducerConsumerImpl {
    // Producer consumer problem data.
    private static final int CAPACITY = 10;
    private final Queue queue = new LinkedList();
    private final Random theRandom = new Random();

    // Lock and condition variables.
    private final Lock aLock = new ReentrantLock();
    private final Condition bufferNotFull = aLock.newCondition();
    private final Condition bufferNotEmpty = aLock.newCondition();

    public void put() throws InterruptedException {
        aLock.lock();
        try {
            while (queue.size() == CAPACITY) {
                s.o.p(Thread.currentThread().getName() + ": Buffer is full, waiting.");
                bufferNotEmpty.await();
            }

            int number = theRandom.nextInt();
            boolean isAdded = queue.offer(number);
            if (isAdded) {
                s.o.p("%s added %d into queue %n", Thread.currentThread().getName(), number);

                // signal consumer thread that, buffer has element now
                s.o.p(Thread.currentThread().getName() + ": Signaling that buffer has data");
                bufferNotFull.signalAll();
            }
        } finally {
            aLock.unlock();
        }
    }
}

```

```

    public void get() throws InterruptedException {
        aLock.lock();
        try {
            while (queue.size() == 0) {
                s.o.p(Thread.currentThread().getName() + ": Buffer is empty, waiting.");
                bufferNotFull.await();
            }

            Integer value = (Integer) queue.poll();
            if (value != null) {
                s.o.p("%s consumed %d from queue %n ", Thread.currentThread().getName(), value);

                // Signal producer thread that, buffer me be empty now
                s.o.p(Thread.currentThread().getName() + ": Signaling buffer is empty ");
                bufferNotEmpty.signalAll();
            }
        } finally {
            aLock.unlock();
        }
    }
}

class Producer extends Thread {
    ProducerConsumerImpl producerConsumer;

    public Producer(ProducerConsumerImpl sharedObject) {
        super("PRODUCER");
        this.producerConsumer = sharedObject;
    }
    @Override
    public void run() {
        try {
            producerConsumer.put();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

class Consumer extends Thread {
    ProducerConsumerImpl producerConsumer;

    public Consumer(ProducerConsumerImpl sharedObject) {
        super("CONSUMER");
        this.producerConsumer = sharedObject;
    }
    @Override
    public void run() {
        try {
            producerConsumer.get();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
}

```

[https://www.youtube.com/watch?v=N0mMm5PF5Ow&index=11&t=1s&list=PLhfHPmPYPPRk6yMrcbfafFGSbE2EPK\\_A6](https://www.youtube.com/watch?v=N0mMm5PF5Ow&index=11&t=1s&list=PLhfHPmPYPPRk6yMrcbfafFGSbE2EPK_A6)

## 10. Executor Framework – ThreadPools

Normally we will create & execute Threads in following way.

```
static class Task implements Runnable {
    public void run() {
        System.out.println("Thread Name: " + Thread.currentThread().getName());
    }
}

public static void main(String[] args) {

    Thread thread1 = new Thread(new Task());
    thread1.start();
    System.out.println("Thread Name: " + Thread.currentThread().getName());
}
```

If we want to run 10 threads, we will do in following way

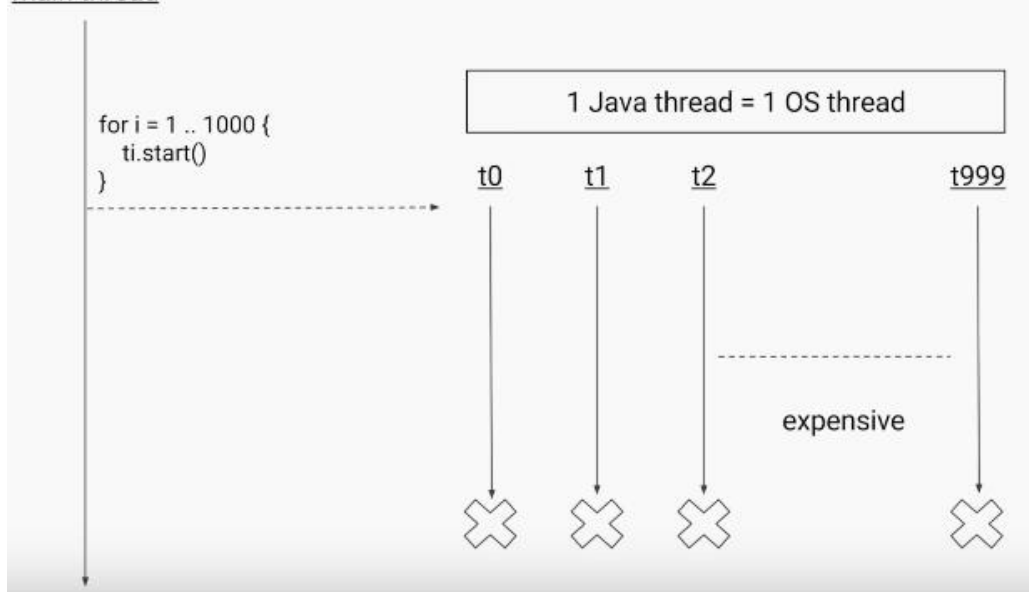
```
static class Task implements Runnable {
    public void run() {
        System.out.println("Thread Name: " + Thread.currentThread().getName());
    }
}

public static void main(String[] args) {

    for (int i = 0; i < 10; i++) {
        Thread thread = new Thread(new Task());
        thread.start();
    }
    System.out.println("Thread Name: " + Thread.currentThread().getName());
}
```

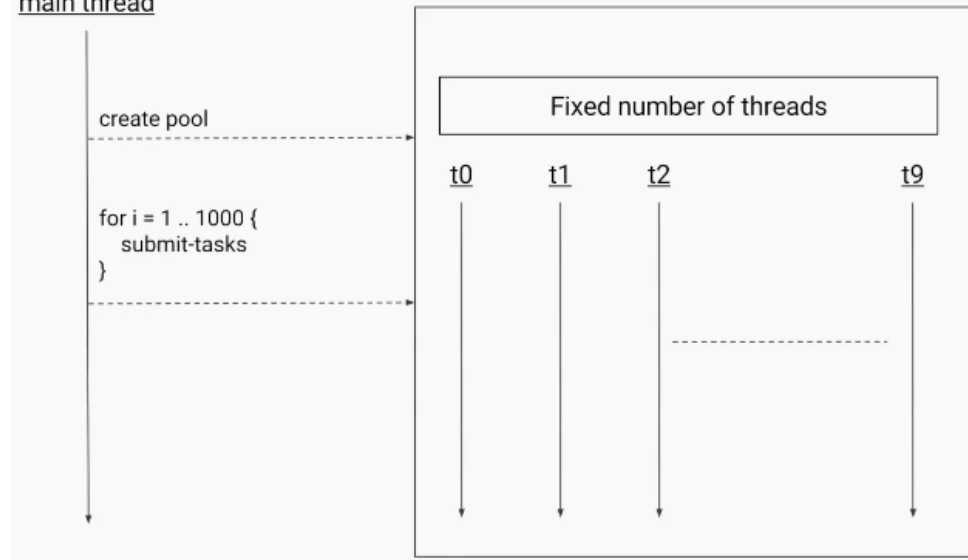
If we want to run 1000 Threads, then it will become more expensive (more OS Threads, more Heap). Java will create 1000 OS level Threads to process this.

main thread



We can avoid above situation by create only 10 OS level Threads & submit 1000 jobs to them.

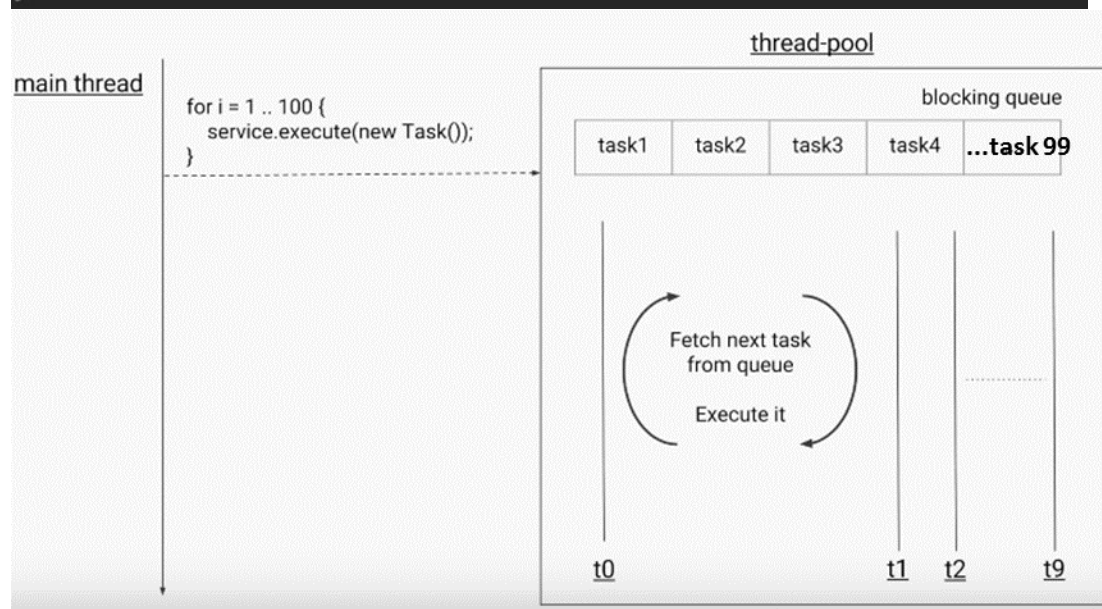
main thread



The code for above example will become

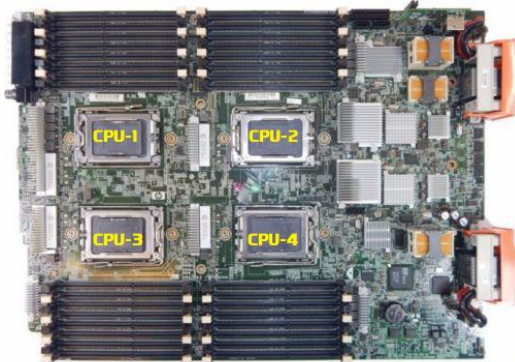
```
static class Task implements Runnable {
    public void run() {
        System.out.println("Thread Name: " + Thread.currentThread().getName());
    }
}
public static void main(String[] args) {
    // create the pool
    ExecutorService service = Executors.newFixedThreadPool( nThreads: 10);

    // submit the tasks for execution
    for (int i = 0; i < 100; i++) {
        service.execute(new Task());
    }
    System.out.println("Thread Name: " + Thread.currentThread().getName());
}
```





We may heard **Dual Core , QuadCore processors**. Core is nothing but **CPU**. Each "core" is the part of the chip that does the processing work. Essentially, each core is **a central processing unit (CPU)**.



**Quad Core Motherboard**



**Dual Core Motherboard**

If we have QuadCore system, we can able to execute 4 Threads at a time, each Thread will execute by one CPU unit of QuadCore. In above examples, we created threadPool of 10, but the best way to provide no. of threads is by getting CPU count(Processor Count)

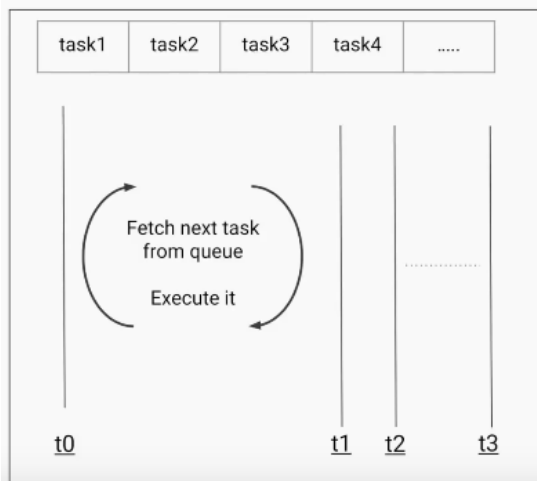
```
public static void main(String[] args) {
    // get count of available cores
    int coreCount = Runtime.getRuntime().availableProcessors();
    ExecutorService service = Executors.newFixedThreadPool(coreCount);

    // submit the tasks for execution
    for (int i = 0; i < 100; i++) {
        service.execute(new CpuIntensiveTask());
    }
}

static class CpuIntensiveTask implements Runnable {
    public void run() {
        // some CPU intensive operations
    }
}
```

The IdealPool size will be no. of CPU cores your system have.

thread-pool



CPU

Core 1	Core 2
Core 3	Core 4

Max 4 threads can run at a time

# Types of ThreadPools

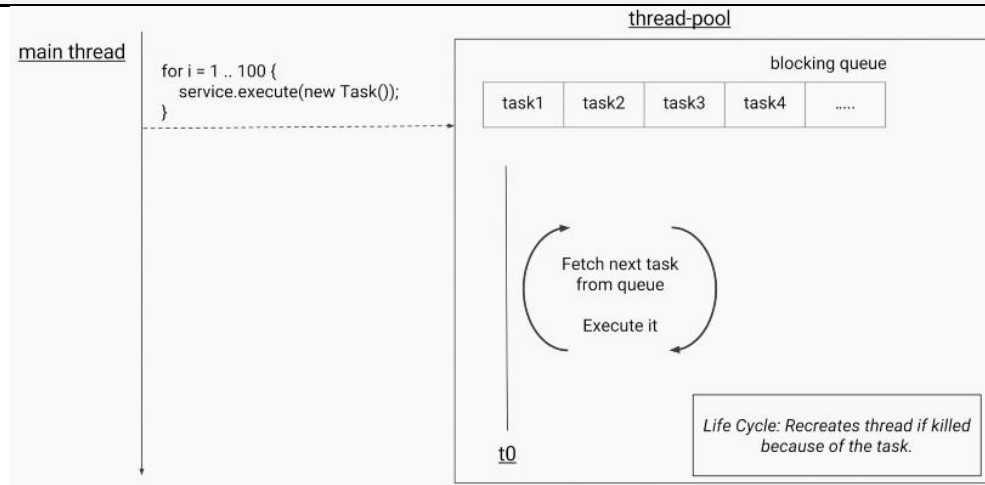
We have following types of Thread Pools

1. **SingleThread Pool**
2. **FixedThreadPool**
3. **CachedThreadPool**
4. **ScheduledThreadPool**
5. **Fork/Join pool**

## 1. SingleThread Pool

A thread **pool with only one thread** with an unbounded queue, which only executes one task at a time.

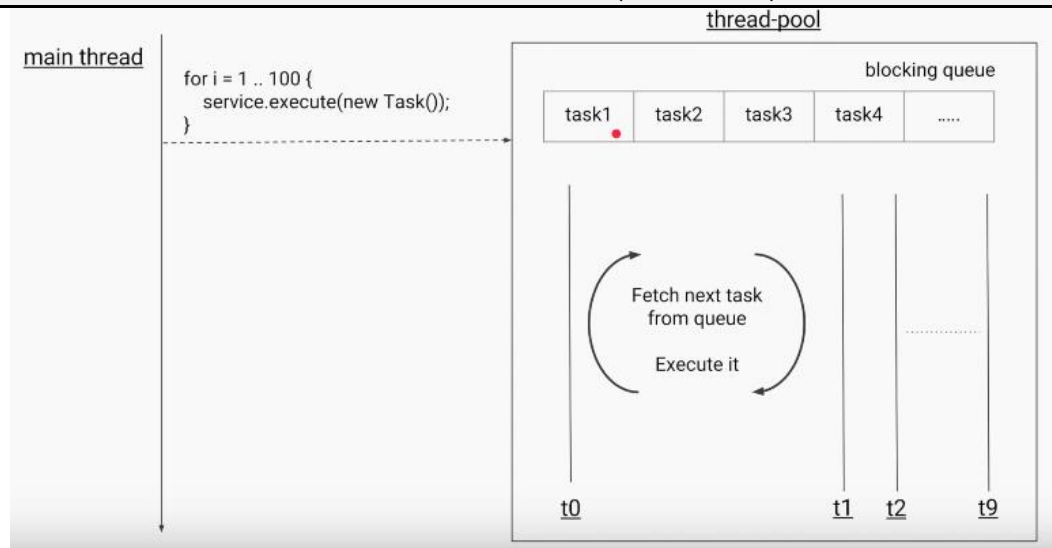
```
static ExecutorService newSingleThreadExecutor()
```



## 2. FixedThreadPool

A thread **pool with a fixed number of threads** which share an unbounded queue; if all threads are active when a new task is submitted, they will wait in queue until a thread becomes available

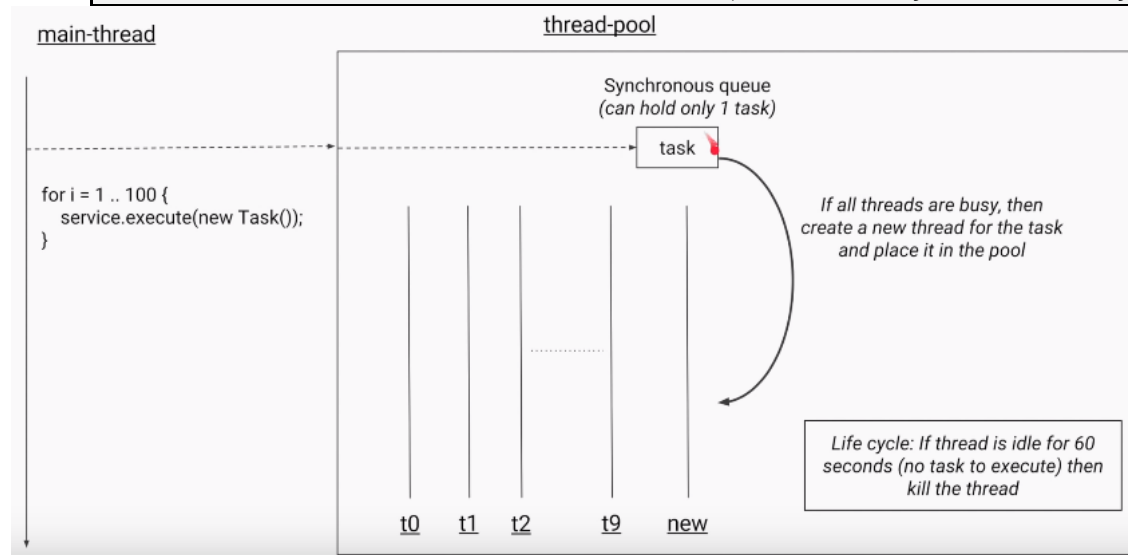
```
static ExecutorService newFixedThreadPool(int nThreads)
```



### 3.CachedThreadPool

Creates a thread pool that **creates new threads as needed**, but will reuse previously constructed threads when they are available

```
static ExecutorService newCachedThreadPool()  
static ExecutorService newCachedThreadPool(ThreadFactory threadFactory)
```



- It doesn't have any Queue like FixedThreadPool, instead it has **synchronous Queue which holds one task at a time.**
- On Submitting Task, it will search for any Thread is free in current Thread Pool. if not, It will create another thread to do the job.
- It will kill the useless threads. if Threads idle for more than 60 sec.,

```
public static void main(String[] args) {  
    // for lot of short lived tasks  
    ExecutorService service = Executors.newCachedThreadPool();  
  
    // submit the tasks for execution  
    for (int i = 0; i < 100; i++) {  
        service.execute(new Task());  
    }  
}  
  
static class Task implements Runnable {  
    public void run() {  
        // short lived task  
    }  
}
```

Above 3 are Part of **ExecutorService**

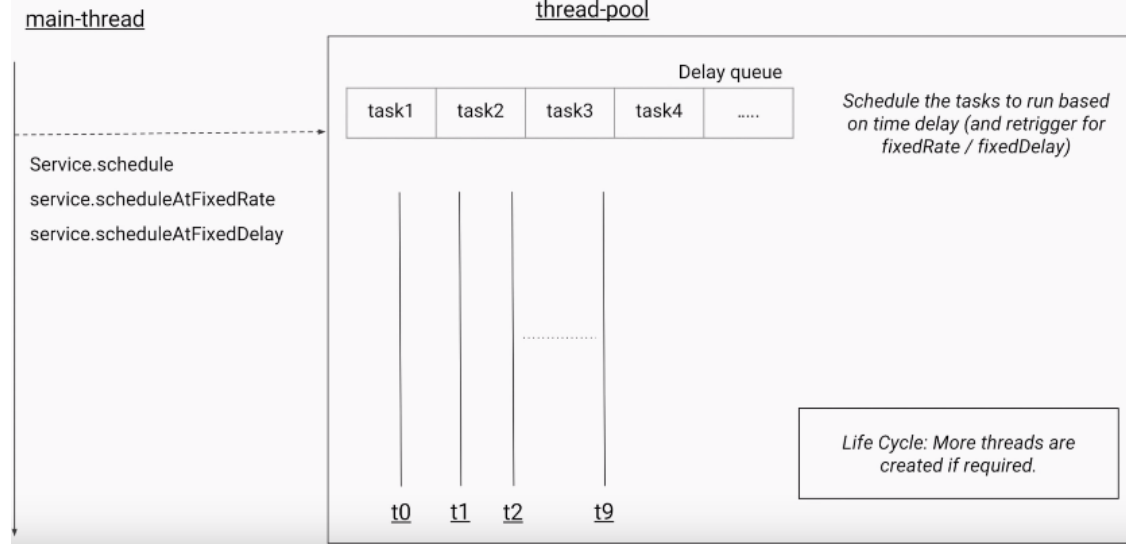
### 4.ScheduledThreadPool

creates an executor that can schedule tasks to execute after a given delay, or to execute periodically.

```
static ScheduledExecutorService newSingleThreadScheduledExecutor()  
static ScheduledExecutorService newScheduledThreadPool(int poolSize)
```



It will store the all the tasks which are submitted in Delay Queue



```

public static void main(String[] args) {
    // for scheduling of tasks
    ScheduledExecutorService service = Executors.newScheduledThreadPool( corePoolSize: 10 );

    // task to run after 10 second delay
    service.schedule(new Task(), delay: 10, SECONDS);

    // task to run repeatedly every 10 seconds
    service.scheduleAtFixedRate(new Task(), initialDelay: 15, period: 10, SECONDS);

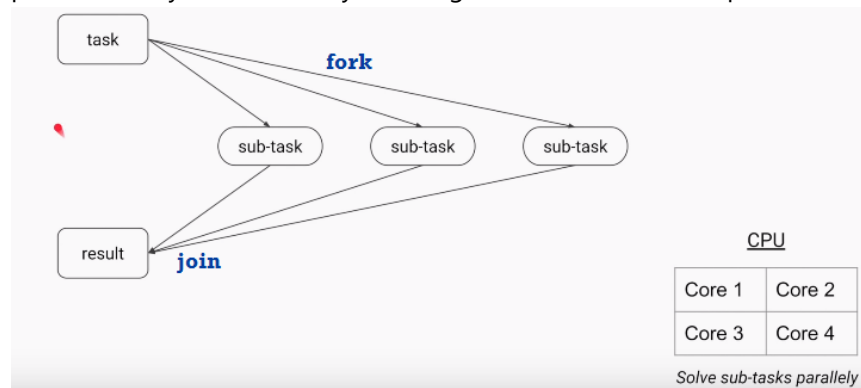
    // task to run repeatedly 10 seconds after previous task completes
    service.scheduleWithFixedDelay(new Task(), initialDelay: 15, delay: 10, TimeUnit.SECONDS);
}

public class Task implements Runnable {
    public void run() {
        // task that needs to run
        // based on schedule
    }
}

```

### 5.Fork/Join pool:

It is a special thread pool that uses the Fork/Join framework to take advantages of multiple processors to perform heavy work faster by breaking the work into smaller pieces recursively.

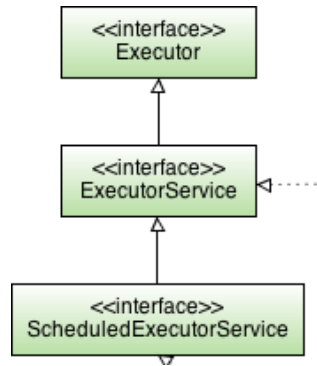


**Remember method names,**

- For Single Thread ends with - **\*\*Executor()**
- For Multiple Threads end with - **\*\*ThreadPool()**

# ExecutorService API

In the `java.util.concurrent` package we have following interfaces to work with Thread pools



**Executor** — Parent class for all Executor services

**ExecutorService** — A subinterface of Executor that adds methods to manage lifecycle of threads used to run the submitted tasks and methods to produce a Future to get a result from an asynchronous computation.

**ScheduledExecutorService** — A subinterface of ExecutorService, to execute commands periodically or after a given delay

**Executors** — Utility class for getting ExecutorService Objects.

## 1. Executors Utility class

Factory and utility methods for `Executor`, `ExecutorService`, `ScheduledExecutorService`, `ThreadFactory`, and `Callable` classes defined in this package. This class supports the following kinds of methods:

- Methods that create and return an `ExecutorService` set up with commonly useful configuration settings.
- Methods that create and return a `ScheduledExecutorService` set up with commonly useful configuration settings.
- Methods that create and return a "wrapped" `ExecutorService`, that disables reconfiguration by making implementation-specific methods inaccessible.
- Methods that create and return a `ThreadFactory` that sets newly created threads to a known state.
- Methods that create and return a `Callable` out of other closure-like forms, so they can be used in execution methods requiring `Callable`.

## 2. Executor Interface

The `Executor` interface provides a single method, `execute`

```
void execute(Runnable command)
```

It designed to be a drop-in replacement for older `start()` & `run()`, it a combination of both of them.

Pool	Queue Type	Why?
FixedThreadPool	LinkedBlockingQueue	Threads are limited, thus unbounded queue to store all tasks.
SingleThreadExecutor	LinkedBlockingQueue	<i>Note: Since queue can never become full, new threads are never created.</i>
CachedThreadPool	SynchronousQueue	Threads are unbounded, thus no need to store the tasks. Synchronous queue is a queue with single slot
ScheduledThreadPool	DelayedWorkQueue	Special queue that deals with schedules/time-delays
Custom	ArrayBlockingQueue	Bounded queue to store the tasks. If queue gets full, new thread is created (as long as count is less than <code>maxPoolSize</code> ).

### 3.ExecutorService Interface

```
ExecutorService executorService = Executors.newSingleThreadExecutor();
ExecutorService executorService = Executors.newFixedThreadPool(10);
ExecutorService executorService = Executors.newCachedThreadPool();
```

```
void execute(Runnable command) //inherited from Executor
```

```
Future submit(Callable task)
```

```
Future submit(Runnable task)
```

- The `ExecutorService` interface implements `Executor` interface with additional `submit()` method.
- Like `execute()` - `submit()` accepts `Runnable` objects, but also accepts `Callable` objects, which allow the task to return a value.
- The `submit()` method returns a `Future` object, which is used to retrieve the `Callable` return value and to manage the status of both `Callable` and `Runnable` tasks.
- `ExecutorService` also provides methods for submitting large **collections of Callable objects**.
- Finally, `ExecutorService` provides a number of methods for managing the **shutdown** of the executor. To support immediate shutdown, tasks should handle `interrupts` correctly.

#### Methods

```
Future submit(Callable<T> task);
```

```
// Executes given tasks, returns list of Future results when all complete.
```

```
List<Future<T>> invokeAll(Collection<Callable<T>> tasks)
```

```
List<Future<T>> invokeAll(Collection<Callable<T>> tasks, long timeout, TimeUnit unit)
```

```
// Executes the given tasks, returns the result of one of the task completed successfully
```

```
T invokeAny(Collection<Callable<T>> tasks)
```

```
T invokeAny(Collection<Callable<T>> tasks, long timeout, TimeUnit unit)
```

```
// Initiates shutdown signal wait for Tasks to complete, but no new tasks will be accepted.
```

```
void shutdown();
```

```
// Attempts to stop all tasks & returns a list of the tasks that were awaiting execution.
```

```
List shutdownNow();
```

```
boolean isShutdown();
```

```
boolean isTerminated();
```

```
boolean awaitTermination(long timeout, TimeUnit unit)
```

### Callable Interface

The `Callable` interface is similar to `Runnable`, contains `call()` method which **returns a Value** & throws **CheckedException**.

```
public interface Callable<V> {
    V call() throws Exception;
}
```

### Future Interface

When ever we use `submit()` method, the result will stored in `Future` Object. We have following methods to process the results from **Future** Object.

- **V get():**get() returns an actual result of the Callable task's execution or null in the case of Runnable task. Calling the get() method while the task is still running will cause execution to block until the task is properly executed and the result is available.

```
Future<String> future = executorService.submit(callableTask);
String result = null;
try {
    result = future.get();
} catch (InterruptedException | ExecutionException e) {
    e.printStackTrace();
}
```

- **V get(long timeout, TimeUnit t):**We can avoid blocking, by specifying time limit to get the result.

```
String result = future.get(200, TimeUnit.MILLISECONDS);
```

If the execution period is longer than specified (in this case 200 milliseconds), a TimeoutException will be thrown.

- And also we can use cancel() methods if get() taking more time.

```
boolean canceled = future.cancel(true);
boolean isCancelled = future.isCancelled();
```

- **boolean isDone():**used to check if the assigned task is already processed or not.

## 1. newSingleThreadExecutor

```
class RunnableJob implements Runnable {
    public static int count = 1;

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName()+" : Runnable Job :"+ count);
        count++;
    }
}

public class ThreadDemo {
    public static void main(String[] args) throws InterruptedException {

        ExecutorService executorService = Executors.newSingleThreadExecutor();
        //ExecutorService executorService = Executors.newFixedThreadPool(10);
        //ExecutorService executorService = Executors.newCachedThreadPool();
        for (int i = 0; i < 20; i++) {
            executorService.submit(new RunnableJob());
        }
    }
}
```

```
pool-1-thread-1 : Runnable Job :1
pool-1-thread-1 : Runnable Job :2
pool-1-thread-1 : Runnable Job :3
pool-1-thread-1 : Runnable Job :4
pool-1-thread-1 : Runnable Job :5
pool-1-thread-1 : Runnable Job :6
pool-1-thread-1 : Runnable Job :7
pool-1-thread-1 : Runnable Job :8
pool-1-thread-1 : Runnable Job :9
pool-1-thread-1 : Runnable Job :10
pool-1-thread-1 : Runnable Job :11
pool-1-thread-1 : Runnable Job :12
pool-1-thread-1 : Runnable Job :13
pool-1-thread-1 : Runnable Job :14
pool-1-thread-1 : Runnable Job :15
pool-1-thread-1 : Runnable Job :16
pool-1-thread-1 : Runnable Job :17
pool-1-thread-1 : Runnable Job :18
pool-1-thread-1 : Runnable Job :19
pool-1-thread-1 : Runnable Job :20
```

Here Single Thread **pool-1-thread-1** is Executing 20 jobs, So Count is Correct & Order is good.

## 2. newFixedThreadPool

If we uncomment below line the Output should be Random, because 10 Threads executing 20 jobs. So output will be random, 10 threads will be executing.

```
ExecutorService executorService = Executors.newFixedThreadPool(10);
```

Below Thread names are different, maximum Thread name is **thread-10**.

```
pool-1-thread-1 : Runnable JOB :1
pool-1-thread-5 : Runnable JOB :1
pool-1-thread-4 : Runnable JOB :1
pool-1-thread-9 : Runnable JOB :3
pool-1-thread-6 : Runnable JOB :1
pool-1-thread-7 : Runnable JOB :5
pool-1-thread-3 : Runnable JOB :1
pool-1-thread-10 : Runnable JOB :8
pool-1-thread-2 : Runnable JOB :1
pool-1-thread-8 : Runnable JOB :6
pool-1-thread-1 : Runnable JOB :11
pool-1-thread-1 : Runnable JOB :12
pool-1-thread-1 : Runnable JOB :13
pool-1-thread-9 : Runnable JOB :5
pool-1-thread-4 : Runnable JOB :14
pool-1-thread-4 : Runnable JOB :16
pool-1-thread-5 : Runnable JOB :14
pool-1-thread-7 : Runnable JOB :17
pool-1-thread-6 : Runnable JOB :16
pool-1-thread-9 : Runnable JOB :16
```

## 3. newCachedThreadPool

If we uncomment below line the Output should be Random, and also there is no limit on Threads. So output will be random, threads created based up on tasks we are submitted.

```
ExecutorService executorService = Executors.newCachedThreadPool();
```

Below Thread names are different, maximum Thread name is **thread-13**.

```
pool-1-thread-4 : Runnable JOB :1
pool-1-thread-6 : Runnable JOB :1
pool-1-thread-7 : Runnable JOB :2
pool-1-thread-5 : Runnable JOB :1
pool-1-thread-3 : Runnable JOB :1
pool-1-thread-1 : Runnable JOB :1
pool-1-thread-2 : Runnable JOB :1
pool-1-thread-9 : Runnable JOB :6
pool-1-thread-11 : Runnable JOB :8
pool-1-thread-8 : Runnable JOB :3
pool-1-thread-10 : Runnable JOB :8
pool-1-thread-11 : Runnable JOB :12
pool-1-thread-6 : Runnable JOB :13
pool-1-thread-13 : Runnable JOB :12
pool-1-thread-12 : Runnable JOB :12
pool-1-thread-4 : Runnable JOB :12
pool-1-thread-9 : Runnable JOB :12
pool-1-thread-10 : Runnable JOB :12
pool-1-thread-11 : Runnable JOB :19
pool-1-thread-8 : Runnable JOB :20
```

## Example

```
class IntCall implements Callable<Integer> {
    @Override
    public Integer call() throws Exception {
        int value = (int) (Math.random() * 50 + 1);
        System.out.println("Generated : " + value);
        return value;
    }
}

public class ExecutorServiceDemo {
    public static void main(String[] args) throws Exception {

        // submit(Callable<T> task)
        ExecutorService service = Executors.newSingleThreadExecutor();
        Future<Integer> future = service.submit(new IntCall());
        System.out.println("Future : " + future.get());

        //invokeAll(Collection<Callable<T> tasks)
        System.out.println("===== invokeAll ===== ");
        List<Callable<Integer>> list = new ArrayList<>();
        list.add(new IntCall());
        list.add(new IntCall());
        list.add(new IntCall());

        List<Future<Integer>> futures = service.invokeAll(list);
        for (Future<Integer> f : futures) {
            System.out.println(f.get());
        }

        //Executes the given tasks, returns the result of one of the task completed successfully
        System.out.println("===== invokeAny ===== ");
        Integer any = service.invokeAny(list);
        System.out.println("invokeAny : " + any);

        // In general, the ExecutorService will not be automatically destroyed when
        // there is not task to process.
        service.shutdown();
        System.out.println("===== Shutdown ===== ");
        System.out.println("isShutdown : " + service.isShutdown());
        System.out.println("isTerminated : " + service.isTerminated());
        System.out.println("shutdownNow : " + service.shutdownNow());
    }
}
```

```
Generated: 34
Future: 34
===== invokeAll =====
Generated: 27
Generated: 24
Generated: 3
27
24
3
===== invokeAny =====
Generated : 32
Generated : 44
invokeAny : 32 (gives 1st completed task result)
===== Shutdown =====
isShutdown : true
isTerminated : true
shutdownNow : []
```

## 4.ScheduledExecutorService Interface

```
ScheduledExecutorService service = Executors.newSingleThreadScheduledExecutor();
ScheduledExecutorService service = Executors.newScheduledThreadPool(10);
```

- The `ScheduledExecutorService` interface is child interface `ExecutorService` .
- It executes a `Runnable` or `Callable` task after a specified delay. In addition, the interface defines `scheduleAtFixedRate` and `scheduleWithFixedDelay`, which executes specified tasks repeatedly, at defined intervals.
- The `ScheduledExecutorService` runs tasks after some predefined delay and/or periodically.

### 1.schedule()

There are two `schedule()` methods that allow you to execute `Runnable` or `Callable` tasks, which will start after the delay

```
ScheduledFuture schedule (Runnable command, long delay, TimeUnit unit)
ScheduledFuture schedule (Callable callable, long delay, TimeUnit unit)
```

### 2.scheduleAtFixedRate() method lets execute a task periodically after a fixed delay

```
scheduleAtFixedRate(Runnable r, long initialDelay, long period, TimeUnit u)
```

The following block of code will execute a task after an initial delay of 100 milliseconds, and after that, it will execute the same task every 450 milliseconds.

```
Future<String> resultFuture = service
    .scheduleAtFixedRate(runnableTask, 100, 450, TimeUnit.MILLISECONDS);
```

If the processor needs more time to execute an assigned task than the period parameter of the `scheduleAtFixedRate()` method, the `ScheduledExecutorService` will wait until the current task is completed before starting the next.

**3.scheduleWithFixedDelay()** If it is necessary to have a fixed length delay between iterations of the task, `scheduleWithFixedDelay()` should be used. For example, the following code will guarantee a 150-millisecond pause between the end of the current execution and the start of another one.

```
ScheduledFuture scheduleWithFixedDelay (Runnable command, long initialDelay, long delay, TimeUnit unit)
```

below `scheduleWithFixedDelay`, `scheduleAtFixedRate` methods are applicable only for `Runnable` types but not `Callable` Types

```
service.scheduleWithFixedDelay(task, 100, 150, TimeUnit.MILLISECONDS);
```

```
ScheduledExecutorService executor = ...;
Runnable command1 = ...;
Runnable command2 = ...;
Runnable command3 = ...;

// Will start command1 after 50 seconds
executor.schedule(command1, 50L, TimeUnit.SECONDS);

// Will start command2 after 20 seconds, 25 seconds, 30 seconds ...
executor.scheduleAtFixedRate(command2, 20L, 5L, TimeUnit.SECONDS);

// Will start command3 after 10 seconds and if command3 takes 2 seconds to be
// executed also after 17, 24, 31, 38 seconds...
executor.scheduleWithFixedDelay(command3, 10L, 5L, TimeUnit.SECONDS);
```

## Example

```

class IntCal implements Callable<Integer> {
    @Override
    public Integer call() throws Exception {
        int value = (int) (Math.random() * 50 + 1);
        System.out.println("Generated : " + value);
        return value;
    }
}

class MyRun implements Runnable{
    public void run() {
        System.out.println("Run End @ : "+new Date());
    }
}

public class ScheduledExecutorServiceDemo {

    public static void main(String[] args) throws InterruptedException, ExecutionException {

        ScheduledExecutorService service = Executors.newSingleThreadScheduledExecutor();

        System.out.println("==== schedule : Start : "+new Date());
        Future f1 = service.schedule(new IntCal(), 5, TimeUnit.SECONDS);
        System.out.println("==== schedule get():"+f1.get()+" End : ");

        System.out.println("==== scheduleAtFixedRate : Start : "+new Date());
        service.scheduleAtFixedRate(new MyRun(), 2, 2, TimeUnit.SECONDS);

        System.out.println("==== scheduleWithFixedDelay : Start : "+new Date());
        service.scheduleWithFixedDelay(new MyRun(), 2, 2, TimeUnit.SECONDS);
    }
}

```

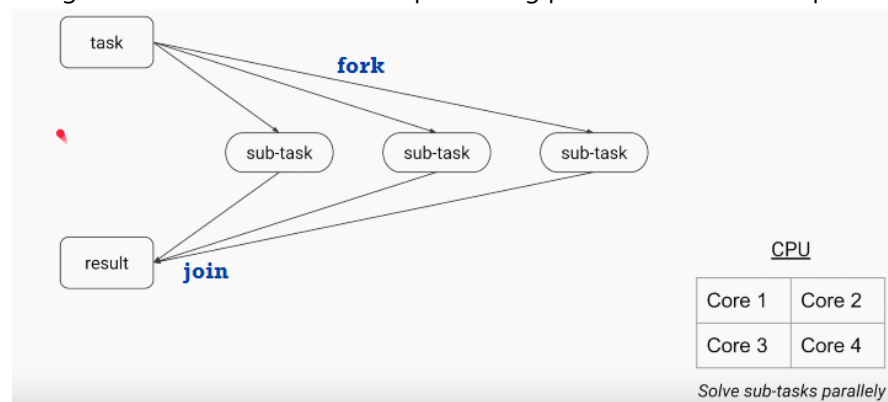
```

==== schedule : Start : Wed Dec 26 18:16:15 IST 2018
Generated : 15
==== schedule get():15: End :
==== scheduleAtFixedRate : Start : Wed Dec 26 18:16:21 IST 2018
==== scheduleWithFixedDelay : Start : Wed Dec 26 18:16:21 IST 2018
Run End @ : Wed Dec 26 18:16:23 IST 2018
Run End @ : Wed Dec 26 18:16:23 IST 2018
Run End @ : Wed Dec 26 18:16:25 IST 2018
Run End @ : Wed Dec 26 18:16:25 IST 2018
Run End @ : Wed Dec 26 18:16:27 IST 2018
Run End @ : Wed Dec 26 18:16:27 IST 2018

```

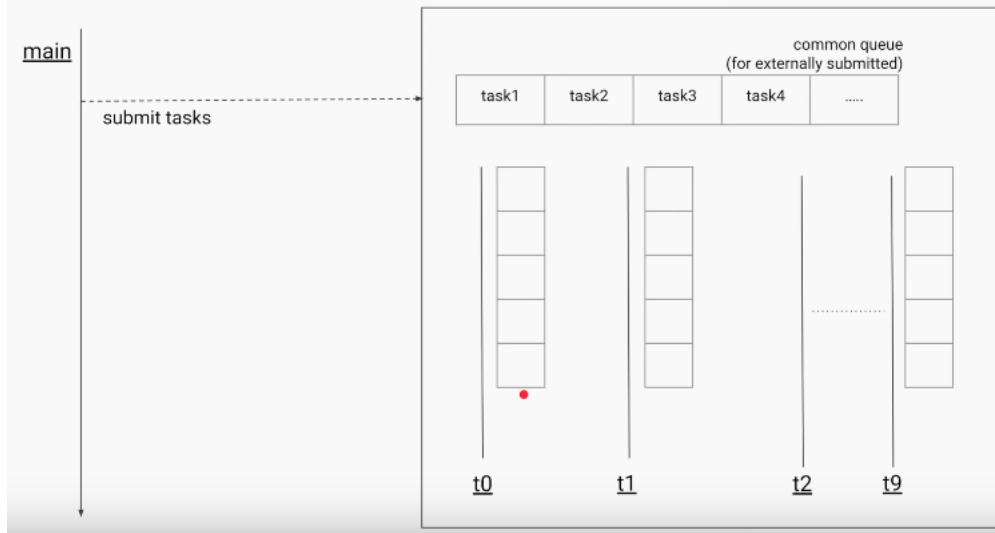
## 5.Fork/Join

The fork/join framework is an implementation of the [ExecutorService](#) interface that helps you take advantage of multiple processors. It is designed for **work can be broken into smaller pieces recursively**. The goal is to use all the available processing power to enhance the performance of your application





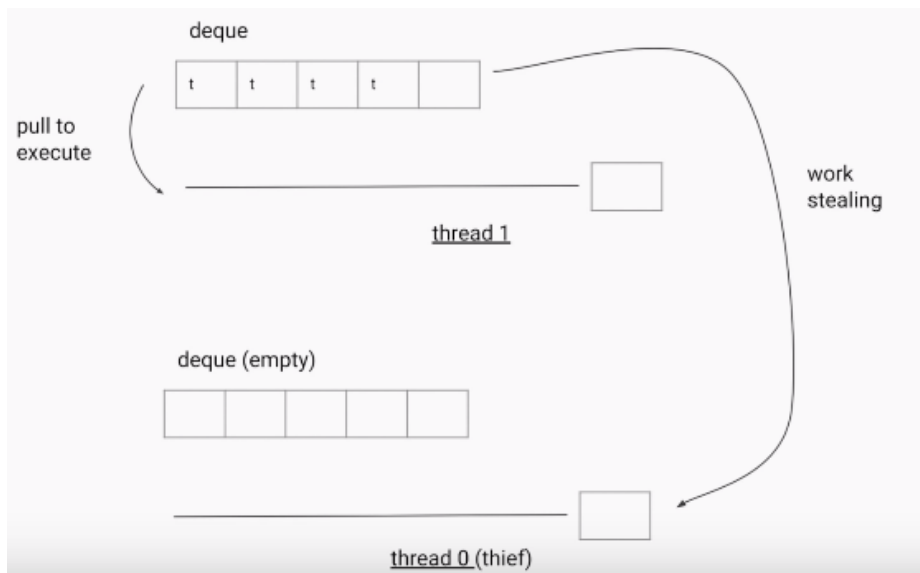
Here Each of the Thread has it's own Queue(DeQue – Double ended Queue).



When on submitting task, the task is divided into no. of sub-tasks (4 subtasks for example), each sub-task will store in one of the locations in DeQue like below



If a Thread doesn't have any tasks to execute, it will steal the tasks from other threads from backside of Queue.



- Just keep picking tasks from own queue
- No blocking (unless during stealing)
- Easier scheduling

	Call from non-fork/join clients	Call from within fork/join computations
Arrange async execution	<code>execute(ForkJoinTask)</code>	<code>fork()</code>
Await and obtain result	<code>invoke(ForkJoinTask)</code>	<code>invoke()</code>
Arrange exec and obtain Future	<code>submit(ForkJoinTask)</code>	<code>fork()</code> (ForkJoinTasks are Futures)

The core classes supporting the Fork-Join mechanism are [ForkJoinPool](#) and [ForkJoinTask](#)

### ForkJoinPool

The ForkJoinPool is basically a specialized implementation of ExecutorService implementing the work-stealing algorithm we talked about above. We create an instance of [ForkJoinPool](#) by providing the target parallelism level i.e. the number of processors as shown below:

```
ForkJoinPool pool = new ForkJoinPool(numberOfProcessors);
Where numberOfProcessors = Runtime.getRuntime().availableProcessors();
```

If you use a no-argument constructor, by default, it creates a pool of size that equals the number of available processors obtained using above technique.

There are three different ways of submitting a task to the [ForkJoinPool](#).

- **execute()** method //Desired asynchronous execution; call its fork method to split the work between multiple threads.
- **invoke()** method: //Await to obtain the result; call the invoke method on the pool.
- **submit()** method: //Returns a Future object that you can use for checking status and obtaining the result on its completion.

### ForkJoinTask

This is an abstract class for creating tasks that run within a [ForkJoinPool](#).

- The [RecursiveAction](#) and [RecursiveTask](#) are the only two direct, known subclasses of [ForkJoinTask](#).
- The only difference between these two classes is that the [RecursiveAction](#) does not return a value while [RecursiveTask](#) does have a return value and returns an object of specified type.
- In both cases, you would need to implement the compute method in your subclass that performs the main computation desired by the task.

The [ForkJoinTask](#) class provides several methods for checking the execution status of a task.

- The **isDone()** method returns true if a task completes in any way
- The **isCompletedNormally()** method returns true if a task completes without cancellation or encountering an exception
- **isCancelled()** returns true if the task was cancelled. Lastly, **isCompletedabnormally()** returns true if the task was either cancelled or encountered an exception.

```

public class ForkJoinDemo {
    public static void main(final String[] arguments) throws Exception {
        int nThreads = Runtime.getRuntime().availableProcessors();
        System.out.println("No.of Threads : "+nThreads);

        int[] numbers = new int[1000];
        for (int i = 0; i < numbers.length; i++) {
            numbers[i] = i;
        }

        ForkJoinPool forkJoinPool = new ForkJoinPool(nThreads);
        Long result = forkJoinPool.invoke(new Sum(numbers, 0, numbers.length));
        System.out.println("Sum of 1000 num : "+result);
    }

    static class Sum extends RecursiveTask<Long> {
        int low;
        int high;
        int[] array;

        Sum(int[] array, int low, int high) {
            this.array = array;
            this.low = low;
            this.high = high;
        }
        protected Long compute() {
            if (high - low <= 10) {
                long sum = 0;

                for (int i = low; i < high; ++i)
                    sum += array[i];
                return sum;
            } else {
                int mid = low + (high - low) / 2;
                Sum left = new Sum(array, low, mid);
                Sum right = new Sum(array, mid, high);
                left.fork();
                long rightResult = right.compute();
                long leftResult = left.join();
                return leftResult + rightResult;
            }
        }
    }
}

```

```

No.of Threads : 2
Sum of 1000 num : 499500

```

## 6. Custom ThreadPool

If you see the Source code of `ExecutorService`, we have below method with parameters.

```

ExecutorService service = Executors.newFixedThreadPool( nThreads: 10);

```

↓

```

public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
        keepAliveTime: 0L, TimeUnit.MILLISECONDS,
        new LinkedBlockingQueue<Runnable>());
}

```

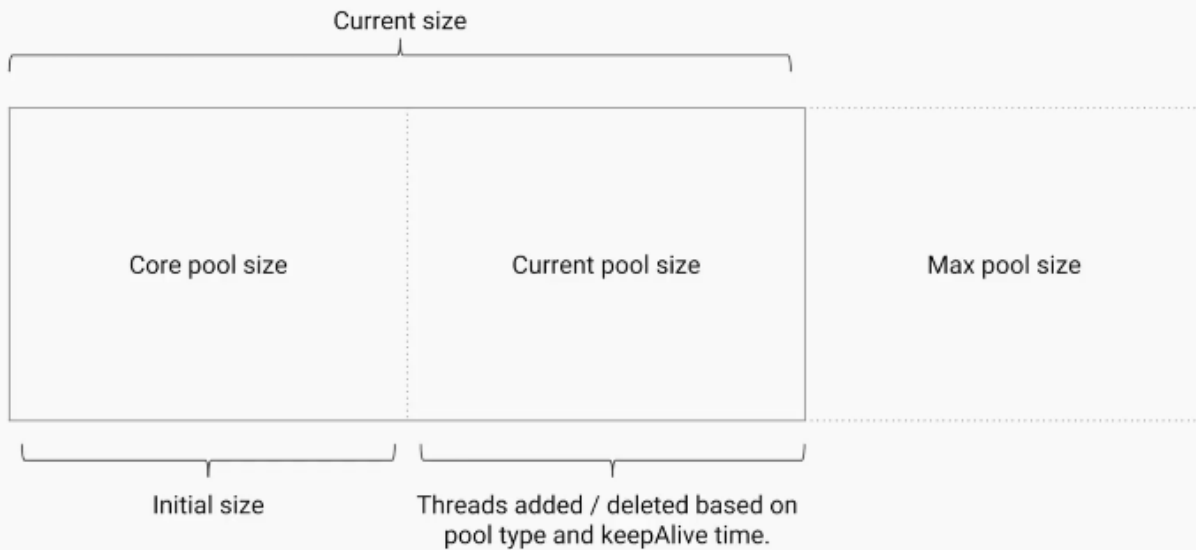
↓

```

public ThreadPoolExecutor(int corePoolSize,
    int maximumPoolSize,
    long keepAliveTime,
    TimeUnit unit,
    BlockingQueue<Runnable> workQueue,
    ThreadFactory threadFactory,
    RejectedExecutionHandler handler) {

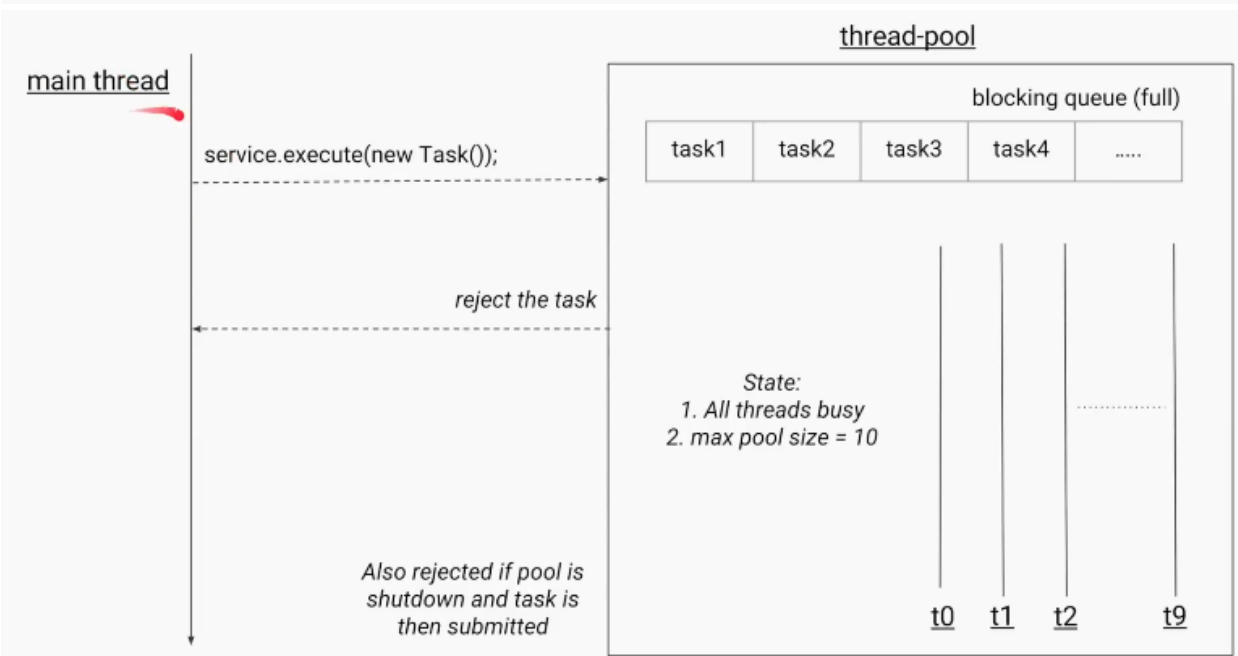
```

Parameter	Type	Meaning
corePoolSize	int	Minimum/Base size of the pool
maxPoolSize	int	Maximum size of the pool
keepAliveTime + unit	long	Time to keep an idle thread alive (after which it is killed)
workQueue	BlockingQueue	Queue to store the tasks from which threads fetch them
threadFactory	ThreadFactory	The factory to use to create new threads
handler	RejectedExecutionHandler	Callback to use when tasks submitted are rejected



Parameter	FixedThreadPool	CachedThreadPool	ScheduledThreadPool	SingleThreaded
corePoolSize	constructor-arg	0	constructor-arg	1
maxPoolSize	same as corePoolSize	Integer.MAX_VALUE	Integer.MAX_VALUE	1
keepAliveTime	0 seconds (NA)	60 seconds	60 seconds	0 seconds (NA)

Pool	Queue Type	Why?
FixedThreadPool	LinkedBlockingQueue	Threads are limited, thus unbounded queue to store all tasks.
SingleThreadExecutor	LinkedBlockingQueue	<i>Note: Since queue can never become full, new threads are never created.</i>
CachedThreadPool	SynchronousQueue	Threads are unbounded, thus no need to store the tasks. Synchronous queue is a queue with single slot
ScheduledThreadPool	DelayedWorkQueue	Special queue that deals with schedules/time-delays
Custom	ArrayBlockingQueue	Bounded queue to store the tasks. If queue gets full, new thread is created (as long as count is less than maxPoolSize).



Policy	What it means?
AbortPolicy	Submitting new tasks throws RejectedExecutionException (Runtime exception)
DiscardPolicy	Submitting new tasks silently discards it.
DiscardOldestPolicy	Submitting new tasks drops existing oldest task, and new task is added to the queue.
CallerRunsPolicy	Submitting new tasks will execute the task on the caller thread itself. This can create feedback loop where caller thread is busy executing the task and cannot submit new tasks at fast pace.

By Defining above parameters properly we can create our own ThreadPoolExecutor.

```

ExecutorService service
    = new ThreadPoolExecutor(
        corePoolSize: 10,
        maximumPoolSize: 100,
        keepAliveTime: 120, TimeUnit.SECONDS,
        new ArrayBlockingQueue<>( capacity: 300));

try {
    service.execute(new Task());
} catch (RejectedExecutionException e) {
    System.err.println("task rejected " + e.getMessage());
}

```

## Synchronization utilities – more options for doing Synchronization

### 1.Semaphore(CountDownSemaphore)

**Semaphore** is one of the synchronization aid provided by Java concurrency util in Java 5 along with other synchronization aids like **CountDownLatch**, **CyclicBarrier**, **Phaser** and **Exchanger**.

The Semaphore **class** present in `java.util.concurrent` package is a counting-semaphore in which a semaphore conceptually maintains a set of **permits**.

Semaphore **class** in Java has two methods that make use of permits-

- **acquire()**- Acquires a permit from **this** semaphore, blocking until one is available, or the thread is interrupted. It has another overloaded version `acquire(int permits)`.
- **release()**- Releases a permit, returning it to the semaphore. It has another overloaded method `release(int permits)`.

### How Semaphore works in Java

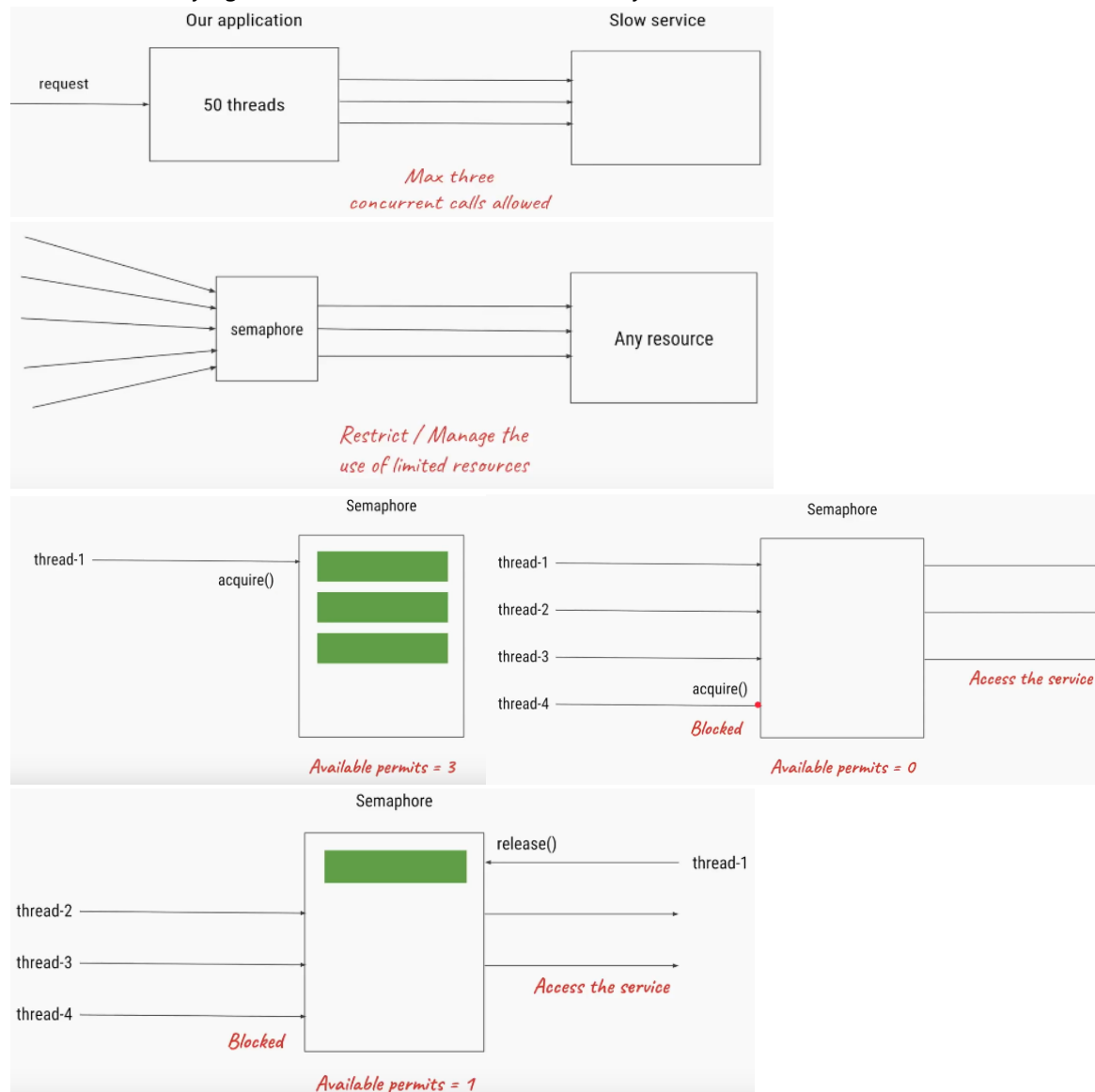
- Thread that wants to access the shared resource tries to acquire a permit using `acquire()` method. At that time if the Semaphore's count is greater than zero thread will acquire a permit and Semaphore's count will be decremented by one.
- If Semaphore's count is zero and thread calls `acquire()` method, then the thread will be blocked until a permit is available.
- When thread is done with the shared resource access, it can call the `release()` method to release the permit. That results in the Semaphore's count incremented by one.

### Java Semaphore constructors

- `Semaphore(int permits)`
- `Semaphore(int permits, boolean fair)`

## Scenario

50 threads are trying to access a slow service , where only 3 threads are allowed.



Method	Meaning
tryAcquire	Try to acquire, if no permit available, do not block. Continue doing something else.
tryAcquire (timeout)	Same as above but with timeout
availablePermits	Returns count of permits available
new Semaphore (count, fairness)	FIFO. Fairness guarantee for threads waiting the longest.

## Java semaphore example

Let's see one example where Semaphore is used to control shared access. Here we have a shared counter and three threads using the same shared counter and trying to increment and then again decrement the count. So every thread should first increment the count to 1 and then again decrement it to 0.

```

class SharedCounter implements Runnable{
    private int c = 0;
    private Semaphore s;

    SharedCounter(Semaphore s){
        this.s = s;
    }

    // incrementing the value
    public void increment() {
        try {
            // used sleep for context switching
            Thread.sleep(10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        c++;
    }

    // decrementing the value
    public void decrement() {
        c--;
    }
    public int getValue() {
        return c;
    }

    @Override
    public void run() {
        try {
            // acquire method to get one permit
            s.acquire();
            this.increment();
            SOP("Value for Thread After increment-"+Thread.currentThread().getName()+" " +this.getValue());

            this.decrement();
            SOP("Value for Thread at last"+Thread.currentThread().getName() + " " + this.getValue());
            // releasing permit
            s.release();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

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

        Semaphore s = new Semaphore(1);
        SharedCounter counter = new SharedCounter(s);
        // Creating three threads
        Thread t1 = new Thread(counter, "Thread-A");
        Thread t2 = new Thread(counter, "Thread-B");
        Thread t3 = new Thread(counter, "Thread-C");
        t1.start();
        t2.start();
        t3.start();
    }
}

```

```

Value for Thread After increment - Thread-A 1
Value for Thread at last Thread-A 0
Value for Thread After increment - Thread-B 1
Value for Thread at last Thread-B 0
Value for Thread After increment - Thread-C 1
Value for Thread at last Thread-C 0

```

Semaphore is Similar to Lock , and **lock()** & **unlock()** methods similar to **acquire()** & **release()**



## 2.Binary Semaphore(Mutex)

A semaphore initialized to one, and which is used such that it only has at most one permit available, can serve as a mutual exclusion lock. This is more commonly known as a binary semaphore, because it only has two states: **one permit available, or zero permits available.**

When used in this way, the binary semaphore has the property (unlike many Lock implementations), that the "lock" can be released by a thread other than the owner (as semaphores have no notion of ownership). This can be useful in some specialized contexts, such as deadlock recovery.

**Mutex** – Only one thread to access a resource at once. Example, when a client is accessing a file, no one else should have access the same file at the same time.

Mutex is the Semaphore with an access count of 1. Consider a situation of using lockers in the bank. Usually the rule is that only one person is allowed to enter the locker room.

```
public class SemaphoreTest {

    // max 1 people
    static Semaphore semaphore = new Semaphore(1);

    // Inner Class
    static class MyLockerThread extends Thread {
        String name = "";
        MyLockerThread(String name) {
            this.name = name;
        }

        public void run() {
            try {
                SOP(name + ":acquiring lock...");
                SOP(name + ":available Semaphore permits Now: "+ semaphore.availablePermits());
                semaphore.acquire();

                System.out.println(name + " : got the permit!");

                try {
                    for (int i = 1; i <= 5; i++) {
                        SOP(name + " : is performing operation " + i);
                        SOP("available Semaphore permits : "+ semaphore.availablePermits());

                        // sleep 1 second
                        Thread.sleep(1000);
                    }
                } finally {
                    // calling release() after a successful acquire()
                    SOP (name + " : releasing lock...");
                    semaphore.release();
                    SOP(name +":available Semaphore permits:"+semaphore.availablePermits());
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        System.out.println("Total available Semaphore permits : " + semaphore.availablePermits());

        MyLockerThread t1 = new MyLockerThread("A");
        t1.start();

        MyLockerThread t2 = new MyLockerThread("B");
        t2.start();

        MyLockerThread t3 = new MyLockerThread("C");
        t3.start();
    }
}
```

```

MyLockerThread t4 = new MyLockerThread("D");
t4.start();

MyLockerThread t5 = new MyLockerThread("E");
t5.start();

MyLockerThread t6 = new MyLockerThread("F");
t6.start();
}

```

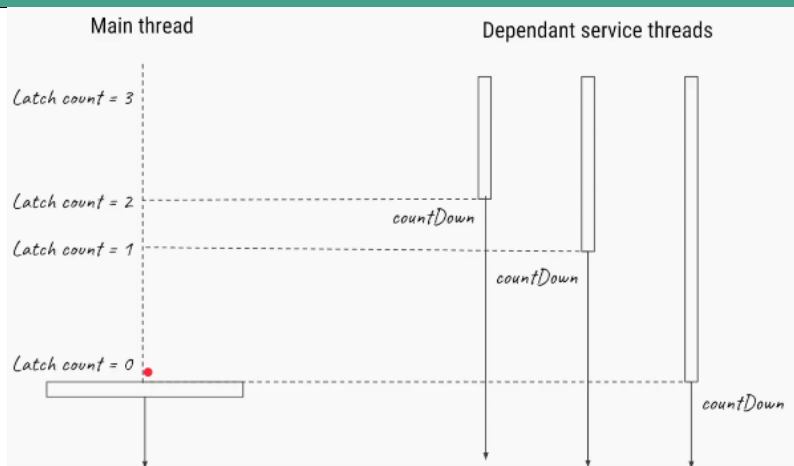
### 3.CountDownLatch

CountDownLatch is a synchronization mechanism. If there is a situation like All Threads must complete `login()`, `getAccountNumber()` execution before actual execution, in that case it allows one or more threads to wait until a these operations completed by remaining threads.

Methods (latch - తలుపునకు వేయు గడియ)

- `await()`
- `countDown()`

#### Scenario



```

public static void main(String[] args) throws InterruptedException {
    ExecutorService executor = Executors.newFixedThreadPool( nThreads: 4);

    CountDownLatch latch = new CountDownLatch(3);
    executor.submit(new DependentService(latch));
    executor.submit(new DependentService(latch));
    executor.submit(new DependentService(latch));

    latch.await();

    System.out.println("All dependant services initialized");
    // program initialized, perform other operations
}

public static class DependentService implements Runnable {
    private CountDownLatch latch;
    public DependentService(CountDownLatch latch) { this.latch = latch; }

    @Override
    public void run() {
        // startup task
        latch.countDown();
        // continue w/ other operations
    }
}

```

```

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

        final CountdownLatch latch = new CountdownLatch(3);
        Thread cacheService = new Thread(new Service("CacheService", 1000, latch));
        Thread alertService = new Thread(new Service("AlertService", 1000, latch));
        Thread validationService = new Thread(new Service("ValidationService", 1000, latch));

        cacheService.start(); // separate thread will initialize CacheService
        alertService.start(); // another thread for AlertService initialization
        validationService.start();

        //application should not start processing any thread until all service is up and ready to do their job.
        // Countdown latch is idle choice here, main thread will start with count 3 and wait until count reaches
        // zero. each thread once up and read will do a count down. this will ensure that main thread is not started
        // processing until all services is up.

        // count is 3 since we have 3 Threads (Services)
        try {
            latch.await(); // main thread is waiting on CountdownLatch to finish
            System.out.println("All services are up, Application is starting now");
        } catch (InterruptedException ie) {
            ie.printStackTrace();
        }
    }
}

/* Service class which will be executed by Thread using CountdownLatch * synchronizer. */
class Service implements Runnable {
    private final String name;
    private final int timeToStart;
    private final CountdownLatch latch;

    public Service(String name, int timeToStart, CountdownLatch latch) {
        this.name = name;
        this.timeToStart = timeToStart;
        this.latch = latch;
    }

    @Override
    public void run() {
        try {
            Thread.sleep(timeToStart);
        } catch (InterruptedException ex) {
        }
        System.out.println(name + " is Up");
        latch.countDown(); // reduce count of CountdownLatch by 1
    }
}

```

#### 4.CyclicBarrier

CyclicBarrier is a synchronization mechanism that allows a set of threads to **all wait for each other to reach a common barrier point.**

First a **new** instance of a CyclicBarrier is created specifying the number of threads that the barriers should wait upon.

```
CyclicBarrier newBarrier = new CyclicBarrier(numberOfThreads);
```

#### Methods

- **await()**

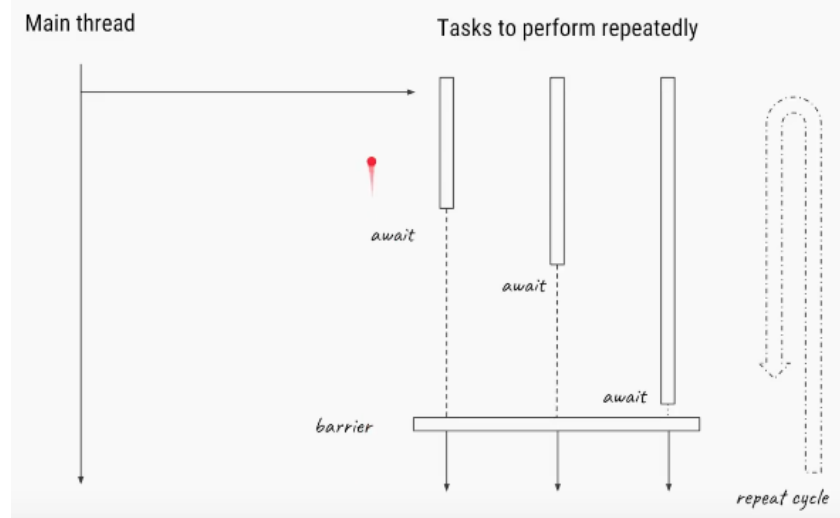
Each and every thread does some computation and after completing it's execution, calls `await()` methods as shown:

```
public void run()
{
    // thread does the computation
    newBarrier.await();
}
```

Once the number of threads that called `await()` equals `numberOfThreads`, the barrier then gives a way for the waiting threads. The `CyclicBarrier` can also be initialized with some action that is performed once all the threads have reached the barrier. This action can combine/utilize the result of computation of individual thread waiting in the barrier.

```
Runnable action = ...
//action to be performed when all threads reach the barrier;
CyclicBarrier newBarrier = new CyclicBarrier(numberOfThreads, action);
```

### Scenario



```
public static void main(String[] args) throws InterruptedException {
    ExecutorService executor = Executors.newFixedThreadPool( nThreads: 4);

    CyclicBarrier barrier = new CyclicBarrier( parties: 3);
    executor.submit(new Task(barrier));
    executor.submit(new Task(barrier));
    executor.submit(new Task(barrier));

    Thread.sleep( millis: 2000);
}

public static class Task implements Runnable {
    private CyclicBarrier barrier;
    public Task(CyclicBarrier barrier) { this.barrier = barrier; }

    @Override
    public void run() {
        while (true) {
            try {
                barrier.await();
            } catch (InterruptedException | BrokenBarrierException e) {
                e.printStackTrace();
            }
            // send message to corresponding system
        }
    }
}
```

```

class Task implements Runnable {

    private CyclicBarrier barrier;

    public Task(CyclicBarrier barrier) {
        this.barrier = barrier;
    }

    @Override
    public void run() {
        try {
            System.out.println(Thread.currentThread().getName() + " is waiting on barrier");
            barrier.await();
            System.out.println(Thread.currentThread().getName() + " COMPLETED");
        } catch (InterruptedException ex) {
        } catch (Exception ex) {
        }
    }
}

public class CyclicBarrierExample {

    public static void main(String args[]) {
        // creating CyclicBarrier with 3 Threads which, meet at this point
        final CyclicBarrier cb = new CyclicBarrier(3, new Runnable() {
            @Override
            public void run() {
                // This task will be executed once all thread reaches barrier
                System.out.println("=====");
                System.out.println("All parties are arrived at barrier, lets play");
                System.out.println("=====");
            }
        });
        // starting each of thread
        Thread t1 = new Thread(new Task(cb), "Thread 1");
        Thread t2 = new Thread(new Task(cb), "Thread 2");
        Thread t3 = new Thread(new Task(cb), "Thread 3");

        t1.start();
        t2.start();
        t3.start();
    }
}

```

```

Thread 1 is waiting on barrier
Thread 3 is waiting on barrier
Thread 2 is waiting on barrier
=====
All parties are arrived at barrier, lets play
=====
Thread 2 COMPLETED
Thread 3 COMPLETED
Thread 1 COMPLETED

```

## 5. Phaser

Phaser is like a **collection of advantages of CountdownLatch and CyclicBarrierClasses**

### The CountdownLatch is :

- Created with a fixed number of threads

```
final CountdownLatch latch = new CountdownLatch(3);
```

- Cannot be reset

- Allows threads to wait (`await()`) or continue with its execution once count becomes 0 (`countDown()`)

### The CyclicBarrier is:

- Can be reset.
- Does not provide a method for the threads to advance. The threads have to wait till all the threads arrive.
- Created with fixed number of threads.

### Now, the Phaser has following properties :

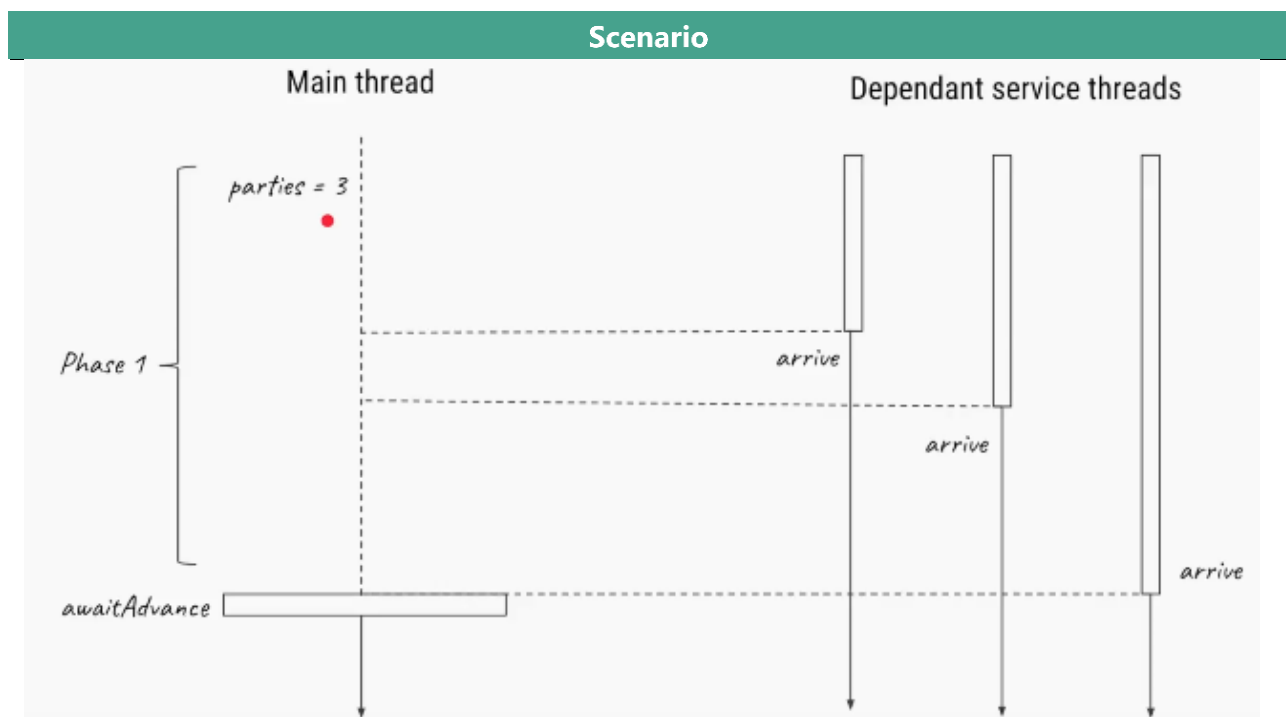
- Number of threads need not be known at Phaser creation time. **Threads can be added dynamically.**
- **Can be reset** and hence is, reusable.
- Allows threads to wait(Phaser#**arriveAndAwaitAdvance()**) or continue with its execution(Phaser#**arrive()**).
- Supports multiple Phases(, hence the name phaser).

We will try to understand how the Phaser Class can be used with an example. In this example, we are creating a three threads, which will wait for the arrival all the threads being created.

Once all the threads have arrived(marked by **arriveAndAwaitAdvance()** method) the Phaser allows them through the barrier.

### Methods

- **awaitAdvance()**
- **arrive()**



```

public static void main(String[] args) throws InterruptedException {
    ExecutorService executor = Executors.newFixedThreadPool( nThreads: 4);

    Phaser phaser = new Phaser( parties: 3);
    executor.submit(new DependentService(phaser));
    executor.submit(new DependentService(phaser));
    executor.submit(new DependentService(phaser));

    phaser.awaitAdvance( phase: 1); Similar to await()

    System.out.println("All dependant services initialized");
    // program initialized, perform other operations
}

public static class DependentService implements Runnable {
    private Phaser phaser;
    public DependentService(Phaser phaser) { this.phaser = phaser; }

    @Override
    public void run() {
        // startup task
        phaser.arrive(); Similar to countdown()
        // continue w/ other operations
    }
}

```

new Phaser(partyCount)	Set count of participating parties
register	Allow party to register itself
bulkRegister(partyCount)	Bulk register extra parties post constructor
arrive	Parties can arrive (and continue)
arriveAndAwaitAdvance	Parties can arrive and await for all parties
arriveAndDeregister	Parties can arrive, deregister (and continue)

```

public class PhaserExample {
    public static void main(String[] args) throws InterruptedException {
        Phaser phaser = new Phaser();
        phaser.register();// register self... phaser waiting for 1 party (thread)
        int phasecount = phaser.getPhase();
        System.out.println("Phasecount is " + phasecount);
        new PhaserExample().testPhaser(phaser, 2000);// phaser waiting for 2 parties
        new PhaserExample().testPhaser(phaser, 4000);// phaser waiting for 3 parties
        new PhaserExample().testPhaser(phaser, 6000);// phaser waiting for 4 parties

        // now that all threads are initiated, we will de-register main thread
        // so that the barrier condition of 3 thread arrival is meet.
        phaser.arriveAndDeregister();
        Thread.sleep(10000);
        phasecount = phaser.getPhase();
        System.out.println("Phasecount is " + phasecount);
    }

    private void testPhaser(final Phaser phaser, final int sleepTime) {
        phaser.register();
        new Thread() {

```

```

        @Override
        public void run() {
            try {
                System.out.println(Thread.currentThread().getName() + " arrived");
                phaser.arriveAndAwaitAdvance();// threads register arrival to the phaser.
                Thread.sleep(sleepTime);
            }
            catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(Thread.currentThread().getName() + " after passing barrier");
        }
    }.start();
}
}

```

## 6.Exchanger

The Exchanger Class provides a sort of meeting point for two threads, **where the threads can exchange their respective Objects with the other thread.**

Whenever a thread arrives at the exchange point, it must wait for the other thread to arrive. When the other pairing thread arrives the two threads proceed to exchange their objects.

The Exchanger Class also provides an overloaded version of the parameterless **exchange()** method, **exchange(V x, long timeout, TimeUnit unit).**

When the exchange method with time-out is used, the waiting thread waits for the period passed as the argument(**long timeout**). If the corresponding pairing thread does not arrive at the exchange point in that time, the waiting Thread throws a **java.util.concurrent.TimeoutException.**

```

public class ExchangerExample {

    Exchanger exchanger = new Exchanger();

    private class Producer implements Runnable {
        private String queue;

        @Override
        public void run() {
            try {
                // create tasks & fill the queue exchange the full queue for a
                // empty queue with Consumer
                queue = (String) exchanger.exchange("Ready Queue");
                SOP(Thread.currentThread().getName() + " : now has " + queue);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    private class Consumer implements Runnable {
        private String queue;

        @Override
        public void run() {
            try {
                // do procesing & empty the queue exchange the empty queue for a
                // full queue with Producer
                queue = (String) exchanger.exchange("Empty Queue");
                SOP(Thread.currentThread().getName() + ":now has " + queue);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

```



```

private void start() {
    new Thread(new Producer(), "Producer").start();
    new Thread(new Consumer(), "Consumer").start();
}

public static void main(String[] args) {
    new ExchangerExample().start();
}
}

```

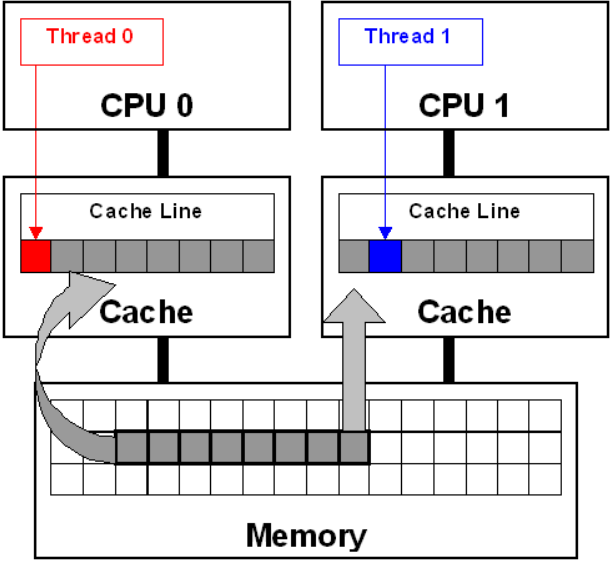
Consumer: now has Ready Queue  
 Producer : now has Empty Queue

- In the above example, we create an Exchanger Object of the type String.
- The Producer thread produces a "filled queue" and exchanges it with the Consumer thread for an "empty queue".
- (The filled and empty queue mentioned here are just dummy string object, for the sake of brevity.). Similarly, we can proceed to exchange any type of object between two threads, merely by changing the type parameter of Exchanger instance.

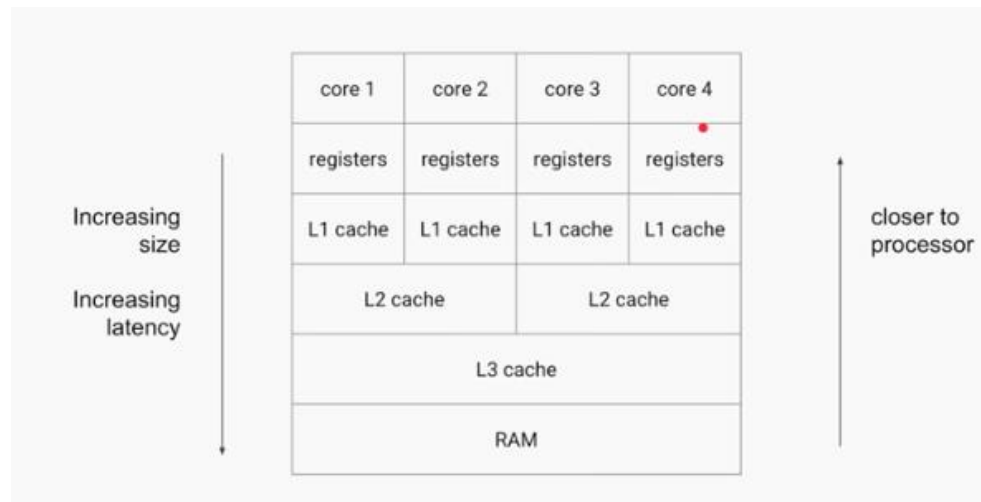
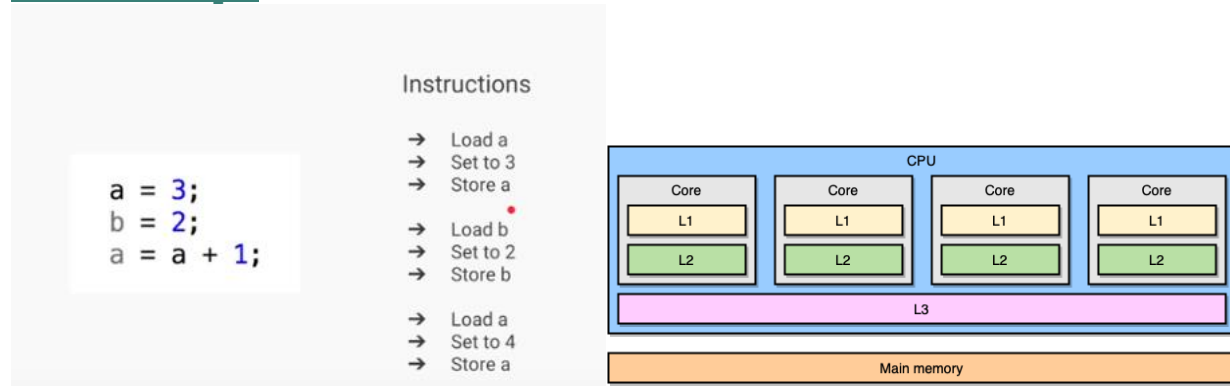
## Atomic Variables

### What is false sharing in the context of multi-threading?

false sharing is one of the well-known performance issues on multi-core systems, where each process has its local cache.



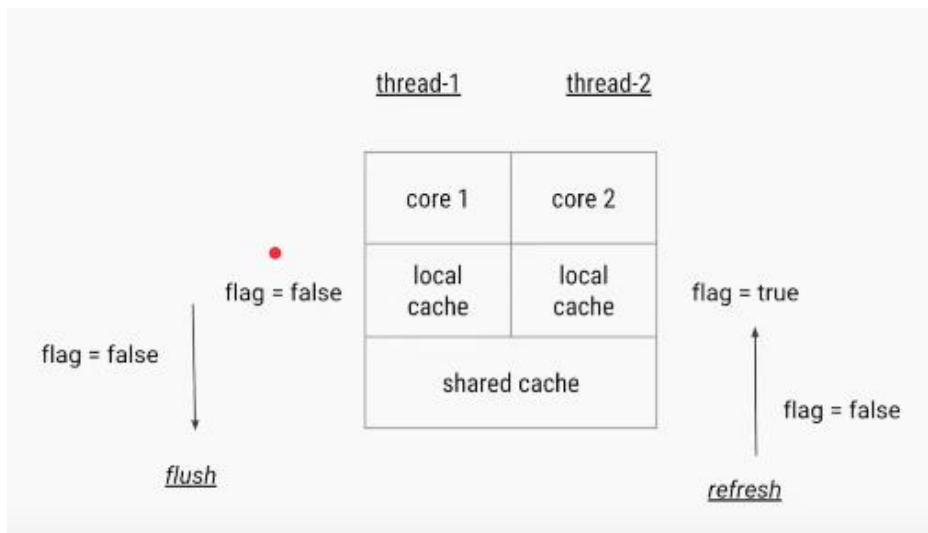
## Volatile Example



- If **writerThread()** is executed by one thread & **readerThread()** is executed by another thread the 'x' value is different for two threads because they are reading value from their LocalCache.
- Here the changes of X value is not **visible** globally (Field Visibility), because they are changing in **LocalCache**.

**To avoid this, we need to use 'volatile' keyword for fields.**

- The Java **volatile** keyword is used to mark a Java variable as "**being stored in main memory**".
- that means, every **read** of a volatile variable will **be read from the main memory (Shared Memory)**, and not from the CPU cache
- **every write** to a volatile variable will be **written to main memory**, and not just to the CPU cache.

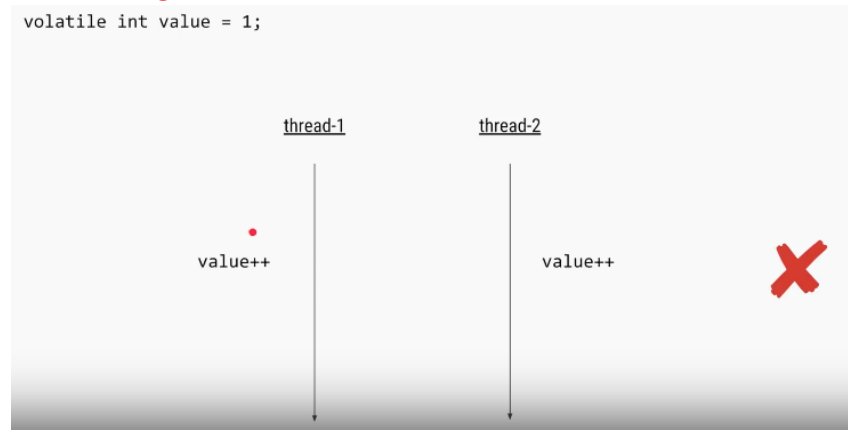


- In the above diagram two threads t1, t2 are trying to change the value of `flag`.
- If Thread1 changes value `flag=false`, then it will flush(push) the changes from LocalCache to SharedCache and it will refresh all Thread LocalCaches with updated value.
- If Thread 2 is trying to read, it will get updated value.
- Volatile solves the visibility problem, where Only one operation is performed.

### Atomic Problem

Atomic - forming a single irreducible unit or component in a larger system.

Increment (`++`) is a **Compound Operation(multiple)**. **AtomicVariables** makes compound operations as **Atomic (Single)**



```
volatile int count = 1;
        count = count++;
```

Even with volatile

#	Thread-1	Thread-2
1	Read value (=1)	
2		Read value (=1)
3	Add 1 and write (=2)	
4		Add 1 and write (=2)

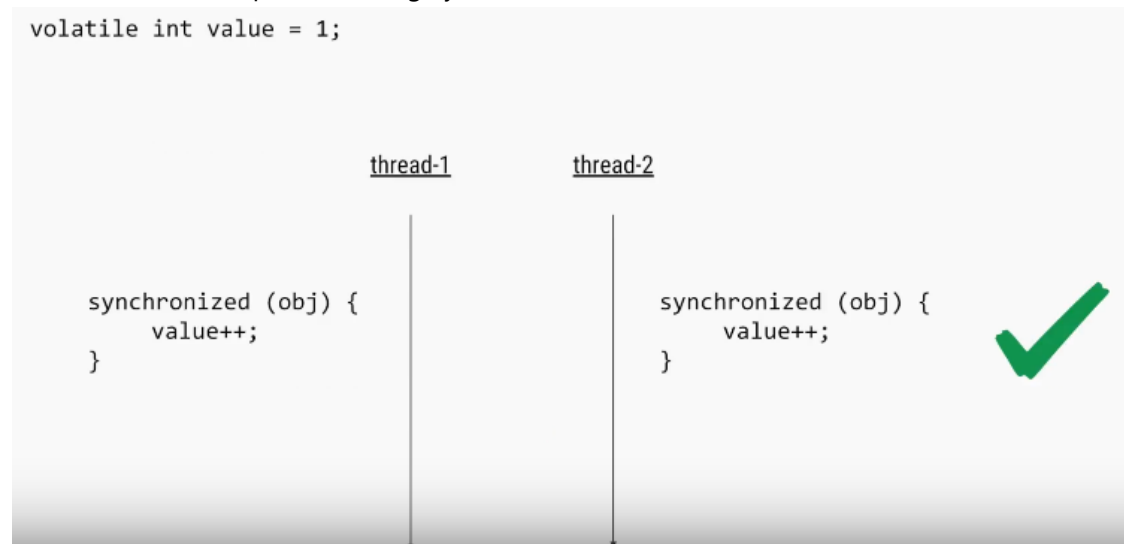


Here volatile works good on Single Operation. When we do Increment like `count++`

- First operation- it will add +1, one operation completed & it flush changes to SharedCache – but if any thread access `count` value, it still shows `count`= 1, NOT 2
- Second Operation- it will store the incremented `count` value to 2

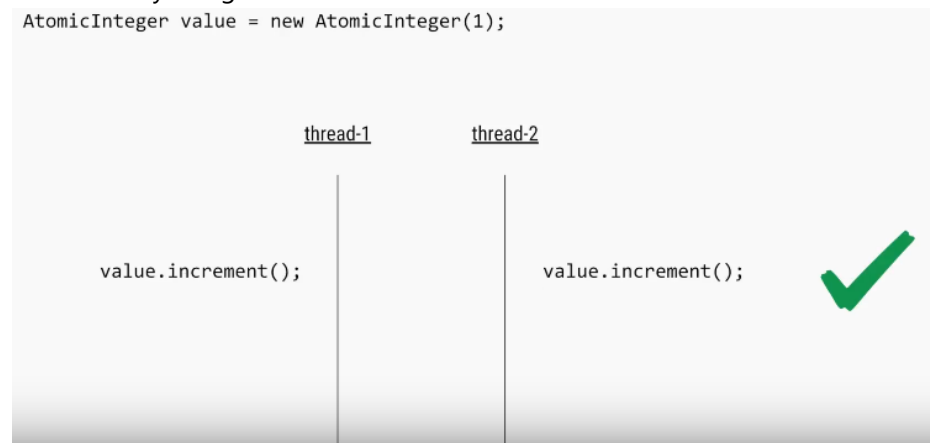
We can solve above problem using Synchronization

```
volatile int value = 1;
```



Another Way using Atomic Variables

```
AtomicInteger value = new AtomicInteger(1);
```



Type	Use Case
volatile	Flags
AtomicInteger AtomicLong	Counters
AtomicReference	Caches (building new cache in background and replacing atomically)  Used by some internal classes  Non-blocking algorithms

## Atomic Variables

The `java.util.concurrent.atomic` package defines classes that support atomic operations on single variables. All classes have **get() and set() methods** that work like reads and writes on volatile variables.

We have following Atomic classes

- AtomicInteger
- AtomicLong
- AtomicBoolean
- AtomicReference
- AtomicIntegerArray
- AtomicLongArray
- AtomicReferenceArray

### Common methods

- **incrementAndGet():** Atomically increments by one the current value.
- **decrementAndGet():** Atomically decrements by one the current value.
- **addAndGet(int delta):** Atomically adds the given value to the current value.
- **compareAndSet(int expect, int update):** Atomically sets the value to the given updated value if the current value == the expected value.
- **getAndAdd(int delta):** Atomically adds the given value to the current value.
- **set(int newValue):** Sets to the given value.

## Terms

### Liveness(live-less):

A liveness failure occurs when an activity gets into a state such that it is permanently unable to make forward progress. For example, if thread A is waiting for a resource that thread B holds exclusively, and B never releases it, A will wait forever.

### Race Conditions

A race condition occurs when the correctness of a computation depends on the relative timing or interleaving of multiple threads by the runtime; in other words, when getting the right answer relies on lucky timing.

## Threads Interview Questions

### StampedLock

Same as ReadWriteLock, but it returns a stamp represented by a **long** value. Each Stamp is a unique value, that stamp value compared at unlockWrote()

```
ExecutorService executor = Executors.newFixedThreadPool(2);
Map<String, String> map = new HashMap<>();
StampedLock lock = new StampedLock();

executor.submit(() -> {
    long stamp = lock.writeLock();
    try {
        sleep(1);
        map.put("foo", "bar");
    } finally {
        lock.unlockWrite(stamp);
    }
});
Runnable readTask = () -> {
    long stamp = lock.readLock();
    try {
        System.out.println(map.get("foo"));
        sleep(1);
    } finally {
        lock.unlockRead(stamp);
    }
};
executor.submit(readTask);
executor.submit(readTask);

stop(executor);
```

### What happens if we start same Thread(ob) Twice?

```
public class ThreadDemo extends Thread {
    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 guarantee volatile variable provides?

volatile provides the guarantee, changes made in one thread is visible to others.

### What is busy spin?

Busy spinning or busy wait in a multi-threaded environment is a technique where other threads loop continuously waiting for a thread to complete its task and signal them to start.

```
while (spinningFlag) {
    System.out.println("Waiting busy spinning");
}
```

### What is Thread Dump? How do you take thread dump in Java?

Process has multiple Threads. **Thread dump** is a summary of the state of all **threads** of the process

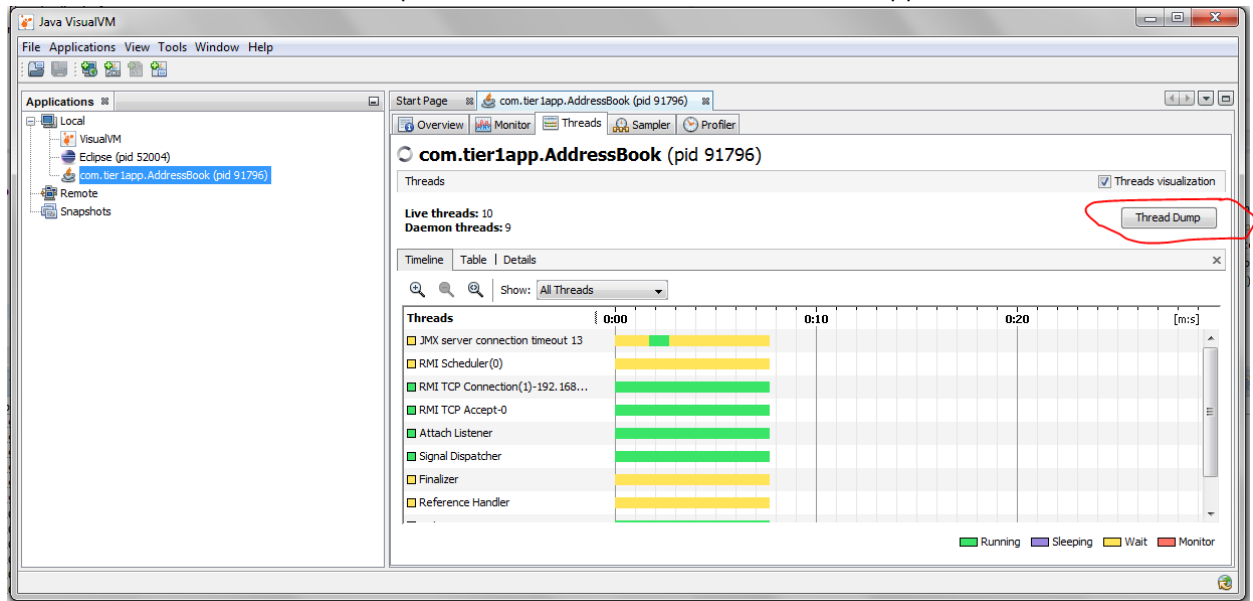
- **'jstack'** is an effective command line tool to capture thread dumps. It takes the *pid* of a process and displays the thread dump in the console. Alternatively, we can redirect its output to a file.

```
C:\Users\kavetis>jps
14464 BootLanguageServerBootApp
10964 XMLServerLauncher
1316 Eclipse
15788 Jps

C:\Users\kavetis>jstack 1316
2021-08-25 10:41:50
Full thread dump OpenJDK 64-Bit Server VM (15.0.2+7-27 mixed mode):

jstack 17264 > /tmp/threaddump.txt
```

- **Java VisualVM** is a GUI tool that provides detailed information about the applications



### Why Swing is not thread-safe in Java?

**User can't click two buttons at a time**, right? Since GUI screens are mostly updated in response of user action e.g. when user click a button, and since events are handled in the same Event dispatcher thread, it's easy to update GUI on that thread.

### What is a ThreadLocal variable in Java?

Thread-local variables are variables restricted to a thread, it's like thread's own copy which is not shared between multiple threads. Java provides a **ThreadLocal** class to support thread-local variables, It extends Object class.

- Basically, it is another way to achieve thread safety apart from writing immutable classes.
- Since Object is no more shared there is no requirement of Synchronization which can improve scalability and performance of application.
- ThreadLocal provides thread restriction which is extension of local variable. ThreadLocal are visible only in single thread. No two thread can see each other's thread local variable.
- These variables are generally **private static** field in classes and maintain its state inside thread.
- **void set(Object value), Object get(), void remove()** methods are available

```

public class ThreadLocalExample {
    public static class MyRunnable implements Runnable {
        private ThreadLocal<Integer> threadLocal = new ThreadLocal<Integer>();
        public void run() {
            threadLocal.set((int) (Math.random() * 1000));
            System.out.println(threadLocal.get());
        }
    }
    public static void main(String[] args) throws InterruptedException {
        MyRunnable sharedRunnableInstance = new MyRunnable();
        Thread thread1 = new Thread(sharedRunnableInstance);
        Thread thread2 = new Thread(sharedRunnableInstance);
        thread1.start();
        thread2.start();

        thread1.join(); // wait for thread 1 to terminate
        thread2.join(); // wait for thread 2 to terminate
    }
}

```

36  
16

- This example creates a single `MyRunnable` instance which is passed to two different threads.
- Both threads execute the `run()` method, and thus sets different values on the `ThreadLocal` instance.
- If the access to the `set()` call had been synchronized, and it had not been a `ThreadLocal` object, the second thread would have overridden the value set by the first thread

### Write code for thread-safe Singleton in Java?

When we say thread-safe, which means Singleton should remain singleton even if initialization occurs in the case of multiple threads.

```

public class DoubleCheckedLockingSingleton {
    private volatile DoubleCheckedLockingSingleton INSTANCE;

    private DoubleCheckedLockingSingleton() {
    }
    public DoubleCheckedLockingSingleton getInstance(){
        if(INSTANCE == null){
            synchronized(DoubleCheckedLockingSingleton.class){
                //double checking Singleton instance
                if(INSTANCE == null){
                    INSTANCE = new DoubleCheckedLockingSingleton();
                }
            }
        }
        return INSTANCE;
    }
}

```

### When to use Runnable vs Thread in Java? (Think Inheritance)

it's better to implement `Runnable` then extends `Thread` if you **also want to extend another class**

### Difference between Runnable and Callable in Java?

`Callable` was added on JDK 1.5. Main difference between these two is that `Callable`'s `call()` method can return value and throw `Exception`, which was not possible with `Runnable`'s `run()` method. `Callable` return `Future` object, which can hold the result of computation.

```

public class CallableDemo {
    public static void main(String[] args) throws Exception {
        ExecutorService service = Executors.newSingleThreadExecutor();
        SumTask sumTask = new SumTask(20);
        Future<Integer> future = service.submit(sumTask);
        Integer result = future.get();
        System.out.println(result);
    }
}

```



```

class SumTask implements Callable<Integer> {
    private int num = 0;
    public SumTask(int num){
        this.num = num;
    }
    public Integer call() throws Exception {
        int result = 0;
        for(int i=1;i<=num;i++){
            result+=i;
        }
        return result;
    }
}

```

### How to stop a thread in Java?

There were some control methods in JDK 1.0 e.g. **suspend()** and **resume()** which are deprecated. To manually stop, programmers either take advantage of volatile boolean variable and check in every iteration if run method has loops or **interrupt** threads to abruptly cancel tasks.

### Why wait, notify and notifyAll are not inside Thread class?

- Java provides lock at **object level** not at thread level. Every object has lock, which is acquired by thread.
- Now if thread needs to wait for certain lock, it make sense to call wait() on that object rather than on that thread. If wait() method declared on Thread class, it was not clear that for which lock thread was waiting.
- In short, since wait, notify and notifyAll operate at lock level, it makes sense to defined it on object class because lock belongs to object.

### What is the difference between livelock and deadlock in Java?

A real-world example of livelock occurs when two people meet in a narrow corridor, and each try to be polite by moving aside to let the other pass. but they end up staying same side without making any progress because they both repeatedly move the same way at the same time. In short, the main difference between livelock and deadlock is that in former state of process change but no progress is made.

### How do you check if a Thread holds a lock or not?

There is a method called **holdsLock()** on [java.lang.Thread](#), it returns true if and only if the current thread holds the monitor lock on the specified object.

```

Thread t = Thread.currentThread();
System.out.println(Thread.holdsLock(t));

```

```

public class ThreadDemo implements Runnable {
    public void run() {
        System.out.println("Currently executing thread is: " + Thread.currentThread().getName());
        System.out.println("Does thread holds lock? " + Thread.holdsLock(this));

        synchronized (this) {
            System.out.println("Does thread holds lock? " + Thread.holdsLock(this));
        }
    }
    public static void main(String[] args) {
        ThreadDemo g1 = new ThreadDemo();
        Thread t1 = new Thread(g1);
        t1.start();
    }
}

```

```

Currently executing thread is: Thread-0
Does thread hold lock? false
Does thread hold lock? true

```

### What is Semaphore in Java?

Semaphore in Java is a new kind of synchronizer. It is a counting semaphore. A semaphore maintains a set of permits. Each **acquire()** blocks if necessary, until a permit is available, and then takes it. Each **release()** adds a permit, potentially releasing a blocking acquirer. However, no actual permit objects are used; the Semaphore just keeps a count of the number available and acts accordingly. Semaphore is used to protect an expensive resource which is available in fixed number e.g. database connection in the pool.

### What is the difference between the submit() and execute() method thread pool in Java?

- **execute(Runnable command)** is defined in Executor interface and executes given task in future, but more importantly, it does not return anything.
- **submit()** is an overloaded method, it can take either **Runnable or Callable** task and can return Future object which can hold the pending result of computation. This method is defined on ExecutorService interface, which extends Executor interface, and every other thread pool class e.g. ThreadPoolExecutor or ScheduledThreadPoolExecutor gets these methods.

### Which method of Swing API are thread-safe in Java?

I know about **repaint()**, and **revalidate()** being thread-safe but there are other methods on different swing components e.g. **setText() method of JTextComponent, insert() and append() method of JTextArea class.**

### What is the difference between the volatile and atomic variable in Java?

For example **count++** operation will not become atomic just by declaring count variable as **volatile**. On the other hand **AtomicInteger** class provides atomic method to perform such compound operation atomically e.g. **getAndIncrement()** is atomic replacement of increment operator. It can be used to atomically increment current value by one. Similarly, you have atomic version for other data type and reference variable as well.

### What happens if a thread throws an Exception inside synchronized block?

To answer this question, no matter how you exist synchronized block, either normally by finishing execution or abruptly by throwing exception, **thread releases the lock it acquired while entering that synchronized block.**

### How do you ensure that N thread can access N resources without deadlock?

Key point here is order, if you acquire resources in a particular order and release resources in reverse order you can prevent deadlock.

### What is busy spin, and why should you use it?

Busy spinning is a waiting strategy in which one thread loop continuously to check certain condition and waiting for other thread to change this condition to break the loop without releasing CPU so that waiting thread can proceed its work further

### What's the difference between Callable and Runnable?

Both of these are interfaces used to carry out task to be executed by a thread. The main difference between the two interfaces is that

- Callable can **return a value**, while Runnable cannot.
- Callable can throw a checked exception, while Runnable cannot.
- Runnable has been around since Java 1.0, while Callable was introduced as part of Java 1.5.

*Callable* is an interface, contains a single *call()* method – which returns a generic value V:

```
public interface Callable<V> {
    V call() throws Exception;
}

class CallableExample implements Callable
{
    public Object call() throws Exception
    {
        Random generator = new Random();
        Integer randomNumber = generator.nextInt(5);
        Thread.sleep(randomNumber * 1000);

        return randomNumber;
    }
}
```

## Object level and Class level locks in Java

In Single line

- Object level lock means **Instance Synchronized method**
- Class level lock means **Static Synchronized method**

**Object level lock** : Every object in java has a unique lock. Whenever we are using synchronized keyword, then only lock concept will come in the picture. If a thread wants to execute **synchronized method** on the given object. First, it has to get lock of that object.

Once thread got the lock then it is allowed to execute any synchronized method on that object. Once method execution completes automatically thread releases the lock. Acquiring and release lock internally is taken care by JVM and programmer is not responsible for these activities. Let's have a look on the below program to understand the object level lock:

```
class Geek implements Runnable {
    public void run() {
        Lock();
    }

    public void Lock() {
        System.out.println(Thread.currentThread().getName());

        synchronized (Geek.class) {
            System.out.println("in block " + Thread.currentThread().getName());
            System.out.println("in block " + Thread.currentThread().getName() + " end");
        }
    }

    public static void main(String[] args) {
        Geek g1 = new Geek();
        Thread t1 = new Thread(g1);
        Thread t2 = new Thread(g1);

        Geek g2 = new Geek();
        Thread t3 = new Thread(g2);

        t1.setName("t1");
        t2.setName("t2");
        t3.setName("t3");

        t1.start();
        t2.start();
        t3.start();
    }
}
```

```

t3
t2
t1
in block t3
in block t3 end
in block t1
in block t1 end
in block t2
in block t2 end

```

In above Threads

- If t1 -gets lock, t2 won't execute – because object lock is there, but t3 can execute – because it is a separate object. So t1, t3 are executed in parallel, but t2 waits until t1 completed its execution.

**Class level lock:** Every class in java has a unique lock which is nothing but class level lock. If a thread wants to execute a **static synchronized method**, then thread requires class level lock. Once a thread got the class level lock, then it is allowed to execute any static synchronized method of that class. Once method execution completes automatically thread releases the lock. Lets look on the below program for better understanding: just changed lock method to static.

```

public static void Lock() {
    System.out.println(Thread.currentThread().getName());

    synchronized (Geek.class) {
        System.out.println("in block " + Thread.currentThread().getName());
        System.out.println("in block " + Thread.currentThread().getName() + " end");
    }
}

```

```

t1
t3
t2
in block t1
in block t1 end
in block t2
in block t2 end
in block t3
in block t3 end

```

In above, If t1 -gets lock, t2, t3 won't execute – because Class lock is there. So above output is ordered.

### Producer-Consumer solution using threads in Java

- The producer's job is to generate data, put it into the buffer, and start again.
- same time, the consumer is consuming the data (i.e. removing it from the buffer), one piece at a time.
- producer won't try to add data into the buffer if it's full & consumer won't try to remove data from an empty buffer

```

class Producer extends Thread {
    List buffer;
    int maxsize;

    public Producer(List buffer, int maxsize) {
        this.buffer = buffer;
        this.maxsize = maxsize;
    }
}

```

```

@Override
public void run() {
    int i = 1;
    while (true) {
        synchronized (buffer) {
            try {
                if (buffer.size() == maxsize) {
                    System.out.println("Maximum Size Reached, wait until consume");
                    buffer.wait();
                } else {
                    buffer.add(i++);
                    System.out.println(i + " : Produced, notify wating COnsumer Thread");
                    buffer.notifyAll();
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

class Consumer extends Thread {
    List buffer;
    int maxsize;

    public Consumer(List buffer, int maxsize) {
        this.buffer = buffer;
        this.maxsize = maxsize;
    }

    @Override
    public void run() {
        while (true) {
            try {
                synchronized (buffer) {
                    if (buffer.isEmpty()) {
                        System.out.println("Consumer : Buffer Empty, wait untill produce");
                        buffer.wait();
                    } else {
                        Object ob = buffer.remove(0);
                        System.out.println(ob + " : Removed, notify Producer waiting for Removing for maxsize");
                        buffer.notifyAll();
                    }
                }
            } catch (Exception e) {
                // TODO: handle exception
            }
        }
    }
}

public class ProducerConsumer {

    public static void main(String[] args) {
        List buffer = new LinkedList<>();
        Producer producer = new Producer(buffer, 10);
        Consumer consumer = new Consumer(buffer, 10);
        producer.start();
        consumer.start();
    }
}
28054 : Produced, notify wating COnsumer Thread
28055 : Produced, notify wating COnsumer Thread
28056 : Produced, notify wating COnsumer Thread
28057 : Produced, notify wating COnsumer Thread
28058 : Produced, notify wating COnsumer Thread
28059 : Produced, notify wating COnsumer Thread
28060 : Produced, notify wating COnsumer Thread

```

```
Maximum Size Reached, wait until consume
28050 : Removed, notify Producer waiting for Removing for maxsize
28051 : Removed, notify Producer waiting for Removing for maxsize
28052 : Removed, notify Producer waiting for Removing for maxsize
28053 : Removed, notify Producer waiting for Removing for maxsize
28054 : Removed, notify Producer waiting for Removing for maxsize
28055 : Removed, notify Producer waiting for Removing for maxsize
28056 : Removed, notify Producer waiting for Removing for maxsize
28057 : Removed, notify Producer waiting for Removing for maxsize
28058 : Removed, notify Producer waiting for Removing for maxsize
28059 : Removed, notify Producer waiting for Removing for maxsize
Consumer : Buffer Empty, wait untill produce
```

### Thread. yield ()

**yield() method:** Theoretically, to 'yield' means to let go, to give up, to surrender. A yielding thread tells the virtual machine that it's willing to let other threads be scheduled in its place.

This indicates that it's not doing something too critical. Note that *it's only a hint*, though, and not guaranteed to have any effect at all.

- Yield is a Static method and Native too.
- Yield tells the currently executing thread to give a chance to the threads that have equal priority in the Thread Pool.
- There is no guarantee that Yield will make the currently executing thread to runnable state immediately.
- **It can only make a thread from Running State to Runnable State, not in wait or blocked state.**

### What do you understand about Thread Priority?

Every thread has a priority, usually higher priority thread gets precedence in execution, but it depends on Thread Scheduler implementation that is OS dependent. We can specify the priority of thread, but it doesn't guarantee that higher priority thread will get executed before lower priority thread.

### How can we make sure main() is the last thread to finish in Java Program?

We can use Thread `join()` method to make sure all the threads created by the program is dead before finishing the main function.

### Why wait(), notify() and notifyAll() methods have to be called from synchronized method or block?

- When a Thread calls `wait()` on any Object, it must have the monitor on the Object that it will leave and goes in wait state until any other thread call `notify()` on this Object.
- Similarly when a thread calls `notify()` on any Object, it leaves the monitor on the Object and other waiting threads can get the monitor on the Object.
- Since all these methods require Thread to have the Object monitor, that can be achieved only by `synchronization`, they need to be called from synchronized method or block.

### How can we achieve thread safety in Java?

There are several ways to achieve thread safety in java – **synchronization, atomic concurrent classes, implementing concurrent Lock interface, using volatile keyword**, using immutable classes and Thread safe classes.

### What is volatile keyword in Java

When we use volatile keyword with a variable, all the threads read it's value directly from the main memory and don't read from cache. This makes sure that the value read is the same as in the memory.

### What is ThreadLocal?

Java ThreadLocal is used to create thread-local variables. We know that all threads of an Object share it's variables, so if the variable is not thread safe, we can use synchronization but if we want to avoid synchronization, we can use ThreadLocal variables.

Every thread has it's own ThreadLocal variable and they can use it's get() and set() methods to get the default value or change it's value local to Thread. ThreadLocal instances are typically private static fields in classes that wish to associate state with a thread

### What is BlockingQueue? implement Producer-Consumer using Blocking Queue?

- `java.util.concurrent.BlockingQueue` is a Queue that supports operations that wait for the queue to become non-empty when retrieving and removing an element, and wait for space to become available in the queue when adding an element.
- BlockingQueue doesn't accept `null` values and throw `NullPointerException` if you try to store null value in the queue.
- BlockingQueue implementations are thread-safe. All queuing methods are atomic in nature and use internal locks or other forms of concurrency control.
- BlockingQueue interface is part of [java collections framework](#) and it's primarily used for implementing producer consumer problem. Check this post for [BlockingQueue](#).

### What is Executors Class?

Executors class provide utility methods for `Executor`, `ExecutorService`, `ScheduledExecutorService`, `ThreadFactory`, and `Callable` classes.

Executors class can be used to easily create Thread Pool in java, also this is the only class supporting execution of Callable implementations.

### What happens when an Exception occurs in a thread?

**Thread.UncaughtExceptionHandler** is an interface, defined as nested interface for handlers invoked when a Thread abruptly terminates due to an uncaught exception.

When a thread is about to terminate due to an uncaught exception the Java Virtual Machine will query the thread for its UncaughtExceptionHandler using `Thread.getUncaughtExceptionHandler()` and will invoke the handler's `uncaughtException()` method, passing the thread and the exception as arguments.

### Why wait, notify and notifyAll are not inside thread class?

One reason which is obvious is that Java provides lock at object level not at thread level.

### How do you check if a Thread holds a lock or not?

Boolean `Thread.holdsLock(Obj)`

# 11. Collections Framework

## Traditional Data Structures

Data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently.

Depending on the organization of the elements, data structures are classified into two types:

- **Linear data structures:** Elements are accessed in a sequential order, but it is not compulsory to store all elements sequentially. **Examples: Linked Lists, Stacks and Queues.**
- **Non - linear data structures:** Elements of this data structure are stored/accessed in a non-linear order. **Examples: Trees and graphs.**

### Analysis of Algorithms?

To go from city "A" to city "B", there can be many ways of accomplishing this: by flight, by bus, by train and by bicycle. Depending on the availability and convenience, we choose the one that suits us. Similarly, in computer science, multiple algorithms are available for solving the same problem (for example, a sorting problem has many algorithms, like insertion sort, selection sort, quick sort and many more).

Algorithm analysis helps us to determine which algorithm is most efficient in terms of time and space consumed.

#### **Worst case**

- Defines the input for which the algorithm takes a **long time** (slowest time to complete).
- Input is the one for which the algorithm runs the slowest.

#### **Best case**

- Defines the input for which the algorithm takes the **least time** (fastest time to complete).
- Input is the one for which the algorithm runs the fastest.

#### **Average case**

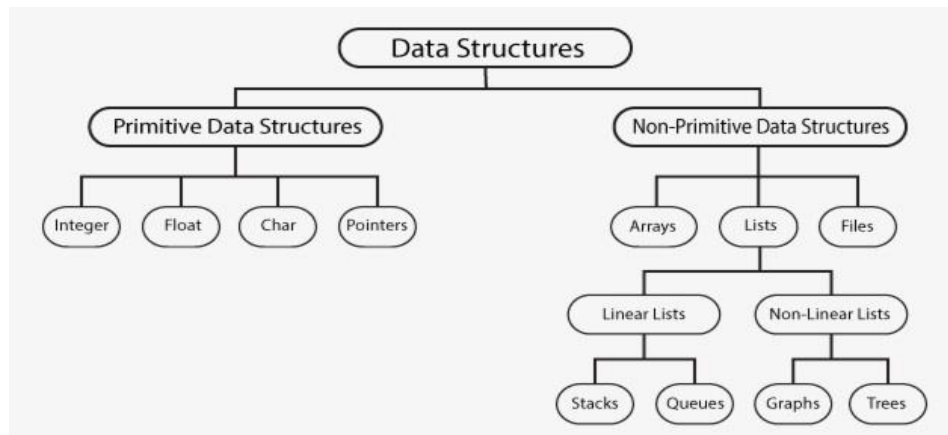
- Provides a prediction about the running time of the algorithm.
- Run the algorithm many times, using many different inputs take the **average** of them.

For analysis (best case, worst case, and average), we try to give the **upper bound (O)** and **lower bound ( $\Omega$ )** and **average running time ( $\Theta$ )**.

## Types of Data Structures

- Primitive data structures
- Non-primitive data structures





### Primitive Data Structures

- Primitive Data Structures are the basic data structures that directly operate upon the machine instructions.
- **Integers, Floating, Character, String constants** and **Pointers** come under this category.

### Non-primitive Data Structures

- Non-primitive data structures are more complicated data structures and are derived from primitive data structures.
- They emphasize on grouping same or different data items with relationship between each data item. **Arrays, Lists** and **Files** come under this category.

### Data Structures

#### Linked Lists

- **Linked List**
- **Doubly Linked List**
- **Circular Linked List**

#### Stack & Queue

- **Stack**
- **Queue**

#### Tree Data Structure

- **Tree Data Structure**
- **Tree Traversal**
- **Binary Search Tree**
- **AVL Tree**
- **Spanning Tree**
- **Heap**

#### Graph Data Structure

- **Graph Data Structure**
- **Depth First Traversal**
- **Breadth First Traversal**

### Arrays

An array is a sequential collection of elements of same data type and stores **data elements in a continuous memory location**. The elements of an array are accessed by using an *index*. The *index* of an array of size N can range from 0 to N-1.

<i>arr</i>	4	12	7	15	9
<i>index</i>	0	1	2	3	4

### Basic Operations

Following are the basic operations supported by an array.

- **Insertion** – Adds an element at the given index.
- **Deletion** – Deletes an element at the given index.
- **Search** – Searches an element using the given index or by the value.
- **Update** – Updates an element at the given index.
- **Traverse** – print all the array elements one by one.

### 1. Linked List

Like arrays, Linked List is a linear data structure. Unlike arrays, **linked list elements are not Stores in a continuous memory location**; the elements are linked using pointers



**Node:** A node is a collection of two sub-elements or parts. A **data** part that stores the element and a **next** part that stores the link to the next node.



### Linked List VS Arrays

Arrays can be used to store linear data of similar types, but arrays have following limitations.

- **The size of the arrays is fixed:** So, we must know the upper limit on the number of elements in advance
- Inserting a new element in an array of elements is expensive. Because room has to be created for the new elements and to create room existing elements have to shifted.

### Types of Linked List

Following are the various types of linked list.

- **Simple Linked List** – Item navigation is forward only.
- **Doubly Linked List** – Items can be navigated forward and backward.
- **Circular Linked List** – Last item contains link of the first element as next and the first element has a link to the last element as previous.

## Simple Linked List

### Basic Operations

Following are the basic operations supported by a list.

- **Insertion** – Adds an element at the beginning of the list.
- **Deletion** – Deletes an element at the beginning of the list.
- **Display** – Displays the complete list.
- **Search** – Searches an element using the given key.
- **Delete** – Deletes an element using the given key.

### Algorithm

1. A linked list is represented by a pointer to the first node of the linked list. The first node is called **head**. If the linked list is empty, then value of head is **NULL**.

2. Each node in a list consists of at least two parts:

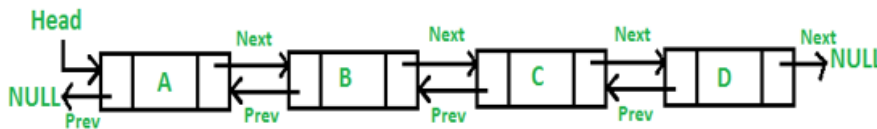
- **Data**
- **Pointer** (Or Reference) to the next node

3. In Java, LinkedList can be represented as a class and **a Node as a separate class**. The LinkedList class contains a reference of Node class type.

```
class LinkedList
{
    Node head; // head of list
    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;
        // Constructor to create a new node Next is by default initialized as null
        Node(int d) {
            data = d;
        }
    }
}
```

## Doubly Linked List

A **Doubly Linked List** (DLL) contains an extra pointer, typically called *previous pointer*, together with next pointer and data which are there in singly linked list.



### Basic Operations

Following are the basic operations supported by a list.

- **Insertion** – Adds an element at the beginning of the list.
- **Deletion** – Deletes an element at the beginning of the list.
- **Insert Last** – Adds an element at the end of the list.
- **Delete Last** – Deletes an element from the end of the list.
- **Insert After** – Adds an element after an item of the list.
- **Delete** – Deletes an element from the list using the key.
- **Display forward** – Displays the complete list in a forward manner.

- **Display backward** – Displays the complete list in a backward manner.

```

public class DLL {
    Node head; // head of list

    /* Doubly Linked list Node*/
    class Node {
        int data;
        Node prev;
        Node next;

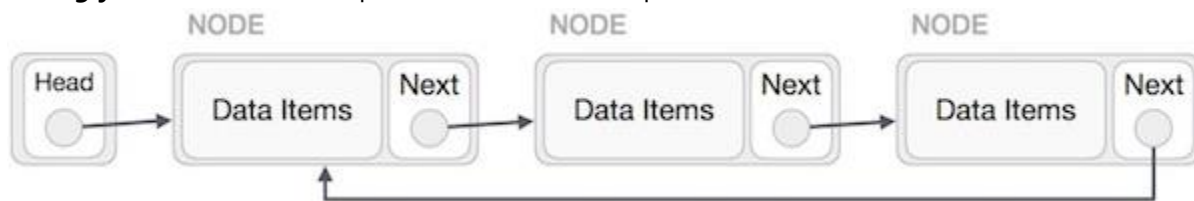
        // Constructor to create a new node next and prev is by default initialized as null
        Node(int d) {
            data = d;
        }
    }
}

```

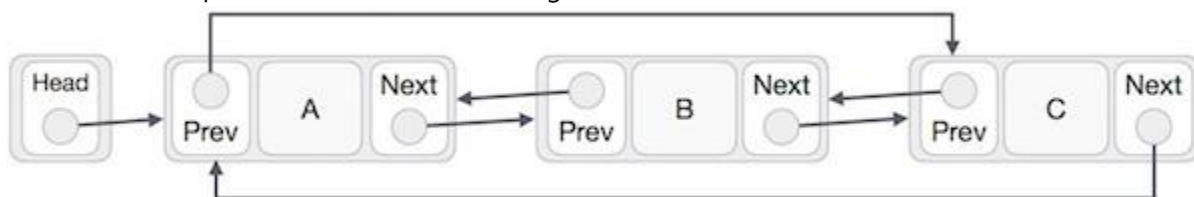
## Circular Linked List

Circular Linked List is a variation of Linked list in which the first element points to the last element and the last element points to the first element. Both Singly Linked List and Doubly Linked List can be made into a circular linked list.

**In singly linked list**, the next pointer of the last node points to the first node.



**In doubly linked list**, the next pointer of the last node points to the first node and the previous pointer of the first node points to the last node making the circular in both directions.



## 2.Stack

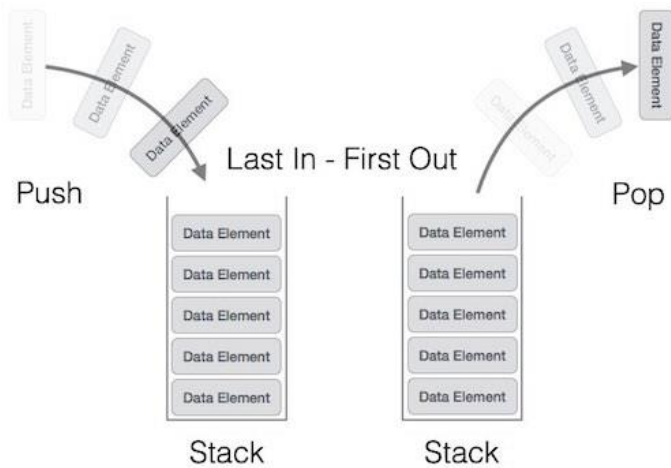
A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.



### **Basic Operations**

- **push()** – Adding (pushing) an element on the stack.
- **pop()** – Removing (accessing) an element from the stack.
- **peek()** – get the top data element of the stack, **without removing it**.

- **isFull()** – check if stack is full.
- **isEmpty()** – check if stack is empty.



#### Push operation involves a series of steps –

- **Step 1** – Checks if the stack is full.
- **Step 2** – If the stack is full, produces an error and exit.
- **Step 3** – If the stack is not full, increments **top** to point next empty space.
- **Step 4** – Adds data element to the stack location, where top is pointing.
- **Step 5** – Returns success.

### 3.Queue

Queue is open at both its ends. **One end is always used to insert data** (enqueue) and the **other is used to remove data (dequeue)**. Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.



#### Basic Operations

- **enqueue()** – add (store) an item to the queue.
- **dequeue()** – remove (access) an item from the queue.
- **peek()** – Gets the element at the front of the queue **without removing it**.
- **isFull()** – Checks if the queue is full.
- **isEmpty()** – Checks if the queue is empty.

#### Steps should be taken to enqueue (insert) data into a queue –

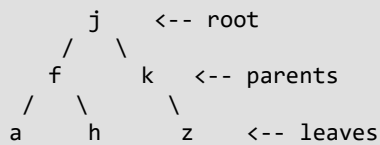
- **Step 1** – Check if the queue is full.
- **Step 2** – If the queue is full, produce overflow error and exit.
- **Step 3** – If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** – Add data element to the queue location, where the rear is pointing.
- **Step 5** – return success.

## 4. Trees

Unlike Arrays, Linked Lists, Stack, and queues, which are linear data structures, trees are hierarchical data structures.

### Tree Vocabulary:

- The topmost node is called **root** of the tree.
- The elements that are directly under an element are called its **children**.
- The element directly above something is called its **parent**.

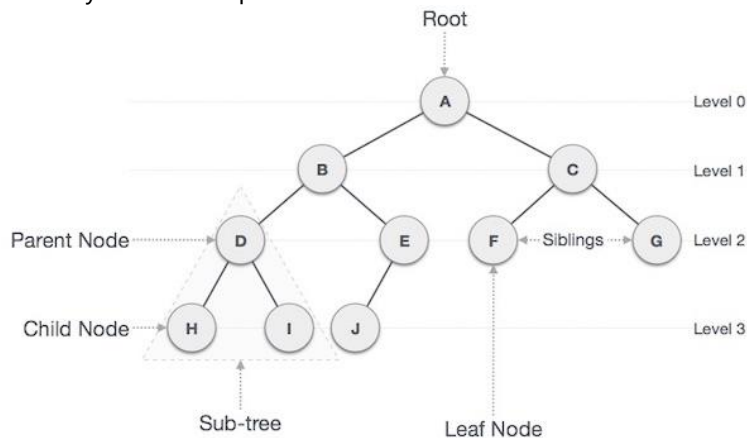


### Why Trees?

1. One reason to use trees might be because you want to store information that naturally forms a hierarchy. **For example, the file system on a computer.**
2. Trees (with some ordering e.g., BST) provide moderate access/search (quicker than Linked List and slower than arrays).
3. Trees provide moderate insertion/deletion (quicker than Arrays and slower than Linked Lists).
4. Like Linked Lists and unlike Arrays, Trees don't have an upper limit on number of nodes as nodes are linked using pointers.

## Binary Tree

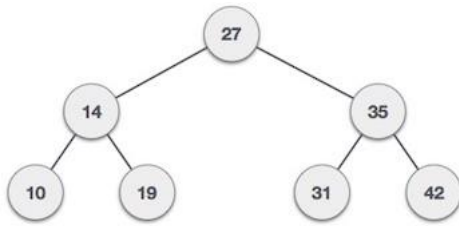
A binary tree has a special condition that **each node can have a maximum of two children**.



## Binary Search Tree

Binary Search tree exhibits a special behavior.

- The node's **left child must less than its parent's value**
- The node's **right child must greater than its parent value.**



### BST Basic Operations

The basic operations that can be performed on a binary search tree data structure, are the following

- **Insert** – Inserts an element in a tree/create a tree.
- **Search** – Searches an element in a tree.
- **Inorder Traversal** – left subtree first, then the root and later the right sub-tree.
- **Preorder Traversal** – root node is first, then the left subtree and later the right sub-tree
- **Postorder Traversal** – left subtree first, then the right subtree and later root node

### Algorithm

```

If root is NULL
  then create root node
  return

If root exists then
  compare the data with node.data

  while until insertion position is located

    If data is greater than node.data
      goto right subtree
    else
      goto left subtree

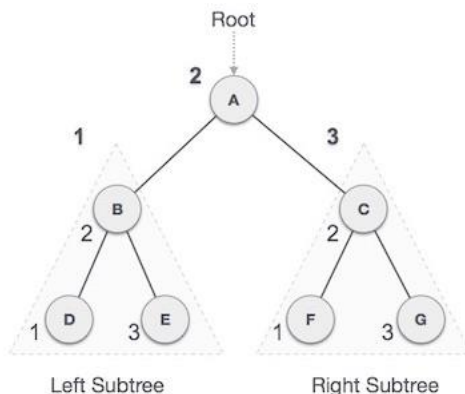
  endwhile

  insert data

end If
  
```

### Tree Traversal

**In-order Traversal** - left subtree is visited first, then the root and later the right sub-tree.



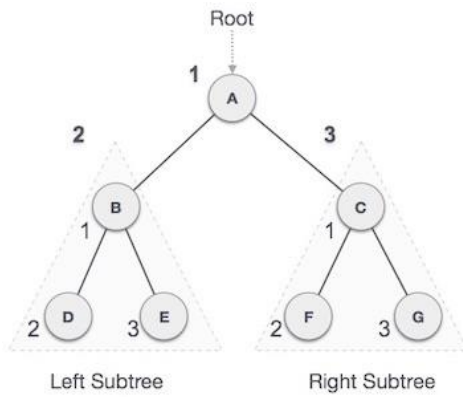
We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be **-D → B → E → A → F → C → G**

### Algorithm

Until all nodes are traversed -  
**Step 1** - Recursively traverse left subtree.  
**Step 2** - Visit root node.  
**Step 3** - Recursively traverse right subtree.

### Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.

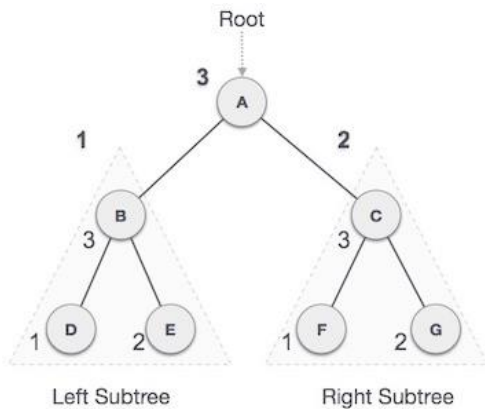


We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be **-A → B → D → E → C → F → G**

Until all nodes are traversed -  
**Step 1** - Visit root node.  
**Step 2** - Recursively traverse left subtree.  
**Step 3** - Recursively traverse right subtree.

### Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First, we traverse the left subtree, then the right subtree and finally the root node.



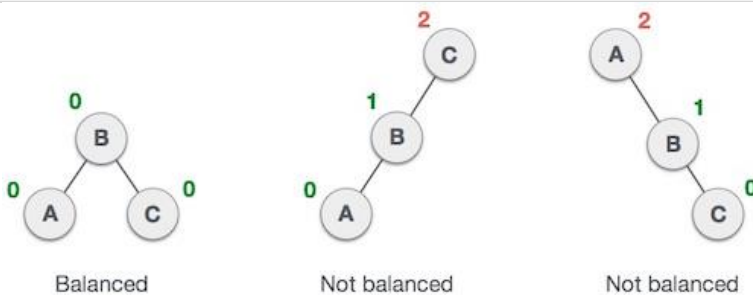
We start from **A**, and following Post-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be **-D → E → B → F → G → C → A**

Until all nodes are traversed -  
**Step 1** - Recursively traverse left subtree.  
**Step 2** - Recursively traverse right subtree.  
**Step 3** - Visit root node.

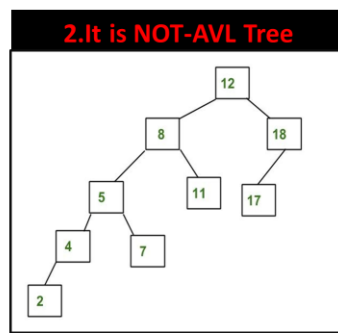
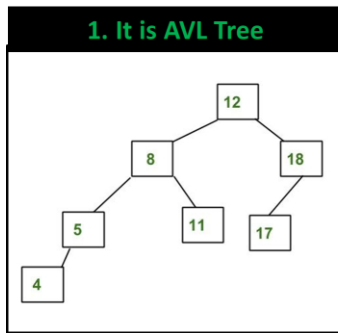
### AVL Trees

AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

$$\text{BalanceFactor} = (\text{height of left subtree}) - (\text{height of right subtree})$$







**1.Tree-1** is AVL because differences between heights of left and right subtrees for **every node is less than or equal to 1.**

**2.Tree-2** is not AVL because differences between heights of left and right subtrees for 8 and 18 is greater than 1

## Sorting Algorithms

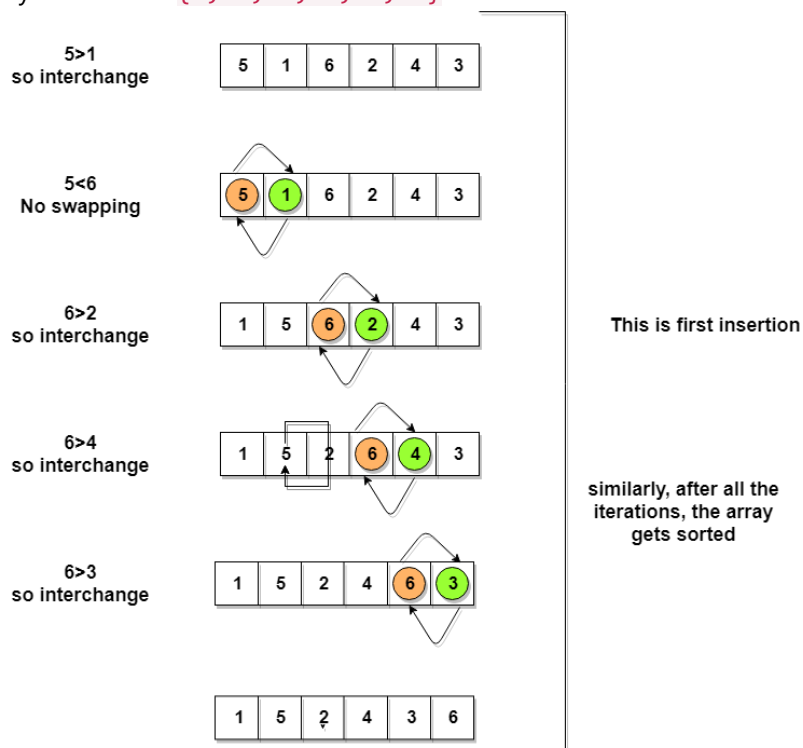
Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order.

### Bubble Sort Algorithm

This sorting algorithm is comparison-based algorithm in **which each pair of adjacent elements is compared, and the elements are swapped if they are not in order.**

This algorithm is not suitable for large data sets as its average and worst-case complexity are of  $O(n^2)$  where **n** is the number of items.

Let's consider an array with values **{5, 1, 6, 2, 4, 3}**



So, as we can see in the representation above, after the first iteration, 6 is placed at the last index, which is the correct position for it. Similarly, after the second iteration, 5 will be at the second last index, and so on.

```
begin BubbleSort(list)
  for all elements of list
    if list[i] > list[i+1]
      swap(list[i], list[i+1])
    end if
  end for

  return list
end BubbleSort
```

### Insertion Sort – Pick & Insert in Correct order using Sorted Sub list

We take an unsorted array for our example.



Insertion sort compares the first two elements.



It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.



Insertion sort moves ahead and compares 33 with 27.



And finds that 33 is not in the correct position.



It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.



By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.



These values are not in a sorted order.



So, we swap them.



However, swapping makes 27 and 10 unsorted.



Hence, we swap them too.



Again, we find 14 and 10 in an unsorted order.



We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.



This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

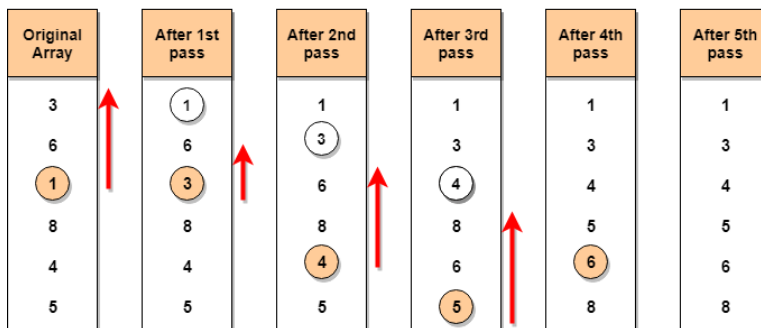
### Algorithm

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

- Step 1 - If it is the first element, it is already sorted. return 1;
- Step 2 - Pick next element
- Step 3 - Compare with all elements in the sorted sub-list
- Step 4 - Shift all the elements in the sorted sub-list that is greater than the value to be sorted
- Step 5 - Insert the value
- Step 6 - Repeat until list is sorted

### Selection Sort

Selection sort is conceptually the simplest sorting algorithm. This algorithm will first find the **smallest** element in the array and swap it with the element in the **first** position, then it will find the **second smallest** element and swap it with the element in the **second** position, and it will keep on doing this until the entire array is sorted.

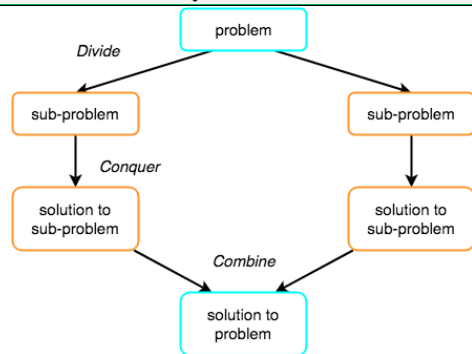


- Step 1 - Set MIN to location 0
- Step 2 - Search the minimum element in the list
- Step 3 - Swap with value at location MIN
- Step 4 - Increment MIN to point to next element
- Step 5 - Repeat until list is sorted

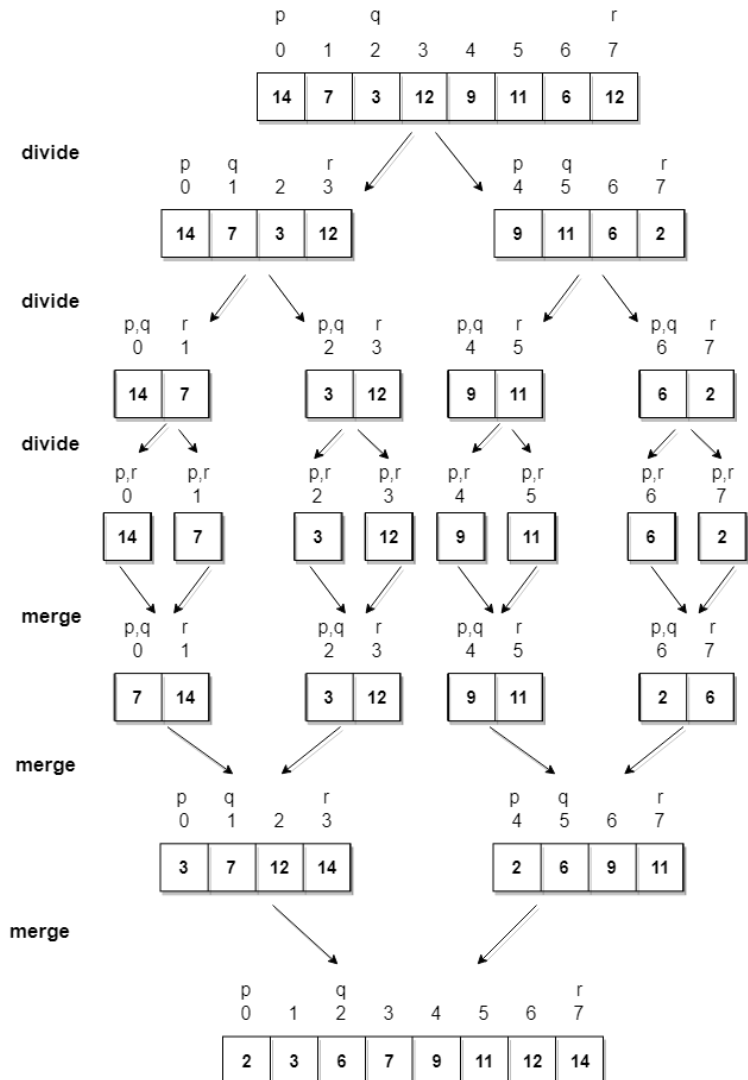
### Merge Sort Algorithm

Merge Sort follows the rule of **Divide and Conquer** to sort a given set of numbers/elements, recursively, hence consuming less time.

#### Divide and Conquer



If we can break a single big problem into smaller sub-problems, solve the smaller sub-problems and combine their solutions to find the solution for the original big problem, it becomes easier to solve the whole problem.



## Algorithm

Merge sort keeps on dividing the list into equal halves until it can no more be divided. If it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

**Step 1** - if it is only one element in the list it is already sorted, return.  
**Step 2** - divide the list recursively into two halves until it can no more be divided.  
**Step 3** - merge the smaller lists into new list in sorted order.

## Quick Sort

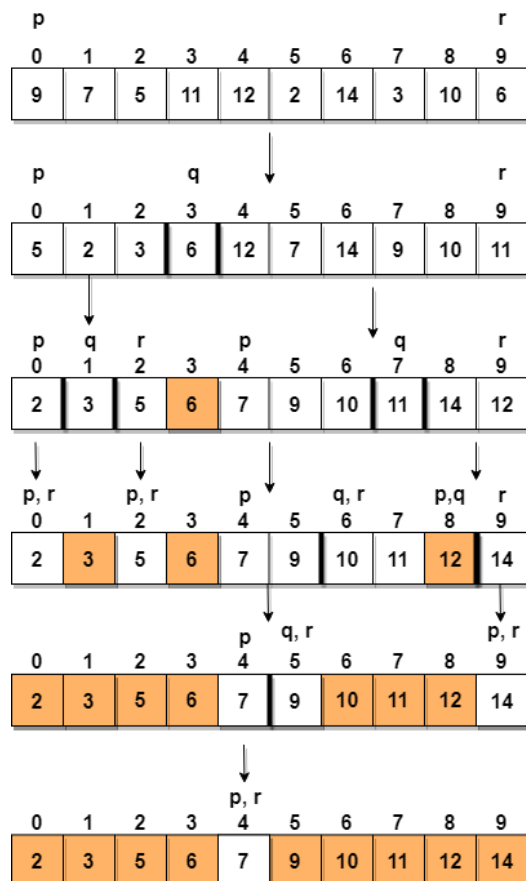
Quick sort is based on the divide-and-conquer approach based on the idea of **choosing one element as a pivot element (normally height index value)** and partitioning the array around it such that:

- Left side of pivot contains all the elements that are less than the pivot element
- Right side contains all elements greater than the pivot

**For example:** In the array {52, 37, 63, 14, 17, 8, 6, 25}, we take 25 as **pivot**. So, after the first pass, the list will be changed like this.

{6 8 17 14 25 63 37 52}

Hence after the first pass, pivot will be set at its position, with all the elements **smaller** to it on its left and all the elements **larger** than to its right. Now 6 8 17 14 and 63 37 52 are considered as two separate subarrays, and same recursive logic will be applied on them, and we will keep doing this until the complete array is sorted.



**Step 1** - Choose the highest index value has pivot

**Step 2** – Take two variables to point left and right of the list excluding pivot  
**Step 3** – left points to the low index  
**Step 4** – right points to the high  
**Step 5** – while value at left is less than pivot move right  
**Step 6** – while value at right is greater than pivot move left  
**Step 7** – if both step 5 and step 6 does not match swap left and right  
**Step 8** – if  $\text{left} \geq \text{right}$ , the point where they met is new pivot

## Searching Algorithms

Well, to search an element in a given array, there are two popular algorithms available:

1. Linear Search
2. Binary Search

### Linear Search

Linear search is a very basic and simple search algorithm. In Linear search, we search an element or value in a given array by traversing the array from the starting, till the desired element or value is found.

For example, consider the following image:

Arr    0   1   2   3   4   5   6   7   8   9

If you want to determine the positions of the **occurrence of the number 7** in this array, we need to compare every element in the array from start to end, i.e., from index 1 to index 10 will be compared with number 7, to check which element matches the number 7.

Linear Search (Array A, Value x)

Step 1: Set i to 1  
Step 2: if  $i > n$  then go to step 7  
Step 3: if  $A[i] = x$  then go to step 6  
Step 4: Set i to  $i + 1$   
Step 5: Go to Step 2  
Step 6: Print Element x Found at index i and go to step 8  
Step 7: Print element not found  
Step 8: Exit

### Binary Search Algorithm

Binary Search is applied on the sorted array or list of large size. Its time complexity of  **$O(\log n)$**  makes it very fast as compared to other sorting algorithms. The only limitation is that the array or list of elements must be sorted for the binary search algorithm to work on it. Binary search works only on a sorted set of elements. To use binary search on a collection, the collection must first be **sorted**.

When binary search is used to perform operations on a sorted set, the number of iterations can always be reduced based on the value that is being searched. Let us consider the following array:

Arr    0   1   2   3   4   5   6   7   8   9

By using **linear search**, the position of element 8 will be determined in the 9th iteration.



**Iterator interface:** provides the facility of iterating the elements in **forward direction only**.

<code>public boolean hasNext()</code>	It returns true if iterator has more elements.
<code>public Object next()</code>	It returns the element and moves the cursor pointer to the next element.
<code>public void remove()</code>	It removes the last elements returned by the iterator. It is rarely used.

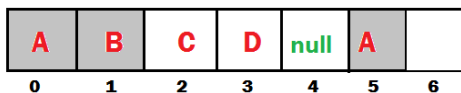
**Collection:** Root interface with basic methods like `add()`, `remove()`, `contains()`, `isEmpty()`, etc.

Method	Description
<code>boolean add(Object o)</code>	insert an element into collection
<code>boolean addAll(Collection c)</code>	insert a collection into collection
<code>void clear()</code>	remove all elements from collection
<code>boolean remove(Object o)</code>	delete an element from collection
<code>boolean removeAll(Collection c)</code>	delete all from specified collection
<code>boolean retainAll(Collection c)</code>	remove all elements except collection c
<code>boolean contains(Object o)</code>	used to search an element
<code>boolean containsAll(Collection c)</code>	used to search an collection
<code>boolean equals(Object o)</code>	used to check quality of object
<code>boolean isEmpty()</code>	check the collection is empty or not
<code>int size()</code>	get number of element in a collection
<code>int hashCode()</code>	return the hashCode number for collection
<code>Iterator iterator()</code>	return an iterator
<code>Object[] toArray()</code>	convert the collection to array

- If you see above **only add, remove methods are there. get() is not there.**
- **get()** methods are implemented based on underlying data Structure .
- **add(Object) method is Object based, not index.** Because these are generalized methods which will apply for all collection classes

## List

1. List is child interface of collection
2. 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
3. **Index** will play very important role in List
4. We can **preserve insertion order via index & differentiate duplicate objects using index**



Add / Remove	Find	Special
<code>Boolean add(int index, Object o)</code> <code>Boolean addAll(int index, Collection c)</code> <code>Boolean remove(int index)</code>	<code>Object get(int index)</code> <code>Object set(int index, Object o)</code> <code>int indexOf(Object o)</code> <code>int lastIndexOf(Object o)</code>	<code>ListIterator listIterator()</code>

### ArrayList – Internal implementation

- 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, Cloneable & RandomAccess**
- **Except TreeSet & TreeMap Everywhere Heterogeneous Objects are allowed**

#### • Example

```
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]
    }
}
```

#### **Internal implementation: (Ref)**

1. **ArrayList** grows dynamically as the elements are added to it. Internally an ArrayList uses an **Object[] Array**.

```
private transient Object[] elementData;
```

2. When an object of ArrayList is created without initial capacity, the default constructor of the ArrayList class is invoked. default capacity of 10 is assigned at a time of empty initialization of ArrayList.

```
public ArrayList() {
    this(10);
}
```

3. In the **add(Object)**, the capacity of the ArrayList will be checked before adding a new element

```
public boolean add(E e) {
    ensureCapacity(size + 1); // Increments modCount!!
    elementData[size++] = e; // elementdata[11] = 100; - adding new element to next index
    return true;
}

public void ensureCapacity(int minCapacity) {
    if (minCapacity > oldCapacity) {
        int newCapacity = (oldCapacity * 3) / 2 + 1;
        elementData = Arrays.copyOf(elementData, newCapacity);
    }
}
```

4. If size of the filled elements is greater than the maximum size of the array, then it will increase the size of array by using below formulae, Then elements will copy to old Array to New Array

Till Java 6 :  $\text{int newCapacity} = (\text{oldCapacity} * 3) / 2 + 1;$

From Java 7 :  $\text{int newCapacity} = \text{oldCapacity} + (\text{oldCapacity} \gg 1);$  (50% of old)

5. If elements are adding in the middle index, **Array Elements will be shifted to Right**

- Adding → NOT good →  $O(n)$
- get(by index) → Good →  $O(1)$

**Interviewer: What is the runtime performance of the get() method in ArrayList, where n represents the number of elements ?**

get(), set(), size() operations run in constant time i.e  $O(1)$

add() operation runs in amortized constant time, i.e adding n elements require  $O(n)$  time.

```
public class ArrayListDemo {
public static void main(String[] args) {
    /*1.ArrayList With Defalut Capacity 10 is created
    * & assiged values as null

    * elemetntdata = [null, null, null,...10 Objects]
    * modcount = 0, no modifications performed
    * size=0
    * */
    ArrayList<String> list = new ArrayList<>();

    /*adding first element, add("one");
    * elementdata=["one,null, null, null ...."]
    * modcount = 1
    * Size = 1
    * */
    list.add("One");
    list.add("Two");
    list.add("Three");
    System.out.println(list);
}
}
```

Name	Value
args	String[0] (id=16)
list	ArrayList<E> (id=18)
elementData	Object[10] (id=31)
[0]	"One" (id=32)
[1]	"Two" (id=33)
[2]	"Three" (id=36)
[3]	null
[4]	null
[5]	null
[6]	null
[7]	null
[8]	null
[9]	null
modCount	3
size	3

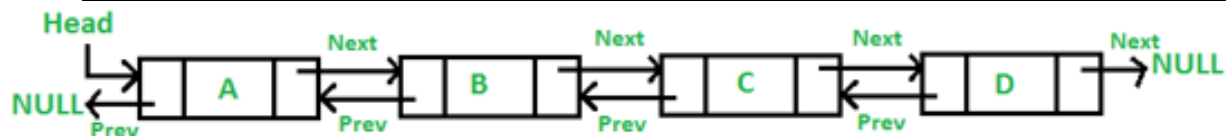
- We use collections to **hold & transfer objects from one location to another location**. To provide support for this requirement every collection class implements **Serializable & Cloneable interfaces**
- **ArrayList and Vector classes** implements **RandomAccess** interface, so that any random element 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 **[ob1, ob2, ob3]**

To get any class internal implementation, open eclipse, write class name, **CTRL+Click** – It will open compiled class file for reference.

### LinkedList- Internal implementation

- Underlying data structure is **DoubleLinkedList**
- **Insertion** order is preserved
- **Duplicates** are **allowed**
- **Heterogeneous** objects are **allowed**
- **null** insertion is **allowed**
- LinkedList **implements Serializable & Cloneable interfaces** but **not RandomAccess**
- **Best Choice for Insertion/Deletion, Worst for Retrieval operation.**
- It maintains the **index** also. But here index is LinkedList node index number.

<code>void addFirst(Object o)</code>	<code>Object getFirst()</code>	<code>Object removeFirst()</code>
<code>void addLast(Object o)</code>	<code>Object getLast()</code>	<code>Object removeLast()</code>
<code>Object set(int index, E element)</code>	<code>Object get(int index)</code>	



```
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
    }
}
```

### Implementation

1. LinkedList class in Java implements **List and Deque** interfaces and LinkedList implements it using **Doubly LinkedList**

2. In the implementation of the LinkedList class in Java there is a **private class Node** which provides the structure for a node in a doubly linked list.

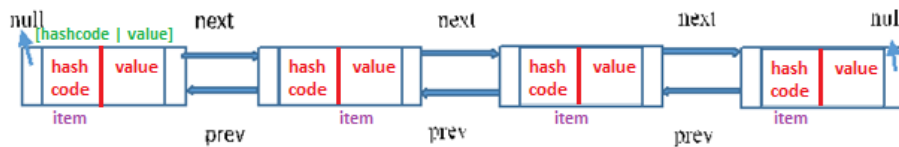
- It has **"item"** variable for holding the value
- and **two reference** to Node class itself for connecting to **next** and **previous** nodes.

```
private static class Node<E> {
    E item;
    Node<E> next;
    Node<E> prev;
}
```

```

Node(Node<E> prev, E element, Node<E> next) {
    this.item = element;
    this.next = next;
    this.prev = prev;
}
}

```



3. Reference to previous element of first node and Reference to next element of last node are **null** as there will be no elements before the first node and after the last node.

4. we have two Node variables - **first** , **last**. These will update the first & last element each time when we trying to add elements.

5. **You can insert the elements at both the ends and also in the middle of the LinkedList.**

6. If you call the regular **add()** method or **addLast()** method internally **linkLast()** method is called

```

/** * Links e as last element. */
void linkLast(E e) {
    final Node<E> l = last;
    final Node<E> newNode = new Node<>(l, e, null);
    last = newNode;
    if (l == null)
        first = newNode;
    else
        l.next = newNode;
    size++;
    modCount++;
}

```

6. If you call **addFirst()** method internally **linkFirst()** method is called.

```

private void linkFirst(E e) {
    final Node<E> f = first;
    final Node<E> newNode = new Node<>(null, e, f);
    first = newNode;
    if (f == null)
        last = newNode;
    else
        f.prev = newNode;
    size++;
    modCount++;
}

```

5. **Insertion and removal operations in LinkedList are faster than** the ArrayList. Because in LinkedList, there is no need to shift the elements after each insertion and removal, because we only add elements at First or Last no in-between. only references of next and previous elements need to be changed.

6. **Retrieval of the elements is very slow in** LinkedList as compared to ArrayList. Because in LinkedList, **you should traverse from beginning or end** (whichever is closer to the element) to reach the element.

```

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

        LinkedList<String> list = new LinkedList<>();
        list.add("One");
        list.add("Two");
        list.add("Three");
        System.out.println(list);
    }
}

```

Name	Value
args	String[0] (id=16)
list	LinkedList<E> (id=19)
first	LinkedList\$Node<E> (id=31)
last	LinkedList\$Node<E> (id=40)
modCount	3
size	3

Name	Value
list	LinkedList<E> (id=19)
first	LinkedList\$Node<E> (id=31)
item	"One" (id=33)
hash	0
value	(id=37)
[0]	0
[1]	n
[2]	e
next	LinkedList\$Node<E> (id=36)
item	"Two" (id=39)
hash	0
value	(id=61)
next	LinkedList\$Node<E> (id=40)
item	"Three" (id=41)
next	null
prev	LinkedList\$Node<E> (id=36)
prev	LinkedList\$Node<E> (id=31)
prev	null
last	LinkedList\$Node<E> (id=40)

### Vector- Internal implementation (**Synchronized** - Same as 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
- Vector implements Serializable, Cloneable & RandomAccess
- **Vector is Synchronized**

Add / Remove	Find	Special
addElement(Object o) removeElement(Object o) removeElementAt(int index) removeAllElements()	Object elementAt(int index) Object firstElement() Object lastElement()	Int size() Int capacity()//to know default/incremental capacity Enumeration elements()

```

public class VectorDemo {
    public static void main(String[] args) {
        Vector v = new Vector<>();
        v.add("one");
        v.add("two");
        v.add("three");
        System.out.println(v);
    }
}

```

Name	Value
args	String[0] (id=16)
v	Vector<E> (id=31)
capacityIncrement	0
elementCount	3
elementData	Object[10] (id=40)
[0]	"one" (id=42)
[1]	"two" (id=43)
[2]	"three" (id=44)
[3]	null
[4]	null
[5]	null
[6]	null
[7]	null
[8]	null
[9]	null
modCount	3

```

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()); //10

    v.addElement("Satya");
    System.out.println(v);
    System.out.println("After adding 11th element -Capacity:" + v.capacity());
    System.out.println("size : " + v.size());
}
}

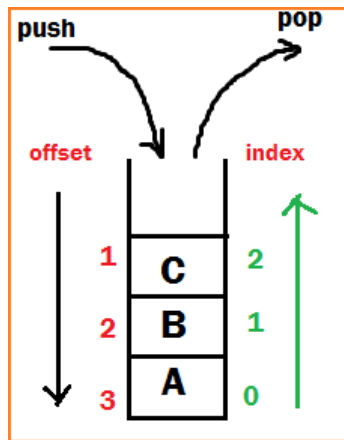
```

```

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

```

### Stack- Internal implementation



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

Object push(Object o) -Insert an object into top of the stack  
 Object pop() -Removes & returns from top of the stack  
 Object peak() -Just returns Object from top of the stack  
 boolean empty() - returns TRUE if stack is empty  
 int search(Object o) - returns offset if available otherwise -1

- Uses **Growable Array**, initial capacity as 10, same as ArrayList
- **Adding item** in Stack is called **PUSH**.
- **Removing item** from stack is called **POP**.
- **Push and pop** operations happen at **Top** of stack.
- Stack follows **LIFO** (Last in first out) - means last added element is removed first from stack
- Push - O(1) [as we push element at Top of Stack in java]
- Pop - O(1) [as popping is also done at Top of Stack in java]

```

public class Stackdemo {
    public static void main(String[] args) {
        Vector v = new Vector<>();
        v.add("one");
        v.add("two");
        v.add("three");
        System.out.println(v);
    }
}

```

Name	Value
args	String[0] (id=16)
v	Stack<E> (id=19)
capacityIncrement	0
elementCount	3
elementData	Object[10] (id=30)
[0]	"one" (id=32)
[1]	"two" (id=35)
[2]	"three" (id=36)
[3]	null
[4]	null
[5]	null
[6]	null
[7]	null
[8]	null
[9]	null
modCount	3

```

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

## Cursors - Enumeration VS Iterator VS ListIterator

ListIterator is subclass of List. so all the methods which are available in Iterator, also there in ListIterator.

**We must follow 3 steps to use Cursors in our application**

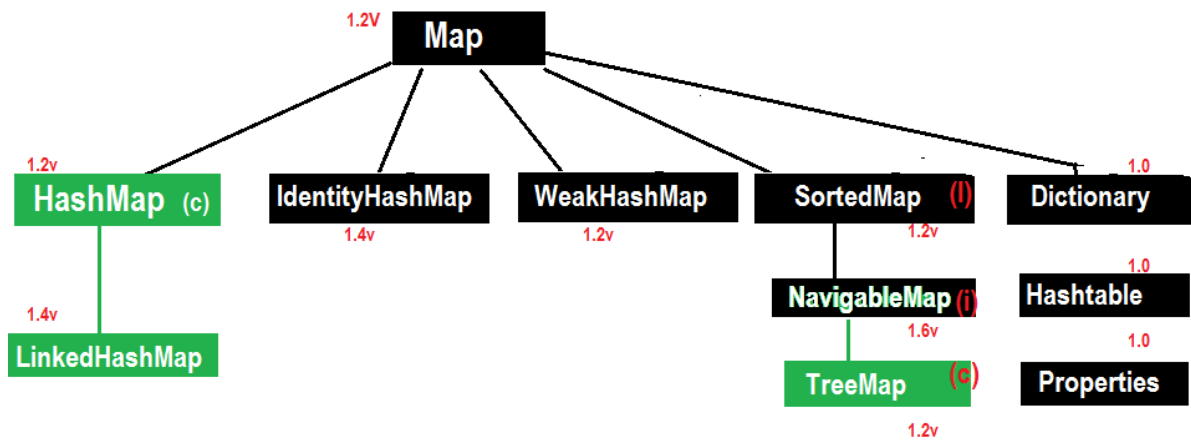
1. Get all elements and save it in cursor(iterator) object. Ex: `Iterator i= list.iterator()`
2. Check is next/previous element is exist or not Ex: `i.hasNext()`
3. Get the element

Enumeration (vector/stack)	Iterator(ArrayList)	ListIterator (LinkedList)
Can iterate over a Collection	Can iterate over a Collection	Can iterate over a Collection
<b>Remove operation not allowed</b>	Remove operation allowed	Remove operation allowed
<b>Add operation not allowed</b>	<b>Add operation not allowed</b>	Add operation allowed
<b>Backward direction not allowed</b>	<b>Backward not allowed</b>	Backward direction allowed
<b>1. Enumeration elements()</b> Ex. Enumeration = v.elements()	<b>1. Iterator iterator()</b> Ex. Iterator = l.iterator()	<b>1. ListIterator listIterator()</b> Ex. ListIterator = l.listIterator ()

<p>2. <b>boolean</b> hasMoreElements() 3. <b>E</b> nextElement()</p>	<p>2. <b>boolean</b> hasNext() 3. <b>E</b> next() void remove()</p>	<p>2. <b>boolean</b> hasNext() <b>boolean</b> hasPrevious()  add(<b>E</b> e) nextIndex() previous() previousIndex() <b>remove()</b> <b>next()</b> set(<b>E</b> e)</p>
--	---	---

Enumeration Example using Vector	Iterator Example using ArrayList
<pre>public class VectorEnumeration { public static void main(String[] args) { Vector v = new Vector(); for (int i = 1; i &lt;= 10; i++) { v.addElement(i); } Enumeration e = v.elements(); while (e.hasMoreElements()) { Object s = (Object) e.nextElement(); System.out.print(s + ","); } } }</pre>	<pre>public class ArrayListIterator { public static void main(String[] a) { ArrayList l = new ArrayList&lt;&gt;(); 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+","); } } }</pre>
1,2,3,4,5,6,7,8,9,10,	A,10,A,null,

## Map



- **Entry Interface:** One Row (pair of <K,V>) treated as an Entry

```
public interface Map {
    .
    .
    interface Entry{
        Object setValue(Object newVal)
        Object getKey();
        Object getValue();
    }
}
```



- **Object put(Object Key, Object value)** – To add one <key, value> pair to the Map.

```

null <= m.put(101, "Satya")
null <= m.put(102, "Rakesh")
Ravi <= m.put(102, "Rakesh")

```

if the <key> is not present, then Value will be placed against <Key> & returns **null**.

if the <key> is already present then oldvalue will be replaced with new value & returns **old value**.

- Object putAll(Map m)
- Void putAll(Map m)
- Object get(Object key)
- Object remove(Object key)
- boolean containsKey(Object Key)
- boolean containsValue(Object value)
- boolean isEmpty()
- int size()
- void clear

### HashMap – Internal implementation (HashTable)

- The underlying data structure is **HashTable(Bucket) + LinkedList(Storing Item)**
- Insertion order is not preserved & it is based on **Hashcode of <Keys>**
- Duplicate keys are NOT allowed, but values can be duplicated.
- Heterogeneous objects are allowed for both Key & Value
- **null is allowed for key (only once)**
- **null is allowed for Values (any no. of times)**
- HashMap implements **Serializable & Cloneable interfaces** but **not RandomAccess**
- HashMap is the best choice for **Searching** operations

### Constructors

- **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
- **HashMap h = new HashMap (int intialcapacity) // Def. fill ratio = 0.75**
- **HashMap h = new HashMap (int intialcapacity, float fillRatio)**
- **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 existing 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 existing 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

```

**Internal implementation:** <http://www.thejavageek.com/2016/03/12/working-of-hashmap-in-java/>

- **Initial Capacity (16):** This is the capacity of HashMap to store number of key value pairs
- **Load Factor/Fill ratio (0.75):** is a parameter responsible to determine when to increase size of HashMap.
- **Threshold value( $cap * 0.75 = 12$ ):** When number of key value pairs is more than threshold value, then HashMap is resized with increased capacity. Here Threshold value is **12**

```

public class HashMapDemo {
    public static void main(String[] args) {
        HashMap<String, String> map = new HashMap<>();

        map.put("one", "AAA");
        map.put("two", "BBB");
        map.put("three", "CCC");
        map.put("four", "DDD");
        System.out.println(map);
    }
}

```

```
{ four=DDD, one=AAA, two=ZZZ, three=CCC} //insertion ordered not preserved
```

**1. On this line of code, it will create the HashMap object with,**

```
HashMap<String, String> map = new HashMap<>();
```

- Default **Bucket size is 16, with Entry table[] of 16 buckets**
- load factor as **0.75**
- initializes with **null** values

Name	Value
map	HashMap<K,V> (id=19)
entrySet	null
keySet	null
loadFactor	0.75
modCount	0
size	0
table	HashMap\$Node<K,V>[16] (id=25)
[0]	null
[1]	null
[2]	null
[3]	null
[4]	null
[5]	null
[6]	null
[7]	HashMap\$Node<K,V> (id=39)
[8]	null
[9]	null
[10]	null
[11]	null
[12]	null
[13]	null
[14]	null
[15]	null
threshold	12

2. Each entry into HashMap is an **Entry Object**, which contains [hashcode, key, value, next] data fields.



```
static class Entry<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    V value;
    Entry<K,V> next;
}
```

## put() method

Now we are ready, by adding `map.put("one", "AAA");` following actions will be performed

```
public V put(K key, V value)
{
    if (key == null)
        return putForNullKey(value);    → hashcode =0, bucketlocation =0

    int hash = hash(key.hashCode());    → 1.Gets hashcode
    int i = indexOf(hash, table.length); → 2.Gets Bucket location

    //3. Loop all Map elements, Compare new Key with existing keys
    for (Entry<K,V> e = table[i]; e != null; e = e.next)
    {
        Object k;
        if (e.hash == hash && ((k = e.key) == key || key.equals(k))) →3.1 If Key already Exist
        {
            V oldValue = e.value;
            e.value = value;
            e.recordAccess(this);
            return oldValue;
        }
    }
    modCount++;
    addEntry(hash, key, value, i); →3.2 If Key not Exist already, add item to Map & Return null
    return null;
}
```

### 1. Using hashCode() method

- First, it checks for the if **the key given is null or not**, if the given **key is null** it will be stored in the **'0'th position** as the hashcode of null will be 0(**hashcode=0**), **& if key is not null, then it gets the hashcode of the Key**

```
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

- Now **indexFor(hash, table.length)** function is called to calculate exact index(Bucket) position for storing the Entry object. It is a BitWise **&(and)** Operation, like **Bucket = (hashcode) & (tablecapacity)**, for the above code **bkt = 110183 &15 = 7(7<sup>th</sup> bucket)**

Basically, following operation is performed to calculate index.

```
index = hashCode(key) & (table.length).
```

Bitwise Calculator

Bitwise AND, OR, XOR Bit Shift			
Data type:	Decimal	Result in binary	111
Number1:	110183	Result in decimal	7
Number2:	15	Result in hexadecimal	7
Bitwise operator:	AND OR XOR		

<https://www.miniwebtool.com/bitwise-calculator/>

## 2.Using equals () method

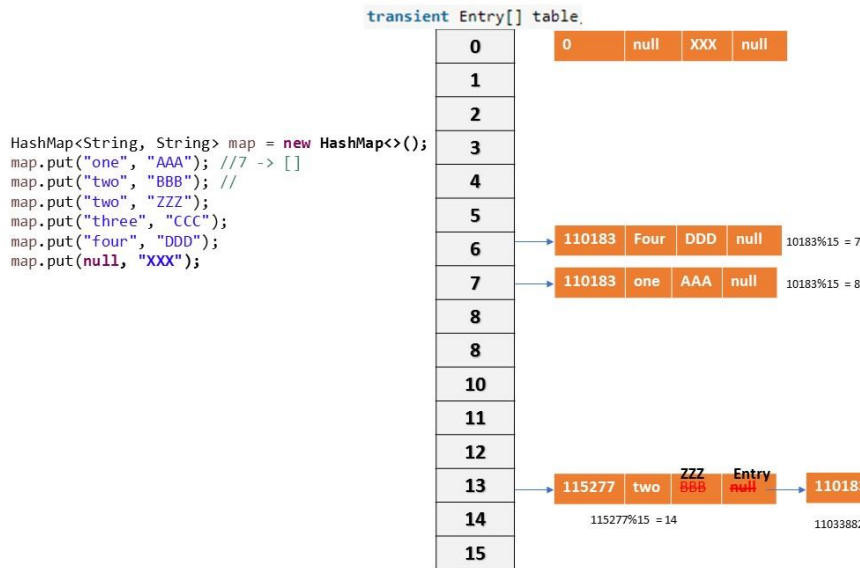
- Now it compares the current key value with existed key values by **newkey.equals(oldkey) method**, if found it updates the value & returns oldValue. if not found it will create the new node by **addEntry(hash, key, value, i);**and add it to the calculated Bucket location(7)

## Get() method

we already know how Entry objects are stored in a bucket and what happens in the case of Hash Collision. it is easy to understand what happens when key object is passed in the **get(key)** method of the HashMap to retrieve a value.

- Using the key (passed in the get() method) again hash value will be calculated to determine the bucket location (index) , where that Entry object is stored.
- In case there are more than one Entry object with in the same bucket location (stored as a linked-list) **equals()** method will be used to find out the correct key.
- Once exact key is found, get() method will return the value object stored in the Entry object

```
public V get(Object key)
{
    if (key == null)
        return getForNullKey();
    int hash = hash(key.hashCode());
    for (Entry<K,V> e = table[indexFor(hash, table.length)];e != null;e = e.next)
    {
        Object k;
        if (e.hash == hash && ((k = e.key) == key || key.equals(k)))
            return e.value;
    }
    return null;
}
```



## Collision

If two key objects are returns same hashcode, then which object should insert at a given Bucket is known as "Collision". So, in case of collision, Entry objects are stored in **LinkedList** form. The address of the next element will be stored in previous element.

## Rehashing

- When the number of items in map, crosses the Load factor limit (**12** if size=16) at that time **HashMap doubles its capacity** and hashcode is re-calculated of already stored elements for even distribution of key-value pairs across new buckets.
- So, for each **existing key-value pair, hashcode is re-calculated with increased capacity** as a parameter, which results in either placing the item in same bucket or in different bucket.
- Rehashing is done to distribute items across the new length HashMap, so that **get()** and **put()** operation time complexity remains **O(1)**.
- HashMap maintain complexity is O(1). This is same while inserting data in and getting data from HashMap. But for 13th key-value pair, put request will no longer be O(1), because as soon as map will realize that 13th element came in, that is 75% of map is filled.

**Rehashing**



**Note:**

1. Good hashcode method distributes item across all buckets.
2. For each put operation, It checks whether Threshold limit of Load Factor 0.75 is reached. if yes, then rehashing process will be started

### Points to note -

- HashMap works on the principal of **hashing**.
- HashMap uses the `hashCode()` method to calculate a hash value. Hash value is calculated using the key object. This hash value is used to find the correct bucket where Entry object will be stored.
- HashMap uses the `equals()` method to find the correct key whose value is to be retrieved in case of `get()` and to find if that key already exists or not in case of `put()`.
- Hashing collision means more than one key having the same hash value. In that case Entry objects are stored as a linked-list within a same bucket.
- Within a bucket, values are stored as Entry objects which contain both key and value.
- On High hash Collisions, java people observed that LinkedList is very slow.
- **In Java 8 hash collision uses BalancedTree instead of LinkedLists** after a certain threshold is reached while storing values. This improves the worst case performance from  $O(n)$  to  $O(\log n)$ .

### Improvements in Java 8

- There is a performance improvement for HashMap objects where there are lots of collisions in the keys by using **BalancedTree** rather than **LinkedLists** to store map entries
- The principal idea is that **once the number of items in a hash bucket grows beyond a certain threshold, that bucket will switch from using a LinkedLists of entries to a BalancedTree. In the case of high hash collisions, this will improve worst-case performance from  $O(n)$  to  $O(\log n)$ .**

### How to avoid Hash Collision

<https://www.journaldev.com/21095/java-equals-hashcode>

If you are planning to use a class as HashMap key, then it's must to override both `equals()` and `hashCode()` methods.

Let's see what happens when we rely on default implementation of `equals()` and `hashCode()` methods and use a custom class as HashMap key.

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class DataKey {

    private int id;
    private String name;

    public static void main(String[] args) {
        Map<DataKey, Integer> map = new HashMap<DataKey, Integer>();
        DataKey key = new DataKey(1, "AAA");
        map.put(key, 100);
        System.out.println("Key 1 : "+key.hashCode());
    }
}
```

```

//Create Key object with Same Data
DataKey key2 = new DataKey(1, "AAA");
System.out.println("Key 2 : "+key2.hashCode());
//We are trying to get VALUE , by Passing same data Key Object as KEY
Integer value = map.get(key2);
System.out.println(value);
}
}

```

When we run above program, it will print **null**. Because notice that hash code values of both the objects are different and hence value is not found.

So to avoid this we need to return

- same `hashCode()` if data for two objects are same.
- Override `equals()` method return true only if all data elements are matched.

```

public class DataKey {
    private int id;
    private String name;

    public boolean equals(Object ob) {
        if(ob == null)
            return false;

        if(this == ob)
            return true;        // Reference equality

        if(!(ob instanceof DataKey))
            return false;

        DataKey obj = (DataKey) ob;
        return (name.equals(obj.getName())) && (id == obj.id);
    }

    public int hashCode() {
        return name.hashCode() ^ id;
    }
}

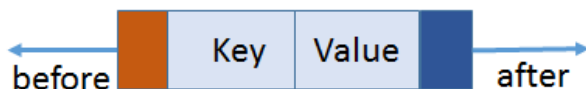
```

for same data `hashCode()` returns same hashcode. Now we will reduce the hash collision, if object with same data is trying put as key, will we just update value – because key with same data already exists.

### LinkedHashMap – Internal implementation

- `LinkedHashMap` is just an extension of `HashMap`, internally uses **HashTable+DoublyLinkedList**
- It has two references **head** and **tail** which will keep track of the latest object inserted and the first object inserted

1. Each node in a `LinkedHashMap` needs to have information about previous node and next node as the order in which they are accessed is important. The structure is as follows.

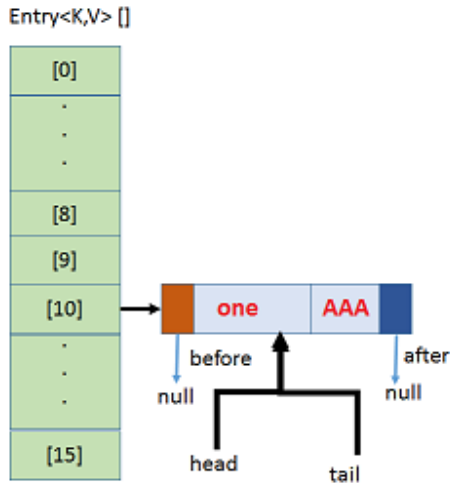


```

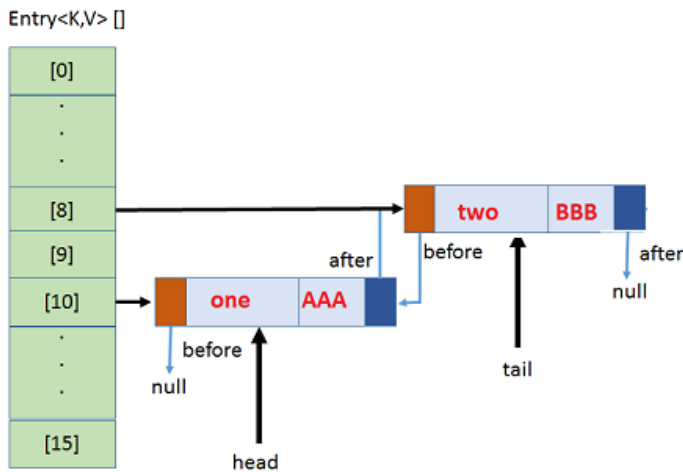
class Entry<K,V> extends HashMap.Node<K,V> {
    Entry<K,V> before, after;
    Entry(int hash, K key, V value, Node<K,V> next) {
        super(hash, key, value, next);
    }
}

```

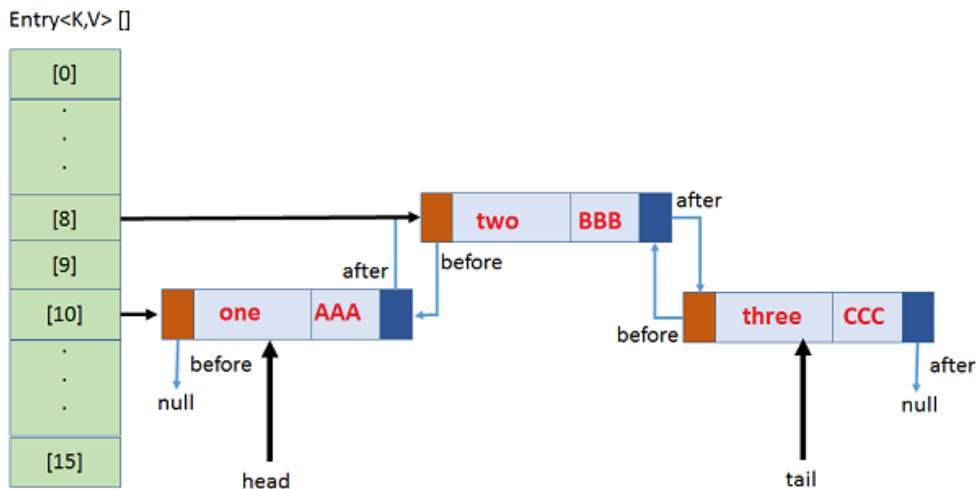
2. After inserting `map.put("one", "AAA");` Bucket with 16 capacity is created, hashcode & index will be calculated. Here **one** is added, **head** and **tail** will refer to it.



3. On adding `map.put("two", "BBB");`, inserted in 8<sup>th</sup> bucket & it is next to "one", so **head** → **one**, **tail** → **two**, **before** & **after** will be linked, like this all elements will be added & Insertion order will be maintained.



4. If any collision, new element will be added in same Bucket location with LinkedList & points to next.





```

public class HashMapDemo {
    public static void main(String[] args) {
        LinkedHashMap<String, String> map = new LinkedHashMap<>();
        map.put("one", "AAA"); //7 -> []
        map.put("two", "BBB"); //
        map.put("two", "ZZZ");
        map.put("three", "CCC");
        map.put("four", "DDD");
        map.put(null, "XXX");

        System.out.println(map);
    }
}
{one=AAA, two=ZZZ, three=CCC, four=DDD, null=XXX}

```

So here Insertion Order is preserved, elements the order they added, same order they will store.

```

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 existing 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 existing key:Ravi
All keys : [one, two, three, four]
All Values : [Satya, Madhu, Rakesh, Surya]
Both Key-Values
-----
one : Satya
two : Madhu
three : Rakesh

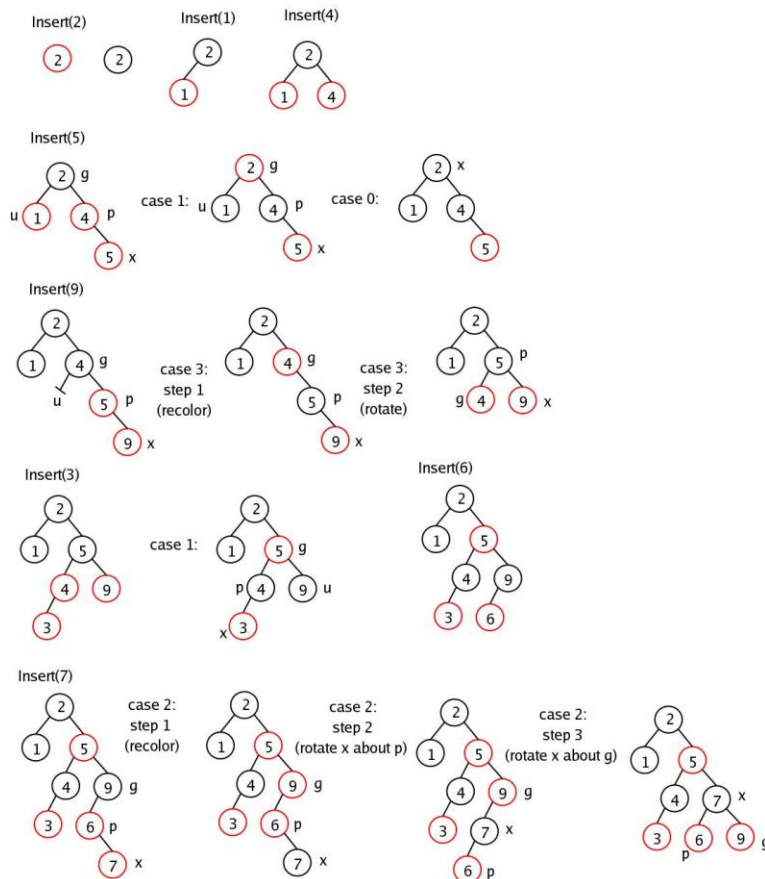
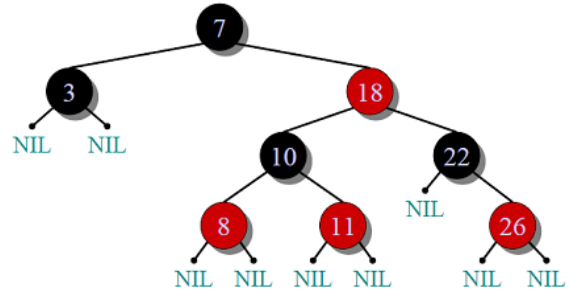
```

### TreeMap – Internal implementation

- 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 **duplicated**
- If we are depending on default **Natural Sorting Order**, then **KEYS should be Homogeneous & Comparable** otherwise we will get Runtime exception saying **ClassCastException**
- If we are defining our **Own Sorting Order by Comparator**, then **KEYS should need not be Homogeneous & Comparable**. We can take Heterogeneous & non-comparable Objects also.

## Red-Black Tree

- 1) Every node has a color either **red** or **black**.
- 2) **Root** of tree is always **black**.
- 3) No two adjacent red nodes (A red node cannot have a red parent or red child).
- 4) Every path from **root to a NULL** node has same number of black nodes.



### Constructors

`TreeMap h = new TreeMap () //Default. Sorting Order`

Creates an Empty TreeMap Object, all elements inserted in **Default Natural Sorting Order**

`TreeMap h = new TreeMap (Comparator c) //Customized. Sorting Order`

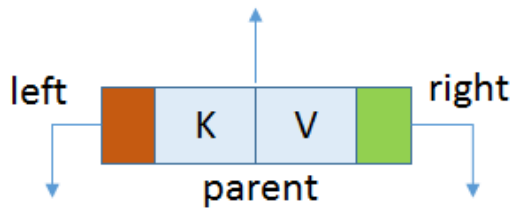
Creates an Empty TreeMap Object, all elements inserted in **Customized Natural Sorting Order**

`TreeMap h = new TreeMap (Map c)`

`TreeMap h = new TreeMap (SortedMap s)`

## Implementation

1. TreeMap is based on tree data structure. Each node in tree will have three references parent(**key,value**), **right** and **left** element.



- The **left** element will always be logically **less** than **parent** element.
- The **right** element will always be logically greater than OR equal to **parent** element

```
static final class Entry<K,V> implements Map.Entry<K,V> {
    Entry<K,V> left;
    Entry<K,V> right;
    Entry<K,V> parent;
    K key;
    V value;
}
```

2. Comparison of Objects is done by natural order i.e. those object who implement **Comparable(default)** interface and override **compareTo(Object obj)** method. Based on the return value,

- If **obj1.compareTo(obj2)**, if **obj1<obj2** then returns negative number
- If **obj1.compareTo(obj2)**, if **obj1>obj2** then returns positive number
- If **obj1.compareTo(obj2)**, if **obj1==obj2** then returns zero

3. When we use **put(K,V)** method, it checks if root is pointing anywhere or not. if not, it makes the instance of **Entry<K,V>** and point to root;

4. The constructor of **Entry<K,V>** takes key, value and parent. In this case parent is **null**;

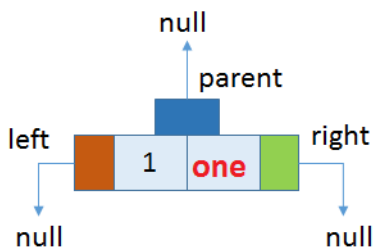
5. For the next time we enter using **put(K,V)** it first identifies the comparison mechanism to use.

6. First it checks the **Comparator** class is present or not. (This class is passed when creating the instance of **TreeMap**). If not present it uses the **Key's Comparable** implementation.

7. It then traverse through root and compares each node with the node entered and depending upon the comparison places the node either left or right of the parent node.

```
treeMap.put(1, "one"); //1
```

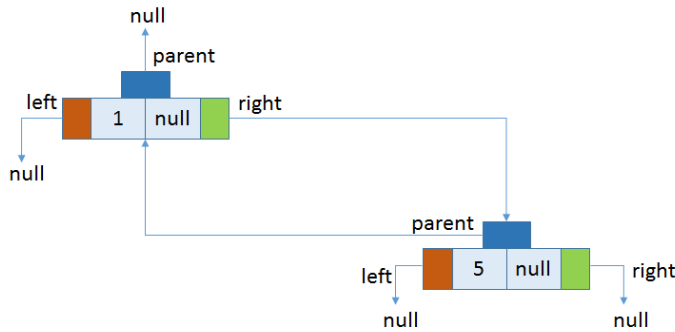
there are no elements in it. So, 1 is the first object being inserted as key. This is treated as root node



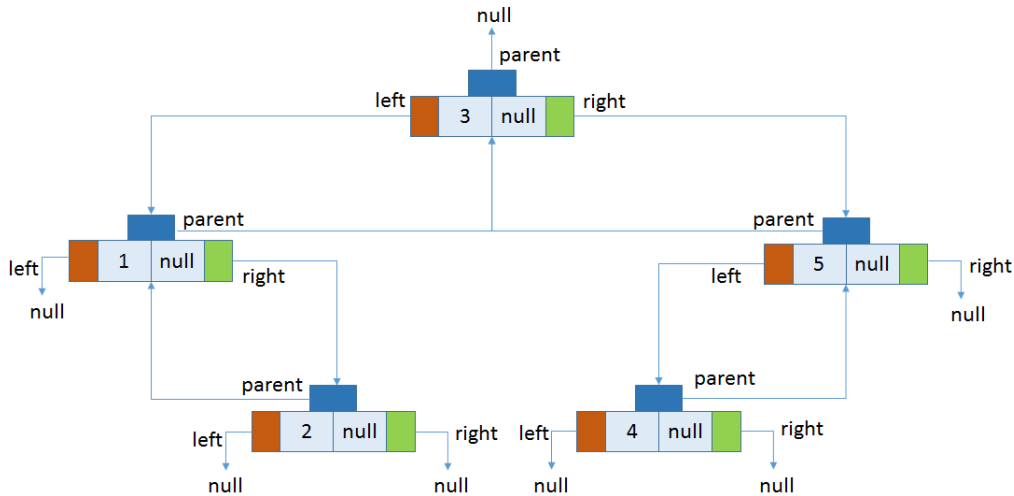
```
treeMap.put(5, "five"); //2
```

Now, **5** is logically greater than **1** and hence according to our rules,

- **5** will be placed to the right of **1**.
- **1** will be parent of **5**.



```
treeMap.put(3, "three"); //3
treeMap.put(2, "two"); //4
treeMap.put(4, "four"); //5, after inserting all these final structure will be
```



args	>String[] (id=10)
treeMap	TreeMap<K,V> (id=19)
comparator	null
descendingMap	null
entrySet	null
keySet	null
modCount	5
navigableKeySet	null
root	TreeMap\$Entry<K,V> (id=36)
color	true
key	Integer (id=37)
left	TreeMap\$Entry<K,V> (id=27)
parent	null
right	TreeMap\$Entry<K,V> (id=38)
value	"three" (id=39)
size	5
values	null

### IdentityHashMap

It is exactly same as [HashMap](#) (including methods & constructors) except following differences

- In the case of Normal HashMap JVM will use `.equals()` method to identify Duplicate keys, which is meant for Content comparison

- But, In the case of IdentityHashMap JVM will use `==` operator to identify Duplicate keys, which is meant **for reference comparison** or **address comparison**

`==` is for **reference comparison** or **address comparison**

`.equals ()` is for **content comparison**

```
Integer i1 = new Integer(10); → i1 -> 10
Integer i2 = new Integer(10); → i2 -> 10
System.out.println(i1 == i2); //FALSE
System.out.println(i1.equals( i2)); //TRUE
```

HashMap Example	IdentityHashMap Example
<pre>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}     } }</pre>	<pre>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}     } }</pre>
{10=Surya}	{10=Satya, 10=Surya}

### WeakHashMap

It is exactly same as HashMap except following difference

- In the case of HashMap even though Object doesn't have any reference it is **NOT eligible** for garbage collection, if it is associated with HashMap. That means HashMap dominates Garbage collector.
- But the case of **WeakHashMap** if Object doesn't have any references it is **eligible** for `gc()` even though object associated with WeakHashMap. that means Garbage collector dominates WeakHashMap

This Temp class is common for HashMap & WeakHashMap 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 output 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);
    }
}

```

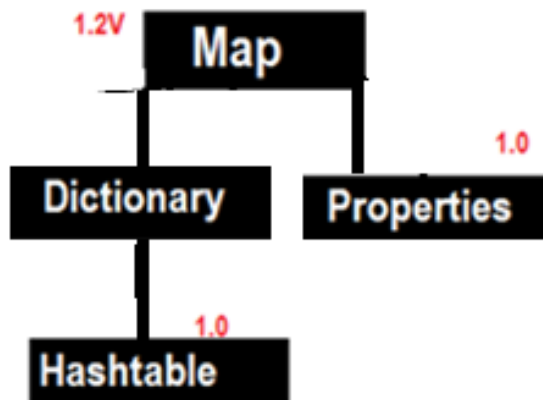
```

{Temp=Satya}
Finalize Called
{}

```

In the above example Temp object is eligible for gc() because it is associated with WeakHashMap.in this case output is {Temp=Satya} Finalize Called {}

### Legacy Classes on Map



The **Dictionary** class is the abstract parent of any class, such as Hashtable, which maps keys to values. Every key and every value is an object.

#### 1.Hashtable

- Underlying D.S is **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
- Heterogeneous 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, Cloneable 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

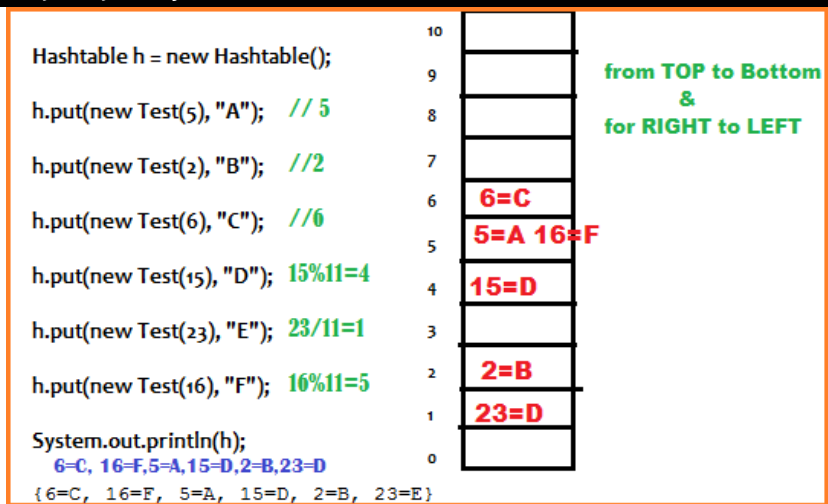
Creates an empty Object with def. initial capacity **16** & def. fill ratio 0.75

**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;
    }
    public int hashCode() {
        return i;
    }
    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}
```



HashMap	Hashtable
HashMap is <b>non-synchronized</b>	Hashtable is <b>synchronized</b>
Performance is high because no threads waiting	Performance is Low because threads may wait
<b>nulls allowed</b> for both <Key & Value>	<b>nulls</b> is NOT allowed for both <Key & Value>

Introduced in 1.2 version

Introduced in 1.0 version (Legacy)

By default, HashMap is non-synchronized but we can get Synchronized version of HashMap by using `synchronizedMap()` of collections class

```
HashMap m = new HashMap()  
Map m1 = Collections.synchronizedMap(m)
```

## 2 Properties

In our project if anything which changes frequently like Database names, username, password etc. we use properties file to store those & java program used to read properties file

```
Properties p = new Properties ()
```

### KEY & Values must be String type

#### Methods

1. `String getProperty(String name);`
2. `String setProperty(String name, value);`
3. `Enumeration propertyNames();`
4. `void load(InputStream is)`  
Load properties from properties file into java properties Object
5. `void store(OutputStream is, String comment)`  
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
```

Multiple values in `java.util.Properties`

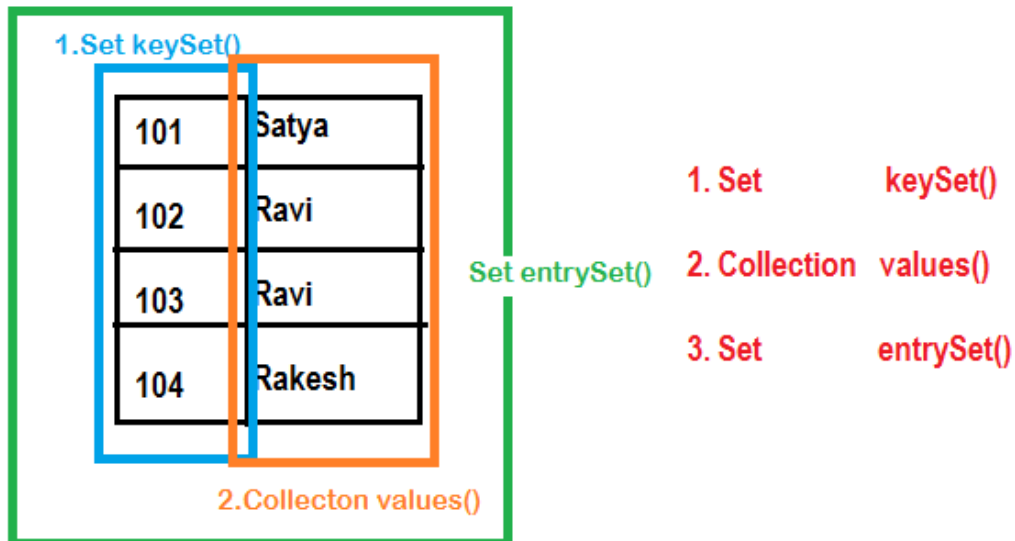
```
foo=1,2  
String[] foos = properties.getProperty("foo").split(",");
```

## Set

- 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 Set



- Set **doesn't have any new methods & we have to use collection interface methods only.**
- In All **Hash related collections insertion** is based on **Hashcode**.so no insertion order preserved.



### HashSet – Internal implementation

- The underlying data structure is **Hashtable**, internally uses **HashMap**
- Duplicate Objects are Not Allowed
- Insertion Order is Not preserved & it is based **hashcode** of Objects
- **null** Insertion is possible (Only once), Heterogeneous Objects are allowed
- Implements **Serializable** & **Cloneable** 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**

### Constructors

**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

**HashSet h = new HashSet (int intialcapacity) // Def. fill ratio = 0.75**

**HashSet h = new HashSet (int intialcapacity, float fillRatio)**

**HashSet h = new HashSet (Collection c)**

### Implementation

**1.HashSet** uses **HashMap** internally to store its objects. Whenever you create a **HashSet** object, one **HashMap** object associated with it is also created.

```
public HashSet()
{
    map = new HashMap<>();           //Creating internally backing HashMap object
}

public HashSet(int initialCapacity, float loadFactor)
{
```

```
map = new HashMap<>(initialCapacity, loadFactor);
}
```

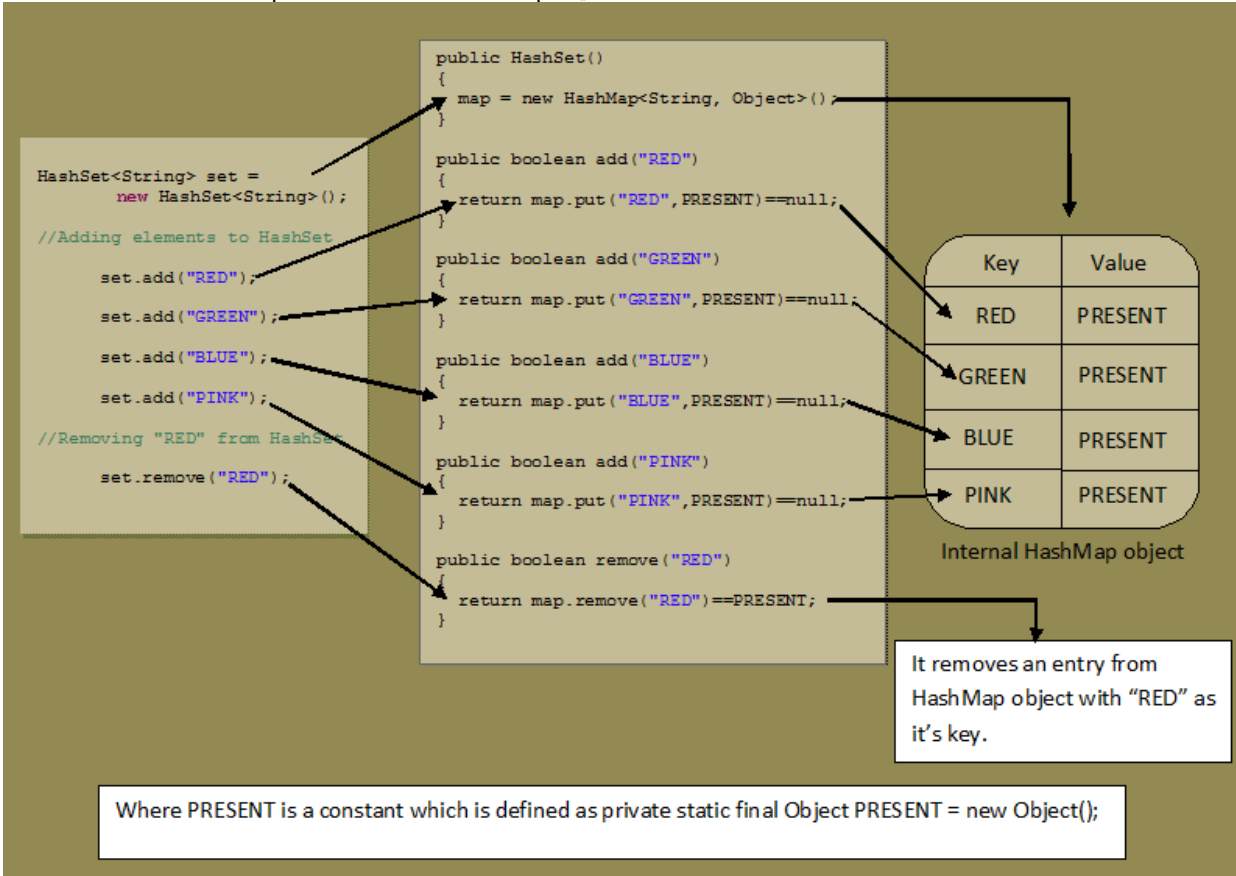
2.The elements you add into HashSet are stored as **keys** of this HashMap object. The value associated with those keys will be a **constant (PRESENT)**.

**Add Method**

- **add()** method of HashSet class internally calls **put()** method of backing HashMap object by passing the element you have specified as a **key** and constant **"PRESENT"** as its value.

```
private static final Object PRESENT = new Object();
public boolean add(E e)
{
    return map.put(e, PRESENT)==null;
}
```

- Here hash function is calculated using **value** we are trying to insert. That's why only unique values are stored in the HashSet.
- When element is added to **HashSet** using **add(E e)** method internally HashSet calls **put()** method of the HashMap where the value passed in the add method becomes key in the put() method. A dummy value "PRESENT" is passed as value in the put() method.



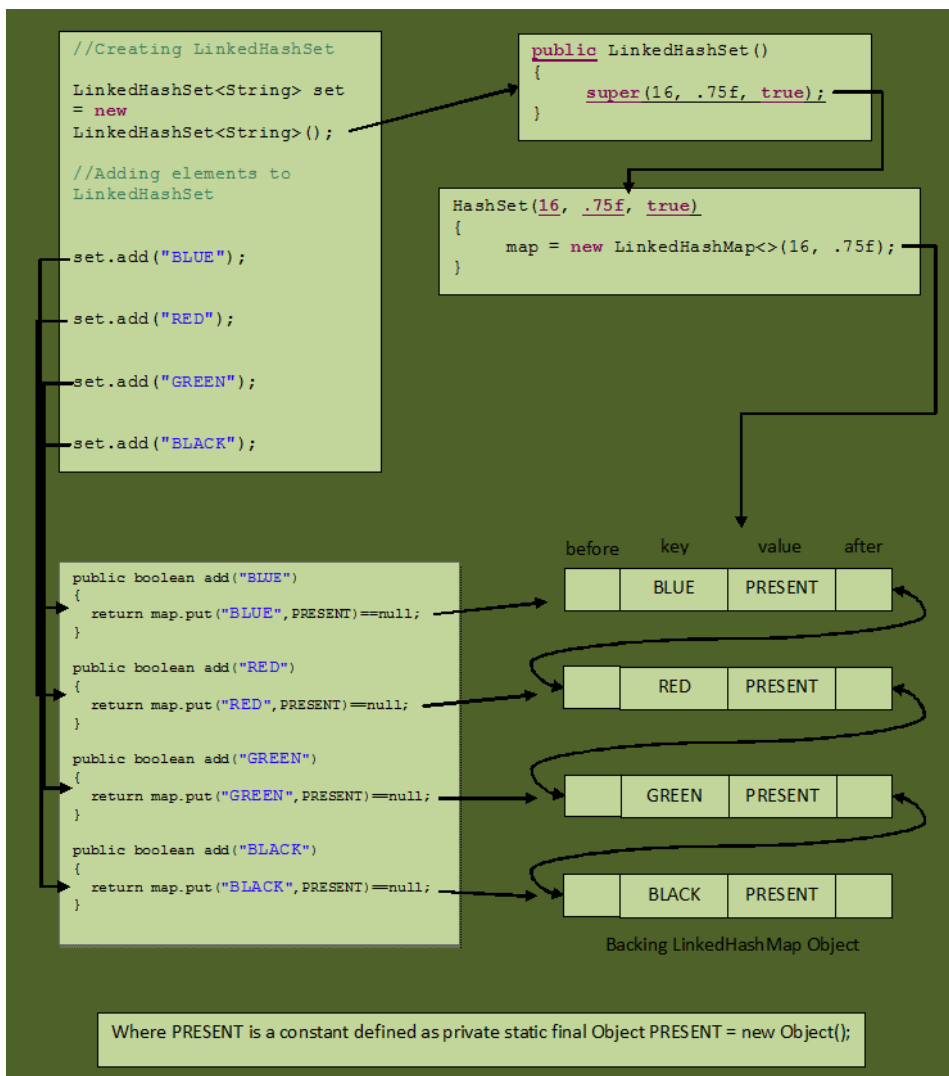
```
public class HashSetDemo {
public static void main(String[] args) {
    HashSet h = new HashSet();
    h.add("A");
    h.add("B");
    h.add("C");
    h.add(10);
    h.add(null);
    System.out.println(h.add("A")); //False
    System.out.println(h);
}
```

```
}  
}  
false  
[null, A, B, C, 10]
```

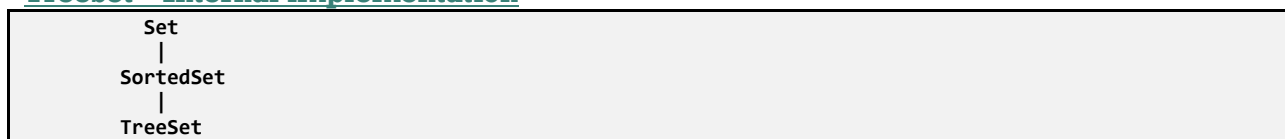
### LinkedHashSet – Internal implementation

- LinkedHashSet is an **extended version** of HashSet. HashSet doesn't follow any order where as LinkedHashSet maintains **insertion order**.
- HashSet uses **HashMap object** internally to store its elements where as LinkedHashSet uses **LinkedHashMap object** internally to store and process it's elements
- LinkedHashSet, elements you insert are stored as **keys of LinkedHashMap** object. The values of these keys will be the same constant i.e **"PRESENT"**.
- The insertion order of elements into LinkedHashMap are maintained by adding two new fields to this class. They are **before** and **after**. These two fields hold the references to previous and next elements. These two fields make LinkedHashMap to function as a doubly linked list.

```
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]
```



## TreeSet – Internal implementation



### java.util.SortedSet (Interface)

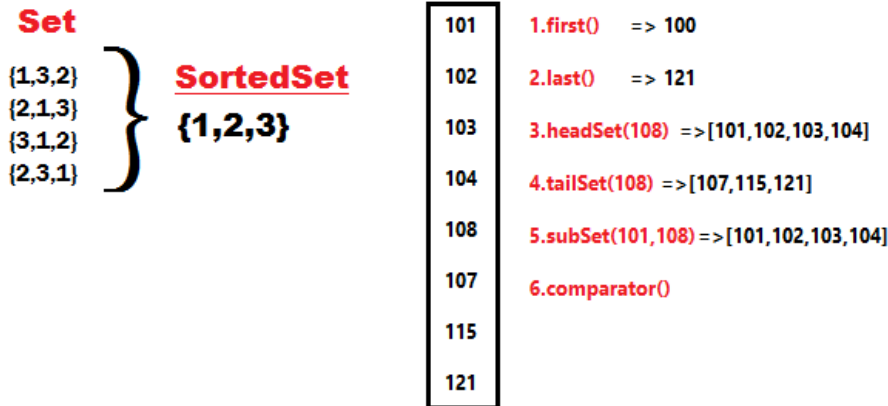
- SortedSet is the child interface of Set
- If we want to represent a group of individual objects according to some sorting order without duplicates, then we should go for SortedSet.

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)



## TreeSet Implementation

- Underlying D.S is Red-Black Tree
- 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
- Non-Comparable objects are not allowed. If we try it throws : [java.lang.ClassCastException: java.lang.StringBuffer cannot be cast to java.lang.Comparable](#)
- **null** Insertion allowed (Only once)
- **TreeSet** implements Serializable & Cloneable but not RandomAccess
- All objects are inserted based on some sorting order either **default** or **customized** sorting order.

## Constructors

**TreeSet h = new TreeSet () //Default. SortingOrder**

Creates an Empty TreeSet Object, all the elements inserted according to Default Natural Sorting Order

**TreeSet h = new TreeSet (Comparator c) //Customized. SortingOrder**

Creates an Empty TreeSet Object, all the elements inserted according to Customized Sorting Order

**TreeSet h = new TreeSet (Collection c)**

**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]

## Implementation

TreeSet is like HashSet which contains the unique elements only but in a sorted manner.

TreeSet uses **TreeMap** internally to store its elements.

```
public class TreeSet<E> extends AbstractSet<E> implements NavigableSet, Cloneable, Serializable
{
    private static final Object PRESENT = new Object();

    public TreeSet() {
        this(new TreeMap<E, Object>());
    }
    public boolean add(E e) {
        return map.put(e, PRESENT) == null;
    }
}
```

## Comparable & Comparator

```
public class TreeSetStringBuffer {
    public static void main(String[] args) {
        TreeSet<StringBuffer> 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 Sorting Order objects should be **Homogeneous** (same type objects) & **Comparable**. Otherwise we will get Runtime Exception [java.lang.ClassCastException](#)

```
public static void main(String[] args) {
    // List list = new ArrayList(); //[1, 2, 3, 4] - No Error
    // Set list = new HashSet(); //[1, 2, 3, 4] - No Error
    Set list = new TreeSet(); // [1, 2, 3, 4] - Error

    list.add("1");
    list.add("2");
    list.add("3");
    list.add(4);
    System.out.println(list);
}
```

Exception in thread "main" [java.lang.ClassCastException](#): java.lang.String cannot be cast to java.lang.Integer  
at java.lang.Integer.compareTo(Integer.java:52)

- 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
```

So it throws Exception in thread "main" [java.lang.ClassCastException](#): java.lang.StringBuffer cannot be cast to java.lang.Comparable

- If we Take **EmpBo**, if we pass employee list Objects to **Collection.sort(EmpBo)** method, it will throw Error, because it only accepts objects of Comparable types only.

Exception in thread "main" java.lang.Error: Unresolved compilation problem:  
The method sort(List<T>) in the type Collections is not applicable for the args.(List<Employee>)

We have Two ways to provide Sorting order for StringBuffer & Other classes which are not implementing Comparable Interface

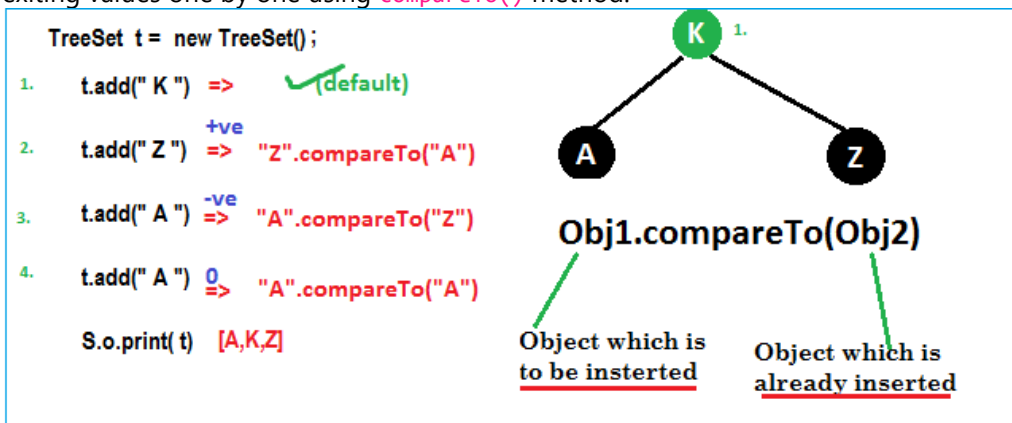
1. Implement `java.lang.Comparable` interface & override `int compareTo(Object)`
2. Implement `java.util.Comparator` interface & override `int compare(Object, Object)`

## 1. Comparable interface

- It provides single sorting sequence only i.e. you can sort the elements on based on single data member only. For example, it may be rollno, name, age or any one of them, not all else.
- Comparable is an interface defining a strategy of comparing an object with other objects of the same type. This is called the class's "natural ordering".so we need to define `compareTo()` method
- We use `public int compareTo(Object obj)` to compare the current object with the specified object.

```
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. It will compare inserting value with exiting values one by one using `compareTo()` method.



```
public class Employee implements Comparable<Employee> {  
    int id;  
    String name;  
    double salary;  
    //Setters & Getters  
  
    public Employee(int id, String name, double salary) {  
        super();  
        this.id = id;  
        this.name = name;  
        this.salary = salary;  
    }  
    @Override  
    public String toString() {  
        return "Employee [id=" + id + ", name=" + name + ", salary=" + salary + "];"  
    }  
}
```

```

    }

    @Override
    public int compareTo(Employee o) {
//Here add(105) object will the CurrentID, Object.ID is existing element ID's. Which are not sorted.
        System.out.println("Current ID : " + this.id + " \t Obj.ID : " + o.id);

        if (this.id < o.id) {
            return -1;
        } else if (this.id > o.id) {
            return 1;
        } else {
            return 0;
        }
    }

    public static void main(String[] args) {

        Set<Employee> employees = new TreeSet<Employee>();
        employees.add(new Employee(105, "Satya", 3000));
        System.out.println("After 105 : -----> " + employees + "\n");

        employees.add(new Employee(102, "RAJ", 2000));
        System.out.println("After 102 : -----> " + employees + "\n");

        employees.add(new Employee(104, "Madhu", 5000));
        System.out.println("After 104 : -----> " + employees + "\n");

        employees.add(new Employee(101, "Srini", 1000));
        System.out.println("After 101 : -----> " + employees + "\n");

        employees.add(new Employee(103, "Vinod", 4000));
        System.out.println("After 103 : -----> " + employees + "\n");

        //See here we are adding 100, which is less than all exiting elements.
        //So, it will compare with almost all elements using compareTo() method
        employees.add(new Employee(100, "Vinod", 1000));
        System.out.println("After 100 : -----> " + employees + "\n");

        System.out.println("After : " + employees + "\n");
    }
}

```

```

Current ID :105  Obj.ID : 105
After 105 : -----> [105]

Current ID :102  Obj.ID : 105
After 102 : -----> [102, 105]

Current ID :104  Obj.ID : 105
Current ID :104  Obj.ID : 102
After 104 : -----> [102, 104, 105]

Current ID :101  Obj.ID : 104
Current ID :101  Obj.ID : 102
After 101 : -----> [101, 102, 104, 105]

Current ID :103  Obj.ID : 104
Current ID :103  Obj.ID : 102
After 103 : -----> [101, 102, 103, 104, 105]

Current ID :100  Obj.ID : 104
Current ID :100  Obj.ID : 102
Current ID :100  Obj.ID : 101
After 100 : -----> [100, 101, 102, 103, 104, 105]

After : [100, 101, 102, 103, 104, 105]

```

Remember, here we can't pass comparable Object to TreeSet(), like comparable.

```

public class Employee implements Comparable<Employee> {
    private int id;
}

```



```

private String name;
private double salary;
//Setters/getters
public Employee(int id, String name, double salary) {
    super();
    this.id = id;
    this.name = name;
    this.salary = salary;
}
@Override
public int compareTo(Employee o) {
    if (this.id < o.id) {
        return -1;
    } else if (this.id > o.id) {
        return 1;
    } else {
        return 0;
    }
}
@Override
public String toString() {
    return "Employee [id=" + id + ", name=" + name + ", salary=" + salary + "];"
}
public static void main(String[] args) {
    List<Employee> employees = new ArrayList<Employee>();
    employees.add(new Employee(105, "Satya", 3000));
    employees.add(new Employee(102, "RAJ", 2000));
    employees.add(new Employee(104, "Madhu", 5000));
    employees.add(new Employee(101, "Srini", 1000));
    employees.add(new Employee(103, "Vinod", 4000));

    System.out.println("Before : " + employees);
    //Until here no sorting will be performed

    Collections.sort(employees);
    //Collection.sort(Comparable) method will intern call CompareTo,
    & it will compare each element with other & Sort the elements

    System.out.println("After : " + employees);
}
}

```

```

Before : [Employee [id=105, name=Satya, salary=3000.0], Employee [id=102, name=RAJ, salary=2000.0],
Employee [id=104, name=Madhu, salary=5000.0], Employee [id=101, name=Srini, salary=1000.0], Employee
[id=103, name=Vinod, salary=4000.0]]

```

```

After : [Employee [id=101, name=Srini, salary=1000.0], Employee [id=102, name=RAJ, salary=2000.0],
Employee [id=103, name=Vinod, salary=4000.0], Employee [id=104, name=Madhu, salary=5000.0], Employee
[id=105, name=Satya, salary=3000.0]]

```

In above we sorted Employees only on their ID type, but if we want to sort by Name & Salary at a time it won't possible. It accepts only one variable comparison at a time.

If we want to sort by Id, Name & Salary at a time, we can use Comparator interface.

## Comparator Interface

Comparator present in `java.util` package & it defines two methods `compare(ob1, ob2)` & `equals(ob1)`

```

public int compare(Object ob1, Object ob2);

public boolean equals(Object ob)

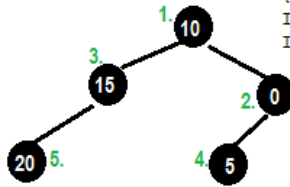
```

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**

```
TreeSet t= new TreeSet(new MyComparator()); (Line 1)
```

1. t.add( 10 ) => 10 ✓
2. t.add( 0 ) => compare(10,0)
3. t.add( 15 ) => compare(15,0)
4. t.add( 5 ) => compare(5,15)
5. t.add( 20 ) => compare(20,5)
6. t.add( 20 ) => ✓



```
class MyComparator implements Comparator
{
public int compare(Object o1, Object o2)
{
Integer i1 = (Integer) o1;
Integer i2 = (Integer) o2;
if (i1 < i2) {
return +1;
} else if (i1 > i2) {
return -1;
} else {
return 0;
}
}
}
```

~~X~~ [0,5,10,15,20] S.o.print(t) [20,15,10,5,0]

```
public class MyComparator implements Comparator {
@Override
public int compare(Object oldObj, Object newObj) {
System.out.println("newObj: " + newObj + ", oldObj: " + oldObj);

Integer i1 = (Integer) oldObj;
Integer i2 = (Integer) newObj;

if (i1 < i2) {
return +1;
} else if (i1 > i2) {
return -1;
} else {
return 0;
}
}

public static void main(String[] args) {
//TreeSet t = new TreeSet();// Line-1
TreeSet t = new TreeSet(new MyComparator()); // Line-2

t.add(50);
System.out.println("After 50: -----> " + t + "\n");

t.add(40);
System.out.println("After 40: -----> " + t + "\n");

t.add(10);
System.out.println("After 10: -----> " + t + "\n");

t.add(30);
System.out.println("After 30: -----> " + t + "\n");

t.add(20);
System.out.println("After 20: -----> " + t + "\n");

//See here we are adding 1, which is less than all exiting elements.
//So, it will compare with almost all elements using compare() method
t.add(1);
System.out.println("After 1: -----> " + t + "\n");
System.out.println(t);
}
}
```

```
newObj: 50, oldObj: 50
After 50: -----> [50]
```

```
newObj: 50, oldObj: 40
After 40: -----> [50, 40]
```

```
newObj: 50, oldObj: 10
newObj: 40, oldObj: 10
After 10: -----> [50, 40, 10]
newObj: 40, oldObj: 30
newObj: 10, oldObj: 30
```

```

After 30: -----> [50, 40, 30, 10]

newObj: 40, oldObj: 20
newObj: 10, oldObj: 20
newObj: 30, oldObj: 20
After 20: -----> [50, 40, 30, 20, 10]

newObj: 40, oldObj: 1
newObj: 20, oldObj: 1
newObj: 10, oldObj: 1
After 1: -----> [50, 40, 30, 20, 10, 1]

[50, 40, 30, 20, 10, 1]

```

[50, 40, 30, 20, 10, 1] 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 **Line2**, 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 TreeseStringComp {
    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]

```

- EmpName implements Comparator for NAME Sorting
- EmpSalary implements Comparator for SALARY Sorting
- Comparable for ID Sorting for Employee Class

### 1. EmpName implements Comparator for NAME Sorting

```

class EmpName implements Comparator<Employee> {
    public int compare(Employee o1, Employee o2) {
        return o1.getName().compareTo(o2.getName());
    }
}

```

### 2. EmpSalary implements Comparator for SALARY Sorting

```

class EmpSalary implements Comparator<Employee> {
    public int compare(Employee o1, Employee o2) {
        if (o1.getSalary() < o2.getSalary()) {
            return -1;
        } else if (o1.getSalary() > o2.getSalary()) {
            return 1;
        }
        return 0;
    }
}

```

### 3. Comparable for ID Sorting for Employee Class

```

public class Employee implements Comparable<Employee> {
    private int id;
    private String name;
    private double salary;
    //Setters & Getters

    public Employee(int id, String name, double salary) {
        super();
        this.id = id;
        this.name = name;
        this.salary = salary;
    }

    public int compareTo(Employee o) {
        if (this.id < o.id) {
            return -1;
        } else if (this.id > o.id) {
            return 1;
        } else {
            return 0;
        }
    }

    public String toString() {
        return "Employee [id=" + id + ", name=" + name + ", salary=" + salary + "];"
    }

    public static void main(String[] args) {

```

```

List<Employee> employees = new ArrayList<Employee>();
employees.add(new Employee(105, "AAA", 3000));
employees.add(new Employee(102, "ZZZ", 2000));
employees.add(new Employee(104, "BBB", 5000));
employees.add(new Employee(101, "DDD", 1000));
employees.add(new Employee(103, "CCC", 4000));

System.out.println("Before : " + employees);
Collections.sort(employees);
System.out.println("ByID :\n " + employees);

//Now we can Sort our Employees based on Multiple Sorting(EmpName, EmpSaltry)
//sort method accepts Comparator: Collections.sort(<list>, Comparator)
Collections.sort(employees, new EmpName());
System.out.println("EmpName : \n "+employees);

Collections.sort(employees, new EmpSalary());
System.out.println("EmpSalary : \n "+employees);
}
}

```

```

Before : [Employee [id=105, name=AAA, salary=3000.0], Employee [id=102, name=ZZZ, salary=2000.0], Employee [id=104, name=BBB, salary=5000.0], Employee [id=101, name=DDD, salary=1000.0], Employee [id=103, name=CCC, salary=4000.0]]
ByID :
[Employee[id=101, name=DDD, salary=1000.0], Employee [id=102, name=ZZZ, salary=2000.0], Employee [id=103, name=CCC, salary=4000.0], Employee [id=104, name=BBB, salary=5000.0], Employee [id=105, name=AAA, salary=3000.0]]
EmpName :
[Employee[id=105, name=AAA, salary=3000.0], Employee [id=104, name=BBB, salary=5000.0], Employee [id=103, name=CCC, salary=4000.0], Employee [id=101, name=DDD, salary=1000.0], Employee [id=102, name=ZZZ, salary=2000.0]]
EmpSalary :
[Employee[id=101, name=DDD, salary=1000.0], Employee [id=102, name=ZZZ, salary=2000.0], Employee [id=105, name=AAA, salary=3000.0], Employee [id=103, name=CCC, salary=4000.0], Employee [id=104, name=BBB, salary=5000.0]]

```

- **Comparable** interface can be used to provide **single way of sorting** whereas **Comparator** interface is used to provide **different ways of sorting**.
- For using Comparable, Class needs to implement it whereas for using Comparator we don't need to make any change in the class, **we can implement it in outside**.
- **Comparable** interface is in **java.lang** package , but **Comparator** interface is present in **java.util**.
- We don't need to make any code changes at client side while using Comparable. For Comparator, client needs to provide the Comparator class to use in compare() method.
- **Arrays.sort()** or **Collection.sort()** methods implemented using compareTo() method of the class.

Comparable	Comparator
1.it is meant for <b>Defalut Nartural Sorting order</b>	1.it is meant for <b>Customized Nartural Sorting order</b>
2.present in <b>java.lang</b> package	2.present in <b>java.util</b> package
3.it defines only one method compareTo()	3.it defines 2 methods compare() & equals()
4. <b>String &amp; all Wrapper classes</b> implments Comparable interface	4. only 2 GUI classes Collator&RulebasedCollator implemets Comparator



Let's say that we need to create a priority queue of String elements in which the String with the smallest *length* is processed first. We can create such a priority queue by passing a custom Comparator that compares two Strings by their length

Since **PriorityQueue** needs to compare its elements and order them accordingly, the user defined class must implement the **Comparable** interface, or you must provide a **Comparator** while creating the priority queue. **Otherwise, PriorityQueue will throw a ClassCastException when you add new objects to it.**

### Constructors of PriorityQueue class

- **PriorityQueue()**: Creates with the default capacity (11) & natural ordering.
- **PriorityQueue(int initialCapacity)**: specified initial capacity & natural ordering.
- **PriorityQueue(int initialCapacity, Comparator comparator)**
- **PriorityQueue(PriorityQueue c)**: Creates a PriorityQueue containing the elements in the specified priority queue.
- **PriorityQueue(SortedSet c)**: Creates a PriorityQueue containing the elements in the specified sorted set.

```
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
```

### Internal implementation

Priority queue represented as a balanced binary heap: the two children of `queue[n]` are `queue[2*n+1]` and `queue[2*(n+1)]`. The priority queue is ordered by comparator, or by the elements' natural ordering.

if comparator is null: For each node `n` in the heap and each descendant `d` of `n`,  $n \leq d$ . The element with the lowest value is in `queue[0]`, assuming the queue is nonempty.

```
private static final int DEFAULT_INITIAL_CAPACITY = 11;
transient Object[] queue;
```

If array Reaches maximum capacity

```
queue = Arrays.copyOf(queue, newCapacity);
```

Establishes the heap invariant (described above) in the entire tree, assuming nothing about the order of the elements prior to the call.

```
@SuppressWarnings("unchecked")
private void heapify() {
    for (int i = (size >>> 1) - 1; i >= 0; i--)
        siftDown(i, (E) queue[i]);
}
```

```
Object[] queue;
PriorityQueue<E> (id=19)
  comparator null
  modCount 5
  queue Object[11] (id=27)
    [0] "A" (id=29)
    [1] "D" (id=34)
    [2] "M" (id=33)
    [3] "Z" (id=32)
    [4] "Y" (id=35)
    [5] null
    [6] null
    [7] null
    [8] null
    [9] null
    [10] null
  size 5
```

```
public static void main(String[] args) {
    PriorityQueue queue = new PriorityQueue<>();
    queue.add("A");
    queue.add("Z");
    queue.add("M");
    queue.add("D");
    queue.add("Y");
    System.out.println(queue);
}
```

## Deque (Double Ended Queue)

The Deque is related to the double-ended queue that supports addition or removal of elements from either end of the data structure, it can be used as a **queue (first-in-first-out/FIFO)** or as a **stack (last-in-first-out/LIFO)**.

Operations	First Element or Head		Last Element or Tail	
	Throws exception	Special Value	Throws exception	Special Value
Insert	addFirst(element)	offerFirst(element)	addLast(element)	offerLast(element)
Remove	removeFirst()	pollFirst()	removeLast()	pollLast()
Examine	getFirst()	peekFirst()	getLast()	peekLast()

### ArrayDeque – Internal implementation

The ArrayDeque class provides the facility of using deque and resizable-array. It inherits AbstractCollection class and implements the Deque interface.

The important points about ArrayDeque class are:

- Unlike Queue, we can add or remove elements from both sides.
- Null elements are not allowed in the ArrayDeque.
- ArrayDeque is not thread safe, in the absence of external synchronization.
- ArrayDeque has no capacity restrictions.
- ArrayDeque is faster than LinkedList and Stack.

```
ArrayDeque<String> queue = new ArrayDeque<>();
queue.add("A");
queue.add("Z");
queue.add("M");
queue.add("D");
queue.add("Y");
System.out.println(queue);
```

Name	Value
args	String[0] (id=16)
queue	ArrayDeque<E> (id=19)
elements	Object[16] (id=28)
[0]	"A" (id=30)
[1]	"Z" (id=31)
[2]	"M" (id=34)
[3]	"D" (id=35)
[4]	null
[5]	null
[6]	null
[7]	null
[8]	null
[9]	null
[10]	null
[11]	null
[12]	null
[13]	null
[14]	null
[15]	null
head	0
tail	4

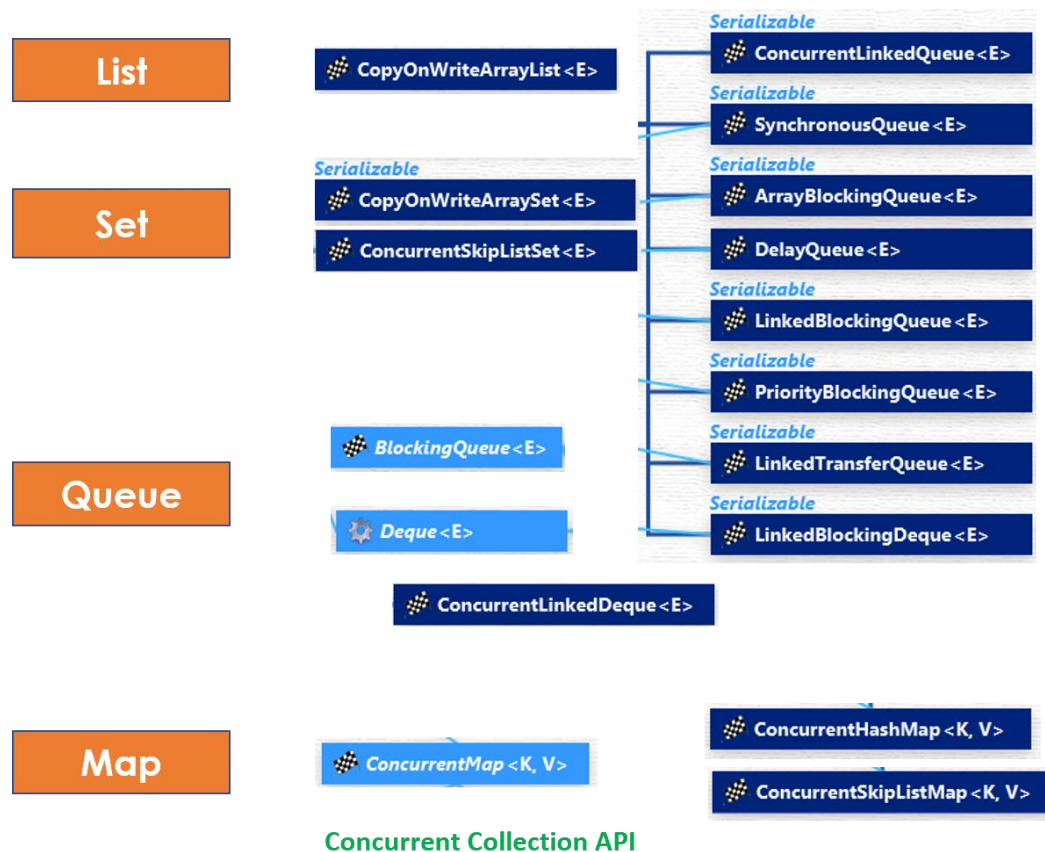
```

graph TD
    Iterable -- EXTENDS --> Collection
    Collection -- EXTENDS --> Queue
    Queue -- EXTENDS --> Deque
    Deque -- IMPLEMENTS --> AbstractCollection
    AbstractCollection -- EXTENDS --> ArrayDeque
    subgraph Class
    AbstractCollection
    ArrayDeque
    end

```



# Concurrent Collections



## Concurrent Collection API

### List – CopyOnWriteArrayList

- **CopyOnWriteArrayList** is a thread-safe variant of `java.util.ArrayList`. All mutative operations (**add**, **set**, and **so on**) are implemented by **making a fresh copy of the underlying array**.
- As the name indicates, `CopyOnWriteArrayList` creates a Cloned copy of underlying ArrayList. For every update operation at certain point both will synchronize automatically which is takes care by JVM. So, there is no effect for threads which are performing read operation.
- It is costly to use because for every update operation a cloned copy will be created. Hence CopyOnWriteArrayList is the best choice for **frequent read** operation, worst choice for **frequent write** operation.
- The underlined data structure is **growable array**.
- **It is thread-safe version of ArrayList.**

### Methods Summary:

```
public boolean addIfAbsent(Object obj)
public int addAllAbsent(Collection C)
```

## Differences between ArrayList & CopyOnWriteArrayList

**1. Synchronization:** ArrayList is not synchronized. CopyOnWriteArrayList is synchronized.

### 2. Iterator

- Iterator of **CopyOnWriteArrayList** is **fail-safe** & doesn't throw **ConcurrentModificationException** even if underlying CopyOnWriteArrayList is modified once Iteration begins.
- Because Iterator is operating on a separate copy of ArrayList. Consequently, all the updates made on CopyOnWriteArrayList is not available to Iterator

### 3. Remove Operation

Iterator of CopyOnWriteArrayList doesn't support remove operation. But, Iterator of ArrayList supports **remove()** operation.

**4. Performance:** ArrayList is faster as it is not synchronized. That means many threads can execute the same piece of code simultaneously. In comparison, CopyOnWriteArrayList is slower.

**5. Added in java version:** ArrayList class was added in java version 1.2, while CopyOnWriteArrayList class was added in java version 1.5 (or java 5).

**6. Package:** ArrayList class is present in **java.util** package, while CopyOnWriteArrayList class is present in **java.util.concurrent** package

```
public class CopyOnWriteArrayListExample {
    public static void main(String args[]) {
        CopyOnWriteArrayList<String> list = new CopyOnWriteArrayList<String>();
        list.add("Java");
        list.add("J2EE");
        list.add("Collection");

        //add, remove operator is not supported by CopyOnWriteArrayList iterator
        Iterator<String> itr = list.iterator();
        while(itr.hasNext()){
            System.out.printf("Read from CopyOnWriteArrayList : %s %n", itr.next());
            // itr.remove(); //not supported in CopyOnWriteArrayList in Java
            list.add("New");
        }
        System.out.println(list);
    }
}
```

## Set – CopyOnWriteArraySet, ConcurrentSkipListSet

### CopyOnWriteArraySet

- **CopyOnWriteArraySet** is a Set that uses an internal CopyOnWriteArrayList for all its operations.
- **CopyOnWriteArraySet** is backed by CopyOnWriteArrayList, which means it also share all basic properties of CopyOnWriteArrayList.
- Iterators of **CopyOnWriteArraySet** class doesn't support remove() operation, trying to remove an element while iterating will result in **UnsupportedOperationException**.
- Only difference between **CopyOnWriteArrayList** & **CopyOnWriteArraySet** are one is List and other is Set. But that brings all difference between Set and List in Java. For example, List is ordered, allows duplicate while Set is unordered, but doesn't allow duplicate.

## ConcurrentSkipListSet

- `ConcurrentSkipListSet` maintains the behavior same as **TreeSet**.
- Since `ConcurrentSkipListSet` implements **NavigableSet** in Java, it is a sorted set just like `TreeSet` with *added feature of being concurrent*. Which essentially means it is a sorted data structure which can be used by multiple threads whereas `TreeSet` is not thread safe.
- The elements of the `ConcurrentSkipListSet` are kept sorted according to their **natural ordering**, or by a Comparator provided at set creation time, depending on which constructor is used.
- `ConcurrentSkipListSet` provides a constructor that takes a `comparator` as a parameter.
- **`ConcurrentSkipListSet(Comparator<? super E> comparator)`** - Constructs a new, empty set that orders its elements according to the specified comparator.
- **`ConcurrentSkipListSet`** implementation provides expected average  $\log(n)$  time cost for the **contains**, **add**, and **remove** operations and their variants. Insertion, removal, and access operations safely execute concurrently by multiple threads.

## Map – ConcurrentMap → ConcurrentHashMap, ConcurrentSkipListMap

### ConcurrentMap:

`ConcurrentMap` is an interface, which is introduced in JDK 1.5 represents a `Map` which is capable of handling concurrent access

It extends map interface in Java. Below are specific methods of `ConcurrentMap` interface:

- **`Object putIfAbsent(K key, V value)`**: If the specified key is not already associated with a value, associate it with the given value.
- **`boolean remove(Object key, Object value)`**: Removes the entry for a key only if currently mapped to a given value.
- **`boolean replace(K key, V oldValue, V newValue)`**: Replaces the entry for a key only if currently mapped to a given value

### ConcurrentHashMap

1. `ConcurrentHashMap` only locks a portion of the collection on update.
2. `ConcurrentHashMap` is better than `Hashtable` and `synchronized Map`.
3. `ConcurrentHashMap` is failsafe does not throw **`ConcurrentModificationException`**.
4. `null` is not allowed as a key or value in `ConcurrentHashMap`.
5. Level of concurrency can be chosen by the programmer on a `ConcurrentHashMap` while initializing it.

```
ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel)
```

By default, `ConcurrentHashMap` allows **16 number of concurrent threads**. We can change this number using the `concurrencyLevel` argument.

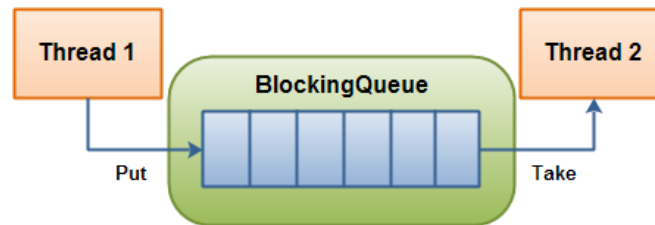
### ConcurrentSkipListMap

- ConcurrentSkipListMap implements ConcurrentNavigableMap, it is a sorted map just like TreeMap (Which also implements NavigableMap interface).
- ConcurrentSkipListMap is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used.
- ConcurrentSkipListMap in Java provides four constructors, out of those 2 relevant ones are -
  - **ConcurrentSkipListMap()** - Constructs a new, empty map, sorted according to the natural ordering of the keys.
  - **ConcurrentSkipListMap(Comparator<? super K> comparator)** - Constructs a new, empty map, sorted according to the specified comparator.
- ConcurrentSkipListMap class in Java implements a concurrent variant of **SkipLists** data structure providing expected average log(n) time cost for the **containsKey**, **get**, **put** and **remove** operations and their variants.
- Insertion, removal, update, and access operations safely execute concurrently by multiple threads.

## Queue – BlockingQueue → PriorityBlockingQueue ,ArrayBlockingQueue,

### BlockingQueue

**BlockingQueue** is used when one thread will produce objects, another thread consumes those Objects.



	Throws Exception	Special Value	Blocks	Times Out
<b>Insert</b>	add(o)	offer(o)	put(o)	offer(o, timeout, timeunit)
<b>Remove</b>	remove(o)	poll()	take()	poll(timeout, timeunit)
<b>Examine</b>	element()	peek()		

- **BlockingQueue** in Java doesn't allow null elements, various implementations like **ArrayBlockingQueue**, **LinkedBlockingQueue** throws NullPointerException when you try to add null on queue

- **Two types of BlockingQueue:**

a. **Bounded queue** – with maximal capacity defined

```
BlockingQueue<String> blockingQueue = new LinkedBlockingDeque<>(10);
```

b. **Unbounded queue** –no maximum capacity, can grow almost indefinitely

```
BlockingQueue<String> blockingQueue = new LinkedBlockingDeque<>();
```

### Producer-Consumer Example

BlockingQueue provides a `put()` method to store the element and `take()` method to retrieve the element. Both are blocking method, which means `put()` will block if the queue has reached its capacity and there is no place to add a new element.

Similarly, `take()` method will block if blocking queue is empty. So, you can see that critical requirement of the producer-consumer pattern is met right there, you don't need to put any thread synchronization code.

```
class Producer extends Thread {
    private BlockingQueue<Integer> sharedQueue;

    public Producer(BlockingQueue<Integer> aQueue) {
        super("PRODUCER");
        this.sharedQueue = aQueue;
    }
    public void run() { // no synchronization needed
        for (int i = 0; i < 10; i++) {
            try {
                System.out.println(getName() + " produced " + i);
                sharedQueue.put(i);
                Thread.sleep(200);
                // if we remove sleep, put will execute 10 times, then take will execute
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

class Consumer extends Thread {
    private BlockingQueue<Integer> sharedQueue;

    public Consumer(BlockingQueue<Integer> aQueue) {
        super("CONSUMER");
        this.sharedQueue = aQueue;
    }
    public void run() {
        try {
            while (true) {
                Integer item = sharedQueue.take();
                System.out.println(getName() + " consumed " + item);
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

public class BlockingQueueDemo {
    public static void main(String[] args) {
        BlockingQueue<Integer> sharedQ = new LinkedBlockingQueue<Integer>();
        Producer p = new Producer(sharedQ);
        Consumer c = new Consumer(sharedQ);
        p.start();
        c.start();
    }
}
```

- **ArrayBlockingQueue** – a blocking queue class based on bounded Java Array. Once instantiated, cannot be resized.
- **PriorityBlockingQueue** – a priority queue-based blocking queue. It is an unbounded concurrent collection.
- **LinkedBlockingQueue** – an optionally bounded Java concurrent collection. It orders elements based on FIFO order.

## Deque - ConcurrentLinkedDeque

ConcurrentLinkedDeque in Java is an **unbounded** thread-safe **Deque** which stores its elements as linked nodes. Since it implements deque interface **ConcurrentLinkedDeque** supports element *insertion and removal at both ends*.

We have methods `addFirst()`, `addLast()`, `getFirst()`, `getLast()`, `removeFirst()`, `removeLast()` to facilitate operations at both ends.

### Usage of ConcurrentLinkedDeque

A ConcurrentLinkedDeque is an appropriate choice when many threads will share access to a common collection as concurrent insertion, removal, and access operations execute safely across multiple threads.

Note that it **doesn't block** operations as done in the implementation of [BlockingDeque](#) interface like [LinkedBlockingDeque](#). So there are no `putFirst()`, `takeFirst()` or `putLast()`, `takeLast()` methods which will wait if required.

## Java.util.Arrays

This class contains various methods for manipulating arrays (such as sorting and searching). This class also contains a static factory that allows arrays to be viewed as lists.

- **Arrays to list**
- **Sorting & Searching**
- **Copying & filling**
- **public static String toString(int[] a)** The string representation consists of a list of the array's elements, enclosed in square brackets ("`[]`").

```
public static void main(String[] args) throws Exception {
    String a[] = {"a", "b", "c"};
    System.out.println("OLD \t :"+a.toString());
    System.out.println("New \t :"+Arrays.toString(a));
}
```

```
OLD : [Ljava.lang.String;@6d06d69c
New : [a, b, c]
```

- **public static List asList(T... a)** - This method returns a **fixed-size list** backed by the specified array. **adding or removing elements from the list aren't allowed** on this created list, you can only read or overwrite the elements
- **public static void sort(int[] a)** – Sorts the specified array into ascending numerical order.
- **public static void sort(int[] a, int fromIndex, int toIndex)** If we wish to sort a specified range of the array into ascending order.
- **public static int binarySearch(int[] a, int key)** Returns an int value for the index of the specified key in the specified array. Returns a negative number if the specified key is not found in the array.
- **public static int[] copyOf(int[] original, int newLength)** Copies the specified array and length. It truncates the array if provided length is smaller and pads if provided.

- **public static int[] copyOfRange(int[] original, int from, int to)** Copies the specified range of the specified array into a new array
- **public static void fill(int[] a, int val)** Fills all elements of the specified array with the specified value.
- **public static void fill(int[] a, int fromIndex, int toIndex, int val)** – Fills elements of the specified array with the specified value from the fromIndex element, but not including the toIndex element.
- **static boolean equals(Object[] a, Object[] a2)** - It will compare the Content of two arrays & must be in same Order

```

public static void main(String[] args) throws Exception {
    String a[] = {"a", "b", "c"};
    String b[] = {"a", "b", "c"};
    String c[] = {"c", "b", "a"};
    System.out.println("OLD : "+a.equals(b));
    System.out.println("New : "+Arrays.equals(a, b));
    System.out.println("Wrong Order : "+Arrays.equals(a, c));
    //It will compare the Content & must be in same Order
}
OLD : false
New : true
Wrong Order : false

```

- **static int hashCode(Object[] a)**- This method returns a hash code based on the contents of the specified array.

## java.util.Collections class

This class consists exclusively of static methods that operate on or return collections. It contains polymorphic algorithms that operate on collections, "wrappers", which return a new collection backed by a specified collection, and a few other odds and ends.

The methods of this class all throw a `NullPointerException` if the collections or class objects provided to them are `null`.

- **Create List/Set/Map from Arrays**
  - **Converting List/Set/Map to Synchronized & unmodifiable**
  - **Sorting & Searching**
  - **Reverse, shuffle, min, max, copy**
- `List<String> list = List.of("foo", "bar", "baz");`
  - `Set<String> set = Set.of("foo", "bar", "baz");`
  - `Map<String, String> map = Map.of("foo", "a", "bar", "b", "baz", "c");`

```

Map<String, String> map = Map.ofEntries(
    new AbstractMap.SimpleEntry<>("foo", "a"),
    new AbstractMap.SimpleEntry<>("bar", "b"),
    new AbstractMap.SimpleEntry<>("baz", "c"));

```

1. **static void sort(List list1):** Sorts the list `list1` into ascending order according to the natural sequence (a before b or 1 before 2) of the elements.

2. **static int binarySearch(List list, Object ob):** Searches the **ob** in the list **list**. Returns the index of the element **obj**. Before applying this method, **the elements must be sorted** earlier with **sort()** method
3. **static Collection synchronizedCollection(Collection col1):** Returns a synchronized version of collection **col1**. It is used for thread-safe operations. We have following specific Methods like
  - **synchronizedList()**
  - **synchronizedSet()**
  - **synchronizedMap()., etc**
4. **static Collection unmodifiableCollection(Collection col1):** make any Collection to unmodifiable (read-only) version of **collection**. Any methods like **add()** or **remove()**, if applied throws **UnsupportedOperationException** We have following specific Methods like
  - **unmodifiableList()**
  - **unmodifiableSet()**
  - **unmodifiableMap()., etc**
4. **static void reverse(List list1):** Existing order of the elements in the list **list1** are reversed.
5. **static void shuffle(List list1):** Shuffles the existing elements of **list1** **randomly**. For repeated execution of the method, elements with different order are obtained.
6. **static void swap(List list1, int index1, int index2):** List **list1** elements at index numbers **index1** and **index2** are swapped.
7. **static void fill(List list1, Object obj1):** Replaces all the elements of **list1** with **obj1**. Earlier elements are lost. This method is used to fill all the elements with the same values.
8. **static void copy(List destination1, List source1):** Copies all the elements of List **source1** into the **destination1** list. It is like [arraycopy\(\)](#) method.
9. **static Object min(Collection col1):** Returns the element with the minimum value
10. **static Object max(Collection col1):** Returns the element with the maximum value
11. **static void rotate(List list1, int dist1):** Rotates the elements in the list **list1** by the specified distance **dist1**.
12. **static boolean replaceAll(List list1, Object oldObj, Object newObj):** Replaces the old element **oldObj** with the new element **newObj** in the list **list1** (all the occurrences). Returns true when the operation is successful (when the **oldObj** exists).
13. **static int frequency(Collection col1, Object obj1):** Checks how many times **obj1** exists in collection **col1**, returns as an integer value.
14. **static boolean disjoint(Collection col1, Collection col2):** Returns true if the collection classes **col1** and **col2** do not have any common elements. Introduced with [JDK 1.5](#).
15. **static boolean addAll(Collection col1, Object obj1):** Here, **obj1** can be a single element or an array. Adds **obj1** to the collection **col1**.



## 12. Stream API

Lambda expressions are new in Java 8. **Java lambda expressions are Java's first step into functional programming.**

Java lambda expressions are commonly used to implement simple event listeners / callbacks, or in functional programming with the [Java Streams API](#)

### Functional Interfaces

```
Runnable r = new Runnable(){
    @Override
    public void run() {
        System.out.println("My Runnable");
    }
};
```

If you look at the above code, the actual part of the code is inside `run()` method. Rest of the code is the way java programs are structured. This is a type of **boiler-plate** code.

**Java 8 Functional Interfaces and Lambda Expressions** help us in writing smaller and cleaner code by removing a lot of **boiler-plate** code. Functional Interfaces has following Characteristics.

- **An interface with exactly one abstract method is called Functional Interface.**
- Using **@FunctionalInterface** annotation, we can mark an interface as functional interface.
- If an interface is annotated with **@FunctionalInterface** annotation and we try to have more than one abstract method, **it throws compiler error.**
- The major benefit of java 8 functional interfaces is that we can use **lambda expressions** to instantiate them and avoid using bulky anonymous class implementation.

Java 8 has defined a lot of functional interfaces in **java.util.function** package. Some of the useful functional interfaces are **Predicate, Function, Consumer, Supplier.**

### java.util.function Package:

**java.util.function** package in Java 8 contains many built-in functional interfaces like-

**1.Predicate(test):** The Predicate interface has an abstract method **test()** which gives a **boolean** value as a result, by taking input argument type **T**. Its prototype is

```
public Predicate
{
    public boolean test(T t);
}
```

**2.Function:** The Function interface has an abstract method **apply()** which takes argument of **one type T** and returns a result of **another type R**. Its prototype is

```
public interface Function
{
    public R apply(T t);
}
```

**3.Consumer Interface:** It accepts an input and returns no result.

```
@FunctionalInterface
public interface Consumer {
    void accept(T t);
}
```

And it contains default method **andThen**(Consumer<? super T> after)

**4.Supplier Interface:** In some scenarios we have **no input** but expected to **return an output**. Those situations **Supplier<T>** can be used without the need to define a new functional interface every time.

```
@FunctionalInterface
public interface Supplier<T> {
    T get();
}
```

**5.BinaryOperator:** The BinaryOperator interface has an abstract method **apply** which takes two arguments and returns **a result of same type**. Its prototype is

```
public interface BinaryOperator
{
    public T apply(T x, T y);
}
```

## Lambdas

Lambda's are used to provide the **implementation of Functional interface, Less coding**.

Lambda Expressions syntax is **(argument) ->(body)**. Now let's see how we can write above anonymous Runnable using lambda expression.

```
Runnable r1 = () -> System.out.println("My Runnable");
```

- The body of a lambda expression can contain zero, one or more statements.
- When there is a single statement curly brackets are Optional, and the return type of the anonymous function is the same as that of the body expression.
- When there are more than one statements, then these must be enclosed in {curly brackets} and the return type of the anonymous function is the same as the type of the value returned within the code block, or void if nothing is returned.

### 1.Zero parameter

```
() -> System.out.println("Zero parameter lambda");
```

### 2.One parameter

```
(p) -> System.out.println("One parameter: " + p);
```

### 3.Multiple parameters

```
(p1, p2) -> System.out.println("Multiple parameters: " + p1 + ", " + p2);
```

## Java Lambda Expression Example: ForEach Loop

Java provides a new method `forEach()` to iterate the elements. It is defined in `Iterable` and `Stream` interface. **It is a default method defined in the `Iterable` interface.**

```
public interface Iterable<T>
{
    Iterator<T>    iterator();
    Spliterator<T> spliterator();
    void          forEach(Consumer<? super T> action);
}
```

Collection classes which extends `Iterable` interface can use `forEach` loop to iterate elements.

```
default void forEach(Consumer<super T>action)
```

*"performs the given action for each element of the `Iterable` until all elements have been processed or the action throws an exception."*

```
public class LambdaExpressionExample7{
    public static void main(String[] args) {
        List<String> list=new ArrayList<String>();
        list.add("ankit");
        list.add("mayank");
        list.add("irfan");
        list.add("jai");
        list.forEach(
            (n)->System.out.println(n)
        );
    }
}
```

This code will print every element of the list. You can even replace lambda expression with [method reference](#), because we are passing the lambda parameter as it is - to `System.out.println()` method as

```
list.forEach(System.out::println);
```

If both Passing Parameter & Printing parameter is same we can use `System.out::println`

- **`forEach()` is a terminal operation**, which means once calling `forEach()` method on stream, **you cannot call another method**. It will result in a runtime exception.
- When you call `forEach()` on parallel stream, the **order of iteration is not guaranteed**, but you can ensure that ordering by calling `forEachOrdered()` method.
- There are two `forEach()` method in Java 8, one defined inside `Iterable` and other inside `java.util.stream.Stream` class.
  - If purpose of `forEach()` is just iteration then you can directly call it e.g. `list.forEach()` or `set.forEach()`
  - if you want to perform some operations e.g. `filter` or `map`, then better first get the stream and then perform that operation and finally call `forEach()` method.

## Java Lambda Expression Example: Comparator

```
Collections.sort(list, (p1,p2)->{
    return p1.name.compareTo(p2.name);
});

for(Product p:list){
    System.out.println(p.id+" "+p.name+" "+p.price);
}
```

## Streams

All of us have watched online videos on **Youtube**. When we start watching a video, a small portion of the video file is first loaded into our computer and starts playing. we don't need to download the complete video before we start watching it. This is called **video streaming**.

At a very high level, we can think of that small portion of the video file as a **Stream** and the whole video as a **Collection**

In Java, `java.util.Stream` interface represents a stream on which one or more operations can be performed. **Stream operations are either intermediate or terminal.**

- **Intermediate** operations return a **stream**. so we can chain multiple intermediate operations without using semicolons.
- **Terminal** operations are either void or return a non-stream result

Streams are created on a source, e.g. a `java.util.Collection` like **List** or **Set**. The **Map is not supported** directly, we can create stream of map keys, values or entries. Stream operations can either be executed **sequentially** or **parallel**. when performed parallelly, it is called a **parallel stream**.

## How Stream Work

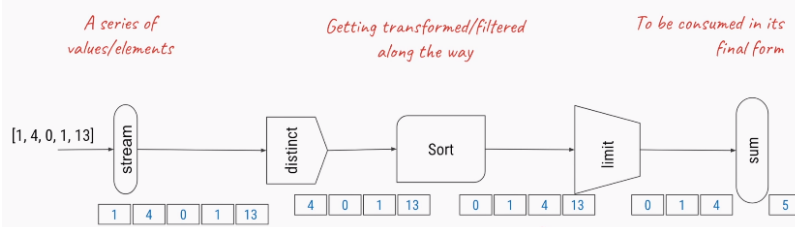
3 distinct parts

- Create
- Process
- Consume

```
int[] numbers = {4, 1, 13, 90, 16, 2, 0};
```

```
IntStream.of(numbers)
    .distinct()
    .sorted()
    .limit(3)
    .forEach(System.out::println);
```

1. Create the stream  
2. Process the stream  
3. Consume the stream



```
List<String> myList = Arrays.asList("a1", "a2", "b1", "c2", "c1");
myList.stream()
    .filter(s -> s.startsWith("c"))
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);
```

In above `filter`, `map` and `sorted` are intermediate operations whereas `forEach` is a terminal operation. Streams can be created from various data sources, especially collections. Lists and Sets support new methods `stream()` and `parallelStream()` to either create a sequential or a parallel stream. **Parallel streams can operate on multiple threads.**

## The features of Java stream are

- A stream is not a data structure.
- it takes input from the **Collections, Arrays** or **I/O** channels.
- **Streams don't change original data structure**; they only provide the result as per the pipelined methods.
- Do not support indexed access
- Lazy access supported
- Parallelizable
- Each intermediate operation is lazily executed and returns a stream as a result. hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.

## 1. Creating Streams

**XxxStream.of()** : Create Steam using **Primitive types (int, char, byte, double)**

```
IntStream is = IntStream.of(3, 4, 5, 6);
DoubleStream is = DoubleStream.of(3.1, 4.0, 5.7, 6.0);
```

**Stream.of()** : Create Steam using **Primitive Object types (Int, String, Wrapper types)**

```
//Primitive Types
Stream<Integer> intStream= Stream.of(1,2,3,4,5,6,7,8);

Stream<Character> charStream = Stream.of('A', 'B', 'C', 'D', 'E');

Stream<String> strStream = Stream.of("Aaa", "Bbbb", "Cccc", "Dddd");
```

[IntStream](#) is a stream of *primitive* int values.

[Stream<Integer>](#) is a stream of Integer *objects*

**Stream.of(array)** : Create Steam using **Array types (Int, String)**

```
Stream<Integer> intArraystream = Stream.of( new Integer[]{1,2,3,4,5,6,7,8,9} );

Stream<Character> charArraystream = Stream.of( new Character[]{'A', 'B', 'C', 'D', 'E'} );

Stream<String> strArraystream = Stream.of( new String[]{"Aaa", "Bbbb", "Cccc", "Dddd"} );
```

**List.stream()** : Create Steam from **Collection types (List, Set, Not MAP)**

```
//Collection types
List<String> list = new ArrayList<String>();
list.add("a");
list.add("b");
list.add("c");
Stream<String> strListStream = list.stream();

List<Integer> list2 = new ArrayList<Integer>();
list2.add(1);
list2.add(2);
list2.add(3);
Stream<Integer> intListStream = list2.stream();
```

### Ex: Create Stream by Splitting String

```
Stream<String> splitStream = Stream.of("A$B$C$D".split("\\$"));
```

## 2. Stream Operations

Stream Operations: Intermediate	
<code>filter(Predicate&lt;T&gt;)</code>	The elements of the stream matching the predicate (Condition)
<code>map(Function&lt;T, U&gt;)</code>	Performs some operation on each element & return something (add, multiply, convert)
<code>flatMap(Function&lt;T, Stream&lt;U&gt;&gt;)</code>	The elements of the streams resulting from applying the provided stream-bearing function to the elements of the stream
<code>distinct()</code>	The elements of the stream, with duplicates removed
<code>sorted()</code>	The elements of the stream, sorted in natural order
<code>Sorted(Comparator&lt;T&gt;)</code>	The elements of the stream, sorted by the provided comparator
<code>limit(long)</code>	The elements of the stream, truncated to the provided length
<code>skip(long)</code>	The elements of the stream, discarding the first N elements
<code>takeWhile(Predicate&lt;T&gt;)</code>	(The elements of the stream, truncated at the first element for which the provided predicate is not true
<code>dropWhile(Predicate&lt;T&gt;)</code>	(Java 9 only) The elements of the stream, discarding the initial segment of elements for which the provided predicate is true

Terminal Operations	
<code>forEach(Consumer&lt;T&gt; action)</code>	Apply the provided action to each element of the stream.
<code>toArray()</code>	Create an array from the elements of the stream.
<code>reduce(...)</code>	Aggregate the elements of the stream into a summary value.
<code>collect(...)</code>	Aggregate the elements of the stream into a summary result container.
<code>min(Comparator&lt;T&gt;)</code>	Return the minimal element of the stream according to the comparator.
<code>max(Comparator&lt;T&gt;)</code>	Return the maximal element of the stream according to the comparator.
<code>count()</code>	Return the size of the stream.
<code>{any,all,none}Match(Predicate&lt;T&gt;)</code>	Return any/all/none of the elements of the stream match the predicate.
<code>findFirst()</code>	Return the first element of the stream, if present.
<code>findAny()</code>	Return any element of the stream, if present.

## Collectors & collect method

```
java.util.stream
|
Interface Collector
|
class Collectors
```

**Collectors class** : Implementations of `Collector` that implement various useful reduction operations, such as gathering elements into collections, summarizing elements according to various criteria, etc.

Some of Useful methods:

To Collections	Math Operations	Map Grouping
<code>toCollection(Supplier)</code>	<code>counting()</code>	<code>groupingBy(Function)</code>
<code>toList()</code>	<code>minBy(Comparator)</code>	<code>groupingByConcurrent(Function)</code>
<code>toSet()</code>	<code>maxBy(Comparator)</code>	
<code>toMap(Function, Function)</code>	<code>summingInt(ToIntFunction),</code>	<code>partitioningBy(Predicate)</code>

joining() mapping(Function, Collector) filtering(Predct, Collector)	summingLong(ToLongFunction), averagingInt(ToIntFunction), averagingLong(ToLongFunction)	reducing(BinaryOperator)
---	---	--------------------------

### Finally, most commonly used Operations

Following are Country names as List – common for all below Operations

```
List<String> list = new ArrayList<String>();
list.add("America");
list.add("China");
list.add("Japan");
list.add("Germany");
list.add("India");
list.add("Italy");
list.add("Russia");
list.add("Sweden");
list.add("Ukraine");
list.add("India");
list.add("Italy");
```

#### Intermediate Operations:

#### 1. Stream.forEach(consumer):

Printing list using foreach

```
list.stream().forEach(System.out::println);
```

#### 2. Stream.filter()

The `filter()` method accepts a `Predicate` to filter all elements of the stream. This operation is intermediate which enables us to call another stream operation (e.g. `forEach()`) on the result.

```
System.out.println("\n \n Print Countries name Start with I");
list.stream()
    .filter((n)->n.startsWith("I"))
    .forEach(System.out::println);
Print Countries name Start with I
India
Italy
India
Italy
```

#### 3. Stream.map()

The `map()` intermediate operation converts each element in the stream into another object via the given function. The following example converts each string into an UPPERCASE string. But we can use `map()` to transform an object into another type as well.

```
System.out.println("\n \n Convert to Upper case");
list.stream()
    .filter((n) -> n.startsWith("I"))
    .map((n) -> n.toUpperCase())
    .map((String::toUpperCase)) //Using reference <RETURN TYPE> :: Operation
    .forEach(System.out::println);
```

## 4. Stream.sorted()

The `sorted()` method is an intermediate operation that returns a sorted view of the stream. The elements in the stream are sorted in natural order unless we pass a custom [Comparator](#).

```
System.out.println("\n \n Convert to Upper case & Sort");
list.stream()
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);

list.stream()
    .map(String::toUpperCase)
    .sorted(Comparator.reverseOrder())
    .forEach(System.out::println);
```

To Sort User Object based on Age

```
List<User> sortedList = users.stream()
    .sorted(Comparator.comparingInt(User::getAge))
    .reversed()
    .collect(Collectors.toList());
sortedList.forEach(System.out::println);
```

To Sort User Object based on Name

```
List<User> sortedList = users.stream()
    .sorted(Comparator.comparing(User::getName))
    .collect(Collectors.toList());
```

## 5. Stream.distinct()

To Remove duplicate items from Stream. [Javadocs](#) say that `distinct()` - Returns a stream consisting of the distinct elements (according to `Object.equals(Object)`) of this stream. In case of object types we need to generate hashCode to made objects equal.

```
System.out.println("\n \n List of Strings Start with I");
List tempList = list.stream()
    .filter(n-> n.startsWith("I"))
    .distinct()
    .collect(Collectors.toList());
System.out.println(tempList);
```

### Terminal Operations:

**forEach:** The `forEach` method is used to iterate through every element of the stream.

```
List number = Arrays.asList(2,3,4,5);
number.stream().map(x->x*x).forEach(y->System.out.println(y));
```

## 5. Stream.collect() :: Collection

The `collect()` method is used to receive elements from a steam and store them in a **collection**.

To Collections	Math Operations	Map Grouping
<code>toCollection(Supplier)</code>	<code>counting()</code>	<code>groupingBy(Function)</code>
<code>toList()</code>	<code>minBy(Comparator)</code>	<code>groupingByConcurrent(Function)</code>
<code>toSet()</code>	<code>maxBy(Comparator)</code>	



toMap(Function, Function)	summingInt(ToIntFunction), summingLong(ToLongFunction),	partitioningBy(Predicate) reducing(BinaryOperator)
joining()		
mapping(Function, Collector)	averagingInt(ToIntFunction), averagingLong(ToLongFunction)	
filtering(Predct, Collector)		

```
System.out.println("\n \n List of Strings Start with I");
List tempList = list.stream()
    .filter((n)-> n.startsWith("I"))
    .collect(Collectors.toList());
System.out.println(tempList);
```

## 6. Stream.match() :: boolean

Various matching operations can be used to check whether a given predicate matches the stream elements. All of these matching operations are terminal and return a boolean result.

```
boolean matchedResult = memberNames.stream()
    .anyMatch((s) -> s.startsWith("A"));
System.out.println(matchedResult);    //true

matchedResult = memberNames.stream()
    .allMatch((s) -> s.startsWith("A"));
System.out.println(matchedResult);    //false

matchedResult = memberNames.stream()
    .noneMatch((s) -> s.startsWith("A"));
System.out.println(matchedResult);    //false
```

## 7. Stream.count() :: Integer/Any Number

The `count()` is a terminal operation returning the number of elements in the stream as a long value.

```
long totalMatched = memberNames.stream()
    .filter((s) -> s.startsWith("A"))
    .count();
System.out.println(totalMatched);    //2
```

## 8. Stream.reduce()

The `reduce()` method performs a reduction on the elements of the stream with the given function. The result is an `Optional` holding the reduced value.

In the given example, we are reducing all the strings by concatenating them using a separator #.

```
Optional<String> reduced = memberNames.stream()
    .reduce((s1,s2) -> s1 + "#" + s2);
reduced.ifPresent(System.out::println);
```

## 9. Stream.anyMatch() :: Boolean

The `anyMatch()` will return true once a condition passed as predicate satisfies. Once a matching value is found, no more elements will be processed in the stream.

In the given example, as soon as a String is found starting with the letter 'A', the stream will end and the result will be returned.

```
boolean matched = memberNames.stream()
    .anyMatch((s) -> s.startsWith("A"));
System.out.println(matched); //true
```

## 10. Stream.findFirst()

The `findFirst()` method will return the first element from the stream and then it will not process any more elements.

```
String firstMatchedName = memberNames.stream()
    .filter((s) -> s.startsWith("L"))
    .findFirst().get();
System.out.println(firstMatchedName); //Lokesh
```

## 11. flatMap()

flattening is referred to as merging multiple collections/arrays into one

```
flatMap() = Flattening + map()
```

`Stream.flatMap()` helps in converting `Stream<Collection<T>>` to `Stream<T>`.

Flattening example 1

```
Before flattening : [[1, 2, 3], [4, 5], [6, 7, 8]]
After flattening : [1, 2, 3, 4, 5, 6, 7, 8]
```

```
List <Integer> list1 = Arrays.asList(1, 2, 3);
List <Integer> list2 = Arrays.asList(4, 5, 6);
List <Integer> list3 = Arrays.asList(7, 8, 9);

List <List <Integer >> listOfLists = Arrays.asList(list1, list2, list3);

List < Integer > listOfAllIntegers = listOfLists.stream()
    .flatMap(x -> x.stream())
    .collect(Collectors.toList());

System.out.println(listOfAllIntegers);
```

## 12. toArray

```
Employee[] employeesArray = employeeList.stream()
    .filter(e -> e.getSalary() < 400)
    .toArray(Employee[]::new);
```

### Example

```
class Product{
    int id;
    String name;
    float price;
    public Product(int id, String name, float price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }
}

public class Streams {
```

```

public static void main(String[] args) {

    List<Product> productsList = new ArrayList<Product>();
    //Adding Products
    productsList.add(new Product(1, "HP Laptop", 25000f));
    productsList.add(new Product(2, "Dell Laptop", 30000f));
    productsList.add(new Product(3, "Lenovo Laptop", 28000f));
    productsList.add(new Product(4, "Sony Laptop", 28000f));
    productsList.add(new Product(5, "Apple Laptop", 90000f));

    List<Float> productPriceList2 =productsList.stream()
        .filter(p -> p.price > 30000)// filtering data
        .map(p->p.price) // fetching price
        .collect(Collectors.toList()); // collecting as list
    System.out.println(productPriceList2); //[90000.0]

    // This is more compact approach for filtering data
    productsList.stream()
        .filter(product -> product.price == 30000)
        .forEach(product -> System.out.println(product.name)); // Dell Laptop

    // count number of products based on the filter
    long count = productsList.stream()
        .filter(product->product.price<30000)
        .count();
    System.out.println(count); //3

    // Converting product List into Set
    Set<Float> productPriceList = productsList.stream()
        .filter(product->product.price < 30000) // filter product on the base of price
        .map(product->product.price) //get the price
        .collect(Collectors.toSet()); // collect it as Set(remove duplicate elements)
    System.out.println(productPriceList); //[25000.0, 28000.0]

}
}

```

### 3. Parallel Streams

- Sequential or Normal streams just work like for-loop using a single core. Parallel streams divide the provided task(like fork& join) into many and run them in different threads, utilizing multiple cores of the computer.
- In parallel execution, if number of tasks are more than available cores at a given time, the remaining tasks are queued waiting for currently running task to finish.
- In above-listed stream examples, anytime we want to do a particular job using multiple threads in parallel cores, all we have to call `parallelStream()` method instead of `stream()` method.

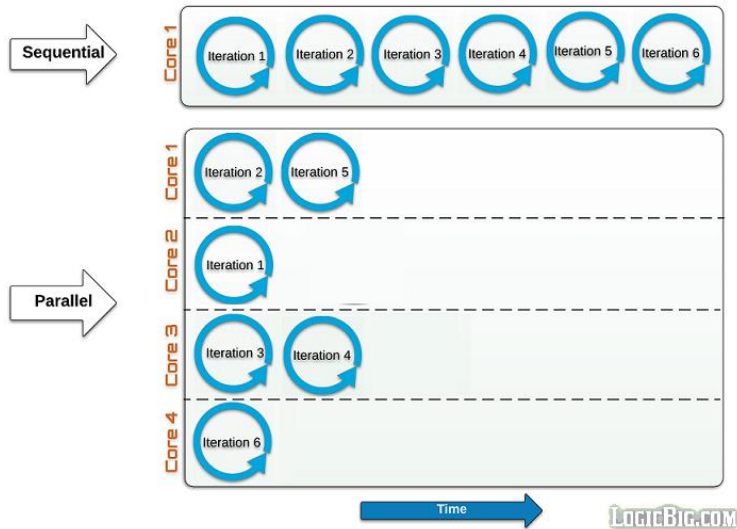
```

//For ArrayTypes
Stream<Integer> parallelStream =Stream.of(nums).parallel();

//For List Types
Stream<String> stream = list.parallelStream();

```

## Sequential vs parallel streams running in 4 cores



```
Integer[] nums = {1,1, 2, 3,3, 4, 5, 6, 7, 8, 9, 10};
System.out.println("\n \n Print Strings using normal Stream");
Stream<Integer> seqStream =Stream.of(nums);
seqStream.distinct()
    .sorted(Comparator.reverseOrder())
    .forEach((n)-> {
        System.out.println(Thread.currentThread().getName()+" : "+n);
    });
```

```
Print Strings using normal Stream
main : 10
main : 9
main : 8
main : 7
main : 6
main : 5
main : 4
main : 3
main : 2
main : 1
```

```
System.out.println("\n \n Print Strings using Parllel Stream");
Stream<Integer> parllelStream =Stream.of(nums).parallel();
parllelStream.distinct()
    .sorted(Comparator.reverseOrder())
    .forEach((n)-> {
        System.out.println(Thread.currentThread().getName()+" : "+n);
    });
```

```
Print Strings using Parllel Stream
main : 4
ForkJoinPool.commonPool-worker-7 : 6
ForkJoinPool.commonPool-worker-5 : 9
ForkJoinPool.commonPool-worker-9 : 2
ForkJoinPool.commonPool-worker-3 : 3
ForkJoinPool.commonPool-worker-15 : 1
ForkJoinPool.commonPool-worker-11 : 8
ForkJoinPool.commonPool-worker-13 : 5
ForkJoinPool.commonPool-worker-5 : 10
ForkJoinPool.commonPool-worker-7 : 7
```

In Above we are using parallelStream, So order is not correct, because the steam is processed by many threads.

## Examples

Creating Streams	Intermediate	Terminal Operations
<a href="#">concat()</a>	<a href="#">filter()</a>	<a href="#">forEach()</a>
<a href="#">empty()</a>	<a href="#">map()</a>	<a href="#">forEachOrdered()</a>
<a href="#">generate()</a>	<a href="#">flatMap()</a>	<a href="#">toArray()</a>
<a href="#">iterate()</a>	<a href="#">distinct()</a>	<a href="#">reduce()</a>
<a href="#">of()</a>	<a href="#">sorted()</a>	<a href="#">collect()</a>
	<a href="#">peek()</a>	<a href="#">min()</a>
	<a href="#">limit()</a>	<a href="#">max()</a>
	<a href="#">skip()</a>	<a href="#">count()</a>
		<a href="#">anyMatch()</a>
		<a href="#">allMatch()</a>
		<a href="#">noneMatch()</a>
		<a href="#">findFirst()</a>
		<a href="#">findAny()</a>

## 1.Count Occurrences of a Char in a String

```
String someString = "elephant";
long count = someString
    .chars()
    .filter(ch -> ch == 'e')
    .count();
```

Using map:

- Invoke the **chars()** method on the input string and which returns the `IntStream` instance. This int stream holds the integer representation of each character in the string.
- Need to convert `IntStream` to `CharStream` using the **mapToObj()** method.
- Last, need to group by characters by calling **Collectors.groupingBy()** and to count call **Collectors.counting()** method.

```
String input = "success";

IntStream intStream = input.chars();
Stream<Character> stream = intStream.mapToObj(ch -> (char) ch);
Map<Character, Long> output = stream.collect(Collectors.groupingBy(ch -> ch, Collectors.counting()));
System.out.println(output);

//Single Line
Map<Character, Long> output2 = input.chars()
    .mapToObj(ch -> (char) ch)
    .collect(Collectors.groupingBy(ch -> ch, Collectors.counting() ) );
System.out.println(output2);
```

## Counting Empty String

```
List<String> strList = Arrays.asList("abc", "", "bcd", "", "defg", "jk");
long c1 = strList.stream()
    .filter(s -> s.isEmpty())
    .count();
System.out.println(c1); //2
```

## Map & joining

```
System.out.println("Count String whose length is more than three");
c1 = strList.stream().filter(s -> s.length()>3).count();
System.out.println(c1);//1

System.out.println("Remove all empty Strings from List");
List l1 = strList.stream()
    .filter(s -> !s.isEmpty())
    .collect(Collectors.toList());

System.out.println("Convert Strings in a list to uppercase and Join them with coma");
String join = strList.stream()
    .map(s -> s.toUpperCase())
    .collect(Collectors.joining(","));
```

## Statistics: Get count, min, max, sum, and the average for numbers

we will learn how to get some statistical data from Collection, e.g. finding the minimum or maximum number from List, calculating the sum of all numbers from a numeric list, or calculating the average of all numbers from List.

Since these statistics operations are numeric in nature, it's essential to call the `mapToInt()` method. After this, we call the `summaryStatistics()`, which returns an instance of an `IntSummaryStatistics`.

It is this object which provides us utility method like `getMin()`, `getMax()`, `getSum()` or `getAverage()`.

```
List<Integer> primes = Arrays.asList(2, 3, 5, 7, 11, 13, 17, 19, 23, 29);
c1 = primes.stream()
    .mapToInt(s -> s)
    .summaryStatistics()
    .getSum();
System.out.println(c1);
```

```
List<Integer> primes1 = Arrays.asList(2, 3, 5, 7, 11, 13, 17, 19, 23, 29);
IntSummaryStatistics stats = primes1.stream().mapToInt((x) -> x).summaryStatistics();
System.out.println("Highest prime number in List : " + stats.getMax());
System.out.println("Lowest prime number in List : " + stats.getMin());
System.out.println("Sum of all prime numbers : " + stats.getSum());
System.out.println("Average of all prime numbers : " + stats.getAverage());
```

## Ref.

<https://howtodoinjava.com/java/stream/java-streams-by-examples/>

<https://www.java67.com/2014/04/java-8-stream-examples-and-tutorial.html>

<https://javabypatel.blogspot.com/2018/06/java-8-stream-practice-problems.html>

## 13. 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.*`
- It has only one class, whose fully qualified name is `java.applet.Applet`

In `java.applet.Applet` we have four life cycle methods.

#### 1. `public void init():`

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

#### 1. `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.

#### 2. `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.

#### 3. `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.

#### 4. `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 to 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";
    }
}
```

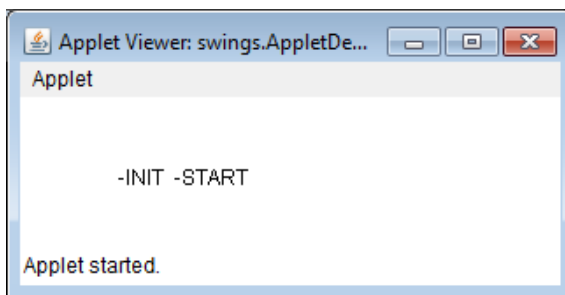
Compile the above Program, Run using any of below methods

### 1) Using HTML program

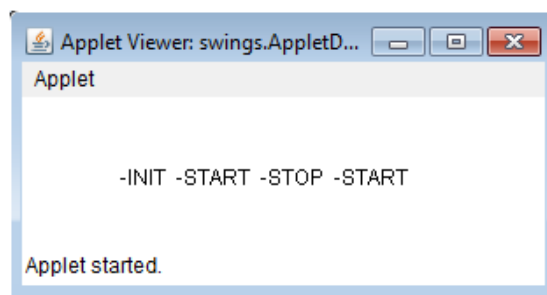
```
<APPLET code="AppletDemo " height=100 width=150>
```

### 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



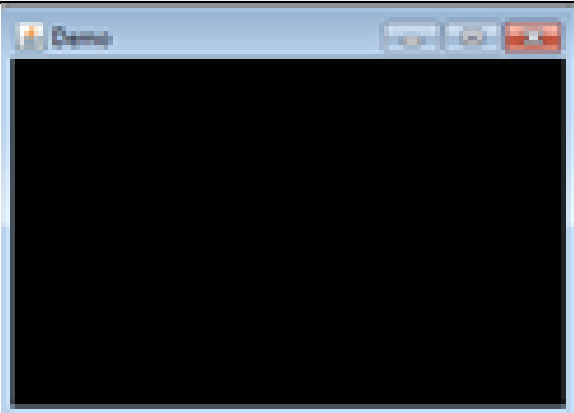
Java AWT	Java Swing
AWT components are <b>platform-dependent</b> .	Java swing components are <b>platform-independent</b> .
AWT components are <b>heavyweight</b> .	Swing components are <b>lightweight</b> .
AWT <b>doesn't support pluggable look and feel</b> .	Swing <b>supports pluggable look and feel</b> .
AWT provides <b>less components</b> than Swing.	Swing provides <b>more powerful components</b> such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
AWT <b>doesn't follows MVC</b>	Swing <b>follows MVC</b> .

```

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();
    }
}

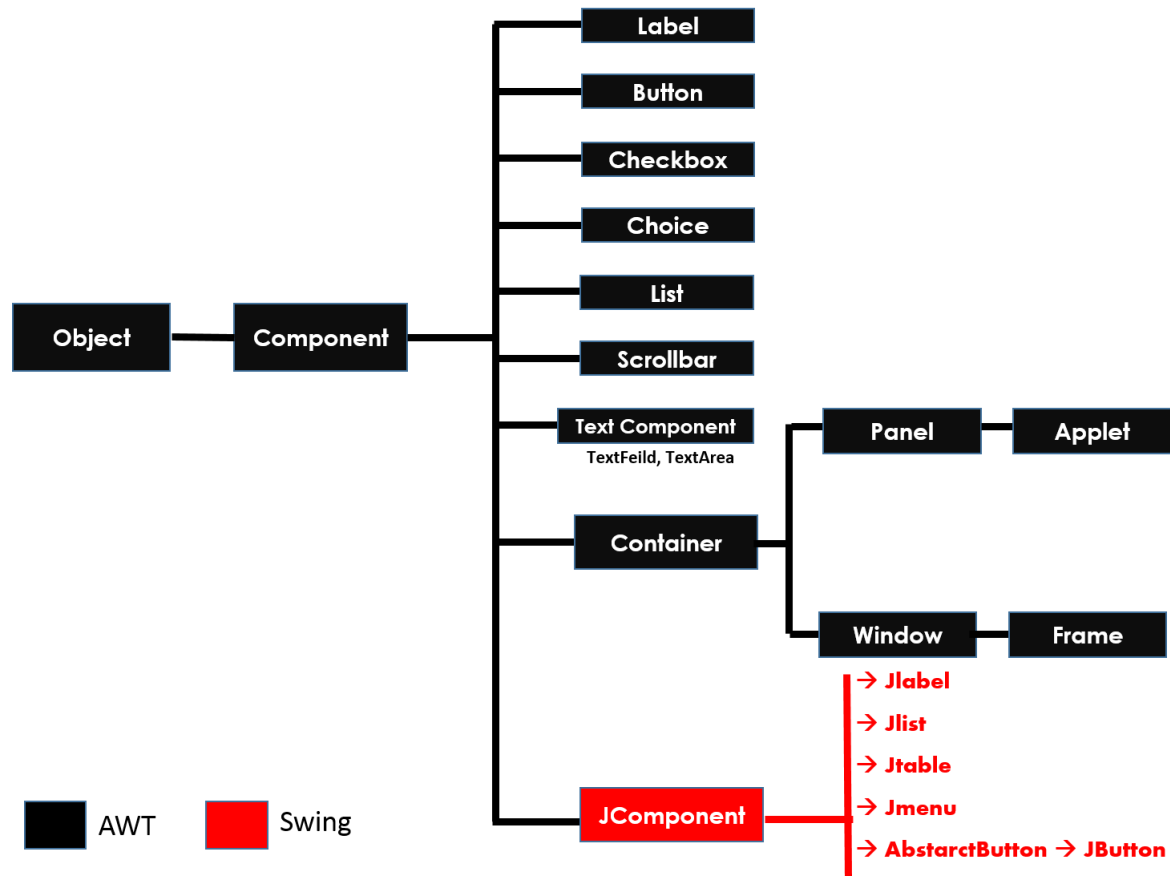
```



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

```

Component class
- manages the Operatons on all Components like Label, Ist e
- It is super class for all AWT components. in API maximum
  all methods of those are in Component API class

public int getX();
public int getY();
public int getWidth();
public int getHeight();
public float getAlignmentX();
public float getAlignmentY();

public void paint( Graphics);
public void repaint();

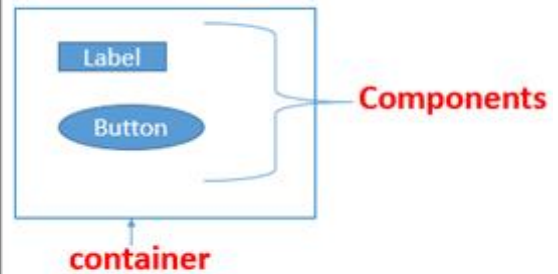
public boolean isVisible();
public void show();
public Color getForeground();
public void setForeground( Color);
public Font getFont();

public boolean mouseDown( Event, int, int);
  
```

## 2. Container

Container is an empty space, where we place components

Container
<ul style="list-style-type: none"> <li>- adding Components</li> <li>- Arrange the Elements Properly</li> </ul>
<pre> public Component add(Component); public Component add(String, Component); public Component add(Component, int);  public void remove(int); public void remove(java.awt.Component); public void removeAll();  public java.awt.LayoutManager getLayout(); public void setLayout(java.awt.LayoutManager);  public void invalidate(); public void validate();  public float getAlignmentX(); public float getAlignmentY();  public void paint(java.awt.Graphics);  public void addNotify(); public void removeNotify();                     </pre>


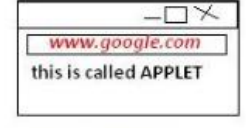


## 3. Window & Frame

Window
<ul style="list-style-type: none"> <li>- A Window object is a top-level window with no borders and no menubar.</li> <li>- A window must have either a frame, dialog, or another window</li> <li>- Windows are capable of generating WindowEvents like WindowOpened, WindowClosed,</li> </ul>
<p>The diagram shows a window with a title bar containing 'title', a red minus sign, a white maximize button, and a red X. The main area of the window contains the text 'this is called "Frame"'. A blue arrow points from the text 'no boarder, no frame After Adding Frame' to the window area. Another blue arrow points from the text 'this is called "Frame"' to the frame area.</p> <pre> public void addWindowListener(event.WindowListener); public final boolean isFocusableWindow(); void preProcessKeyEvent(event.KeyEvent);  boolean eventEnabled(AWTEvent);  public void setSize(int, int); public void setVisible(boolean);                     </pre>

Frame Class
<ul style="list-style-type: none"> <li>- Frame operations are Performed Here</li> <li>- setTitle, Maximize, Minimize, Close, Cursor, Resizable, etc</li> </ul>
<p>The diagram shows a frame with a title bar containing 'title', a red minus sign, a white maximize button, and a red X. The main area of the frame contains the text 'this is called "Frame"'. A red arrow points from the text 'this is called "Frame"' to the frame area.</p> <pre> public String getTitle(); public void setTitle(String);  public boolean isResizable(); public void setResizable(boolean);  public void setCursor(int); public int getCursorType();  public void setMaximizedBounds(Rectangle); public Rectangle getMaximizedBounds();                     </pre>

## 5. Panel & Applet

Panel	Applet
<p>- A Panel is a Logical Space, which is use to adding Containers, Panels also</p> <p>- Adding Layouts, Panels, Containers Etc</p>	<p>- Applet is run on Browser, so It contains Self Life Cycle methods</p> <p>- URL, isAlive, Resize the page these Operations are here</p>
 <pre>public java.awt.Panel(); public java.awt.Panel(java.awt.LayoutManager);</pre>	 <pre>public void init(); public void start(); public void stop(); public void destroy();  public void resize(int, int); public void resize(java.awt.Dimension);  public java.net.URL getDocumentBase(); public java.net.URL getCodeBase();  public boolean isActive();</pre>

## 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. Internalize the Component
2. Create a class which implement Listener Interface
3. Component must register with Listener
4. Get the object of Event class
5. Implement event method.

```
//2.Create a class which implement Listener Interface  
public class LoginDemo extends Frame implements ActionListener {  
  
    public LoginDemo() {  
        //1. Internalize the Component  
        Button login = new Button("Login");  
  
        // 3.Component must register with Listener  
        login.addActionListener(this);  
    }  
  
        // 4.Get the object of Event class  
    public void actionPerformed(ActionEvent e) {  
        //5.Implement event method  
    }  
}
```

### 2. Class which implement Listener Interface

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

Button	<b>java.awt.event.ActionListener</b>
Choice	<b>java.awt.event.ItemListener</b>
TextField	<b>java.awt.event.TextListener</b>
TextArea	<b>java.awt.event.TextListener</b>
Scrollbar	<b>java.awt.event.AdjustmentListener</b>

### 3. 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	<code>public void addActionListener (ActionListener);</code>	<code>public void removeActionListener (ActionListener);</code>
Choice	<code>public void addItemListener (ItemListener);</code>	<code>public void removeItemListener (ItemListener);</code>
TextField	<code>public void addTextListener (TextListener);</code>	<code>public void removeTextListener (TextListener);</code>
TextArea	<code>public void addTextListener (TextListener);</code>	<code>public void removeTextListener (TextListener);</code>

### 4. 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:

- **Name of the component.**
- **Reference of the component.**

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

Component name	Event name
Button	<b>java.awt.event.ActionEvent</b>
choice	<b>java.awt.event.ItemEvent</b>
textField	<b>java.awt.event.TextEvent</b>
textArea	<b>java.awt.event.TextEvent</b>
scrollbar	<b>java.awt.event.AdjustmentEvent</b>

### 4. Implement Event method

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

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

## 5. Components

### 1. Label

constructors	
<code>public Label()</code>	Constructs a Label displaying <i>aString</i> with the <i>alignment</i> , which can be one of CENTER, LEFT or RIGHT, as specified.
<code>public Label( String aString)</code>	
<code>public Label( String aString,           int alignment)</code>	

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

### 2. Button

constructors	
<code>public Button()</code>	Constructs a Button without a label, or with <i>aString</i> as its label.
<code>public Button( String aString)</code>	

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

### 3. Text Field (TextArea)

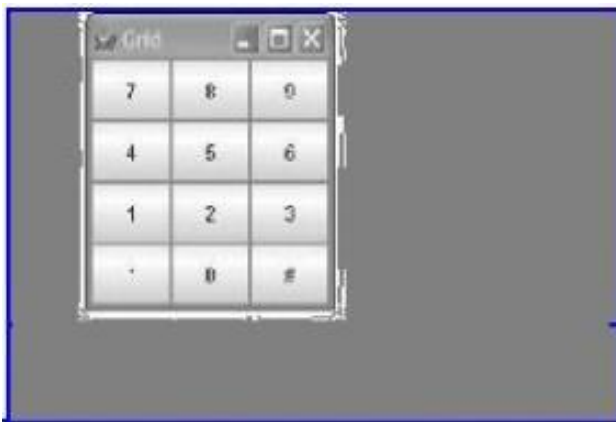
TextFields constructors	
<code>public TextField()</code>	Creates a new empty <code>TextField</code> , or one with contents and number of columns, or default number of columns, as specified
<code>public TextField( String contents)</code>	
<code>public TextField( String contents           int columns)</code>	

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







1. Arranges the components in 'ROW X Col'
2. Components must be of 'SameSize'
3. 'No Space' btwn the components



1. Arranges the components 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. Declaring components
    Label l1, l2, status;
    TextField t1, t2;
    Button login;

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

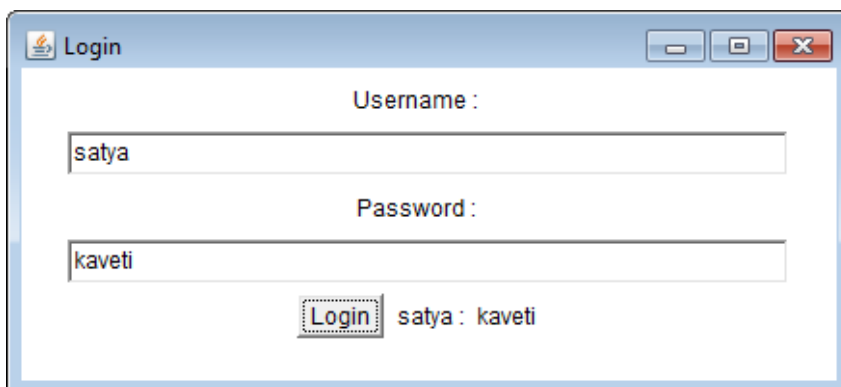
        // 6.creating Component Objects
        l1 = new Label("Username : ");
        l2 = new Label("Password : ");
        status = new Label("Status");
        t1 = new TextField(50);
        t2 = new TextField(50);
        login = new Button("Login");

        // 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 APLET 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 & initialize 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.

```
public class AppletClass extends Applet implements ActionListener {
    Label l;
    Button a, b, c, d, e;

    public void init() {
        setSize(300, 300);
        a = new Button("NORTH EXIT");
        b = new Button("SOUTH");
        c = new Button("WEST");
        d = new Button("EAST");
        l = new Label("OK", Label.CENTER);

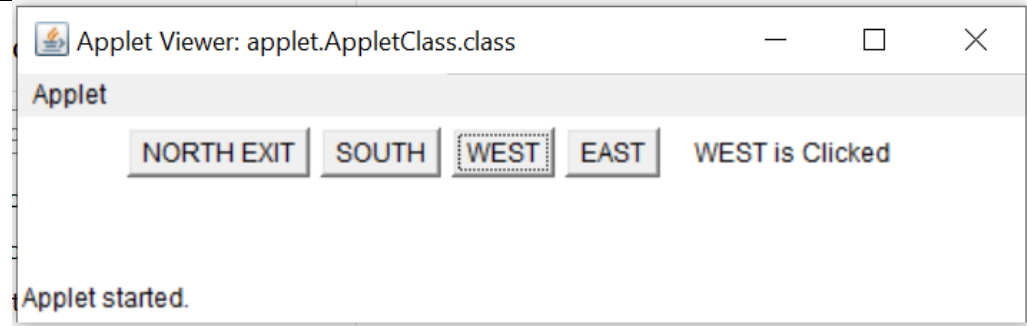
        add(a, "North");
        add(b, "South");
        add(c, "East");
        add(d, "West");
        add(l, "Center");
    } // init

    public void start() {

        a.addActionListener(this);
        b.addActionListener(this);
        c.addActionListener(this);
        d.addActionListener(this);

        // defaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    } // Start
```

```
public void actionPerformed(ActionEvent ae) {  
    if (ae.getSource() == a) {  
        l.setText("NORTH is Clicked");  
        // System.exit(0);  
    }  
    if (ae.getSource() == b) {  
        l.setText("SOUTH is Clicked");  
    }  
    if (ae.getSource() == c) {  
        l.setText("WEST is Clicked");  
    }  
    if (ae.getSource() == d) {  
        l.setText("EAST is Clicked");  
    }  
}
```

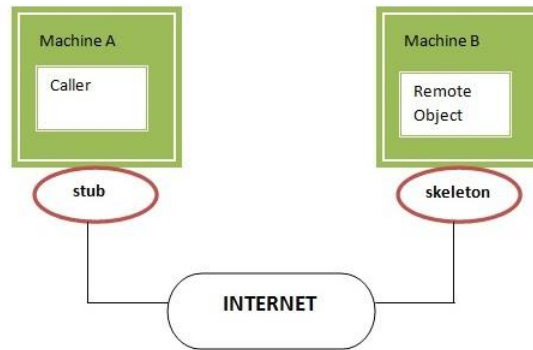


## Features by Version

### Java 4 (2002)

#### RMI(1.1V)

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 invoke 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**.

**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 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 extending `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 extending UnicastRemoteObject

```
public class RemoteClass extends UnicastRemoteObject implements RemoteInterface {
    protected RemoteClass() throws RemoteException {
        super();
    }
    @Override
    public String show(String name) {
        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{
        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"));
    }
}
```

#### JDBC(1.1V) -

We know it already.

#### Assertions

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); //Augmented 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 manually 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.

#### 1. Simple assert

```
assert (boolean expression);
```

- Here the Expression Should be **Boolean** type.
- If expression is **TRUE** it won't return anything,
- Otherwise it will throws **Runtime Exception: java.lang.AssertionError: Default message**

```
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: Default message**

#### 2.Augmented assert

```
assert (boolean expression1): (anytype expression2); //Augmented assert
```

**Expression2** → is used to Display some message along with Error Message

- Here the 1<sup>st</sup> Expression Should be **Boolean** type, 2<sup>nd</sup> Expression can be **Any type**
- If expression is **TRUE** it won't return anything,
- Otherwise it will throws **Runtime Exception: java.lang.AssertionError: expression2**

## Second parameter mostly used for customized error message

```
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** File → Run As → Run Config.,>Vm args = -ea > Save>Run

## 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 Compiled pattern
- **Matcher** class - used for performing matching operations on compiled pattern
- **PatternSyntaxException** - checks syntax error in a regular expression pattern.

**1. Pattern class** it represents the Compiled pattern

Method	Description
static Pattern <b>compile</b> (String regex)	Compiles given regex and return the instance of pattern.
Matcher <b>matcher</b> (CharSequence input)	Retunes char sequence to be compare with patten
boolean <b>matches</b> (String regex, String charSequence)	Directly we can compare Expression with Sequence
String <b>pattern</b> ()	returns the regex pattern.

**2. Matcher class** -used for performing matching operations on compiled pattern

Method	Description
boolean <b>matches</b> ()	Test whether the regular expression matches the pattern.
boolean <b>find</b> ()	Finds the next expression that matches the pattern.
boolean <b>find</b> (int start)	Finds the next expression that matches the pattern from the given start number.
String <b>group</b> ()	Returns the matched subsequence.
int <b>start</b> ()	Returns the starting index of the matched subsequence.
int <b>end</b> ()	Returns the ending index of the matched subsequence.
int <b>groupCount</b> ()	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: [a-d-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
<b>X?</b>	X occurs once or not at all
<b>X+</b>	X occurs once or more times
<b>X*</b>	X occurs zero or more times
<b>X{n}</b>	X occurs n times only
<b>X{n,}</b>	X occurs n or more times
<b>X{y,z}</b>	X occurs at least y times but less than z times

## 3. Regex Metacharacters

The regular expression metacharacters work as a short code.

Regex	Description
<b>.</b> (dot)	Any character (may or may not match terminator)
<b>\d</b>	Any digits, short of [0-9]
<b>\D</b>	Any non-digit, short for [^0-9]
<b>\s</b>	Any whitespace character, short for [\t\n\x0B\f\r]
<b>\S</b>	Any non-whitespace character, short for [^\s]
<b>\w</b>	Any word character, short for [a-zA-Z_0-9]

<b>\W</b>	Any non-word character, short for [^\w]
<b>\b</b>	A word boundary
<b>\B</b>	A non-word boundary

```

public class REDemo {
public static void main(String[] args) {
    System.out.println("1.Regex Character classes\n-----");
    System.out.println(Pattern.matches("[amn]", "abcd")); //false (not a or m or n)
    System.out.println(Pattern.matches("[amn]", "a")); //true (among a or m or n)
    System.out.println(Pattern.matches("[amn]", "ammmna")); //false(m & a morethan once)

    System.out.println("\n2.Regex Quantifiers\n-----");
    System.out.println("? quantifier ....");
    System.out.println(Pattern.matches("[amn]?", "a")); //true (a or m or n comes one time)
    System.out.println(Pattern.matches("[amn]?", "aaa")); //false (a comes more than one time)
    System.out.println(Pattern.matches("[amn]?", "aammnn")); //false (a m n comes more than one time)

    System.out.println("+ quantifier ....");
    System.out.println(Pattern.matches("[amn]+", "a")); //true (a or m or n once or more times)
    System.out.println(Pattern.matches("[amn]+", "aaa")); //true (a comes more than one time)

    System.out.println("\n3.Regex Metacharacters\n-----\n");
    System.out.println(Pattern.matches("\\d", "abc")); //false (non-digit)
    System.out.println(Pattern.matches("\\d", "1")); //true (digit and comes once)
    System.out.println(Pattern.matches("\\d", "4443")); //false (digit but comes more than 1)
}
}

```

## Logging API

In common we use **System.out.println()** statements for DEBUGGING. But these are printed at console and they will be 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 Format 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 enable Log4j in our java class
- Logger methods are used to generate log statements in a java class instead of sop's

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 ("")**

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

- **ConsoleAppender** [ 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**
- **HTMLayout**
- **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 program 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
```

## Java 5 (2005)

### Annotations

Java **Annotation** is a tag that represents the *metadata* i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.

Annotations have several uses, below are few among them:

- **Information for the compiler** — Annotations can be used by the compiler to detect errors or suppress warnings.
- **Compile-time and deployment-time processing** — Software tools can process annotation information to generate code, XML files, and so forth. (Spring Autowire, Java Doc kind of)
- **Runtime processing** — Some annotations are available to be examined at runtime

The annotation can include **elements(parameters)**, which can be named or unnamed, and we need to provide values for those elements:

```
@Author(  
    name = "Benjamin Franklin",  
    date = "3/27/2003"  
)  
class MyClass() { ... }
```

If there is **just one element** named value, then the name can be omitted, as in:

```
@SuppressWarnings("unchecked")  
void myMethod() { ... }
```

If the annotation has no elements, then the parentheses can be omitted

example : `@Override`

It is also possible to use multiple annotations on the same declaration:

```
@Author(name = "Jane Doe")  
@EBook  
class MyClass { ... }
```

If the annotations have the same type, then this is called a repeating annotation Repeating annotations are supported as of the Java SE 8 release.

```
@Author(name = "Jane Doe")  
@Author(name = "John Smith")  
class MyClass { ... }
```

**Java Custom annotations** or Java User-defined annotations are easy to create and use.

The **@interface** element is used to declare an annotation.

- Method should **not throw Exception**
- Method **should not have any parameter.**
- Method should **return something**
- It **may assign a default value to the method.**

```
@interface MyAnnotation{  
    int value() default 0;  
}
```

### Example

Suppose Every class in a given Project should contain author info. Writing Author info in every class is difficult like below

```
public class GenrateWorkSheet extends DataList {  
    // Author: John Doe  
    // Date: 3/17/2002  
    // Current revision: 6  
    // Last modified: 4/12/2004  
    // By: Jane Doe  
    // Reviewers: Alice, Bill, Cindy  
  
    // class code goes here  
}
```

By Using Annotation, we can provide above information. For doing this we need to create an annotation & all property should define using methods.

```
@interface AuthaorInfo {  
    String author();  
    String date();  
    int currentRevision() default 1;  
}
```

```
String lastModified() default "N/A";
String lastModifiedBy() default "N/A";
// Note use of array
String[] reviewers();
}
```

Once annotation ready, you can write that annotation top of each class with values

```
@AuthoInfo (
    author = "John Doe",
    date = "3/17/2002",
    currentRevision = 6,
    lastModified = "4/12/2004",
    lastModifiedBy = "Jane Doe",
    // Note array notation
    reviewers = {"Alice", "Bob", "Cindy"}
)
public class Generation3List extends Generation2List {
    // class code goes here
}
```

### Autoboxing

- The automatic conversion of primitive data types into its equivalent Wrapper type is known as boxing and opposite operation is known as unboxing.
- Up to JDK 1.4, all the data structures of Java stores only objects and when retrieved returns objects. Autoboxing permits to store data types directly in DS and retrieve back data types.
- If a **method (remember only method – not direct)** requires Integer Object value, we can directly pass primitive value without issue. Autoboxing will take care about these.

We can also do direct initializations (1.8 V)

```
Integer i = 10; // it will create Integer value of 10 using Autoboxing
int j = i; // it will convert Integer to int using Autoboxing
```

Previously it shows

```
Integer i = 10; // it will create Integer value of 10 using Autoboxing
int j = i; // But we cant assign int to Integer Type mismatch: cannot convert from Integer to int
```

### Generics

Generics are introduced in Java 1.5 Version to solve **Type-safety & 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);
    }
}
```

- **line1:** ArrayList is **NOT generic** type – so we can add any type of elements results **Type-Safety**
- **line2:** It returns added String data as Object. So manually we have to **Type-cast to String**

To resolve above problems Generics are introduced

```
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**, because **TYPE-CHECKING** done at **COMPILE-TIME** itself

### Generics can be used in following areas

- **Interface level**
- **Class 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. Similarly, we have to follow below naming conventions to where to use which Naming conventions. 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 etc)
3. **E – Element** - Exclusively 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. Generics at Class /Method /Constructor 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) {
```

```
SmIGen<String> s = new SmIGen<String>();
s.add("Satya");
System.out.println(s.get());
}
}
```

## 2. Generics at Interface level

A generic interface is same as generic class.

```
public interface List<T> extends Collection<T> {
...
}
```

## 2. Wildcard in Generics

? Operator is used to represents wildcards in generics. We have two types of Wildcards

- 1) **Unbounded wildcards**
- 2) **Bounded wildcards**

### 1. Unbounded wildcards

Unbounded wildcard like <?> - means the generic can be any type.it is not bounded with any type.

### 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 **Parent class Objects of the type<T>**

## Covariant Return Type (Java 5)

Before Java5, it was not possible to override any method by changing the return type. But, since Java5, it is possible to override method by changing the return type. If subclass overrides any method whose return type is **Non-Primitive(Object type)**, it can changes its return type to subclass type.

### Enhanced for 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){}
```

## 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,

## 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

## Var-args

Sometimes 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 ... variablename)
```

```

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

```

static void show(String... var) ✓
static void show(String... var, int i) ✗
static void show(int i, String... var) ✓

```

### 2. Only one Var-arg is allowed per a Method

```

static void show(String... var) ✓
static void show(String... var, int...y) ✗
static void show(String i, int...y, String v) ✗
static void show(String i, int y, String... v) ✓

```

## 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, Constructors, 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 Enum\_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

### Static imports

The static import feature of Java 5 facilitates the java programmer to 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");
    }
}
```

### C-lang printf()

**printf()** is an extra method added to **PrintStream** class from JDK 1.5 version. **printf()** is used to print at command-prompt

```
public class PrintfDemo
{
```



```

public static void main(String args[])
{
    boolean hot = false;
    char alphabet = 'A';
    int rate = 5;
    float temperature = 12.3456789f;

    System.out.printf("The boolean hot: %b", hot);           // b for boolean
    System.out.printf("\nThe char alphabet: %c", alphabet); // c for char
    System.out.printf("\nThe int rate: %d", rate);          // d for int
    System.out.printf("\nThe float marks is %f", temperature); // f for float

    System.out.printf("\n value prefixed with 4 zeros: %04d", rate); // filling with zeros
    System.out.printf("\float temp %.3f", temperature); // precision to three decimal values
    System.out.printf("\nThe float temperature in exponential form: %e", temperature);
    System.out.printf("\n%s, %s and also %s belong to java.lang package", "System","String","Math");
    System.out.printf("\ntemperature is %.2f", temperature); // width is 4 and precision to 2 decimal
}
}

```

Java6 not many features

## Java 7 (2011)

### Underscores in Numeric Literals (Java 7)

Java allows you to use underscore in numeric literals.

- You cannot use underscore at the beginning or end of a number.
- You cannot use underscore adjacent to a decimal point in a floating-point literal. ( 10\_0)
- You cannot use underscore prior to an F or L suffix (**long** a = 10\_100\_00\_L);
- added in Java 7. You can have underscores in numbers to make them easier to read:

```

int million1 = 1000000;
int million2 = 1_000_000;

double notAtStart = _1000.00; // DOES NOT COMPILE
double notAtEnd = 1000.00_; // DOES NOT COMPILE
double notByDecimal = 1000_.00; // DOES NOT COMPILE

```

### String in switch statement (Java 7)

```

public class StringInSwitchStatementExample {
    public static void main(String[] args) {
        String game = "Cricket";
        switch(game){
            case "Hockey":
                System.out.println("Let's play Hockey");
                break;
            case "Cricket":
                System.out.println("Let's play Cricket");
                break;
            case "Football":
                System.out.println("Let's play Football");
        }
    }
}

```

### The try-with-resources (Java 7)

- try-with-resources statement is a try statement that declares one or more resources. The resource is as an object that must be closed after finishing the program.
- The try-with-resources statement ensures that each resource is closed at the end of the statement execution.
- You can pass any object that implements java.lang.AutoCloseable, which includes all objects which implement java.io.Closeable

### Caching Multiple Exceptions by single catch (Java 7)

Java allows you to catch multiple type exceptions in a single catch block. It was introduced in Java 7 and helps to optimize code. You can use vertical bar (|) to separate multiple exceptions in catch block.

### Diamond operator <>, removes right side of generics

you can replace the type arguments with an empty set of type parameters (<>). This pair of angle brackets is informally called the diamond.

The following approach is used in Java 6 and prior version.

```
Ex. List<Integer> list = new List<Integer>();
```

Now, you can use the following new approach introduced in Java 7.

```
Ex. List<Integer> list = new List<>(); // Here, we just used diamond
```

## Java 8 (2014)

### Functional Interfaces (Java 8)

An Interface that contains exactly one abstract method is known as functional interface. It can have any number of default, static methods but can contain only one abstract method. It can also declare methods of object class

```
@FunctionalInterface
interface sayable{
    void say(String msg);
}
```

### Lambda Expressions (Java 8)

The Lambda expression is used to provide the implementation of an interface which has functional interface

### Stream (Java 8)

Another major change introduced **Java 8 Streams API**, which provides a mechanism for processing a set of data in various ways that can include filtering, transformation, or any other way that may be useful to an application.

Streams API in Java 8 supports a different type of iteration where you simply define the set of items to be processed, the operation(s) to be performed on each item, and where the output of those operations is to be stored.

## Default Methods & Static methods Interface (Java 8)

Java 8 allows you to add non-abstract methods in interfaces. These methods must be declared default methods. Default methods enable you to add new functionality to the interfaces of your libraries and ensure binary compatibility with code written for older versions of those interfaces.

```
public interface Moveable {
    default void move(){
        System.out.println("I am moving");
    }
}
static void isNull(String str) {
    System.out.println("Interface Null Check");
}
```

If class willingly wants to customize the behavior of move() method then it can provide it's own custom implementation and override the method. In this case we must remove **default** keyword.

**Static methods:** Java interface static method is similar to default method except that we can't override them in the implementation classes.

## Java 8 Date/Time API (Java 8)

**Date** class has even become obsolete. The new classes intended to replace Date class are **LocalDate**, **LocalTime** and **LocalDateTime**.

- The **LocalDate** class represents a date. There is no representation of a time or time-zone.
- The **LocalTime** class represents a time. There is no representation of a date or time-zone.
- The **LocalDateTime** class represents a date-time. There is no representation of a time-zone.

```
LocalDate localDate = LocalDate.now();
LocalTime localTime = LocalTime.of(12, 20);
LocalDateTime localDateTime = LocalDateTime.now();
OffsetDateTime offsetDateTime = OffsetDateTime.now();
ZonedDateTime zonedDateTime = ZonedDateTime.now(ZoneId.of("Europe/Paris"));
```

**Duration** class is a whole new concept brought first time in java language. It represents the time difference between two time stamps.

```
Duration duration = Duration.ofMillis(5000);
duration = Duration.ofSeconds(60);
duration = Duration.ofMinutes(10);
```

**Duration** deals with small unit of time such as milliseconds, seconds, minutes and hour. They are more suitable for interacting with application code. To interact with human, you need to get bigger durations which are presented with **Period** class.

```
Period period = Period.ofDays(6);
period = Period.ofMonths(6);
period = Period.between(LocalDate.now(), LocalDate.now().plusDays(60));
```

## Optional Class

- Every Java Programmer is familiar with **NullPointerException**. We need too many null checks in our program.
- So, to overcome this, Java 8 has introduced a new class **Optional** in **java.util** package. It can help in writing a neat code without using too many null checks.
- By using **Optional**, we can specify alternate values to return or alternate code to run.

The Optional class has many methods but only two methods are most used in the coding. Those two are **isPresent()** and **get()** methods.

- **isPresent()** returns true if the optional has non-null values, otherwise false.
- **get()** returns the actual value from Optional object. If optional has null value means it is empty. In such a case, `NoSuchElementException` is thrown.

## Java 8 Optional Class Methods & Description

Methods	What it does?
<code>empty()</code>	Creates an empty Optional object.
<code>of()</code>	Creates an Optional object with specified non-null value.
<code>ofNullable()</code>	Creates an Optional object with specified value if it is non-null, otherwise returns empty Optional.
<code>get()</code>	Returns value present in Optional object. If the value is absent, throws <code>NoSuchElementException</code> .
<code>orElse()</code>	Returns value if present, otherwise returns supplied value.
<code>ifPresent()</code>	Performs specified action if value is present, otherwise no action taken.
<code>isPresent()</code>	Checks whether value is present or not.
<code>orElseGet()</code>	Returns value if present, otherwise returns result of specified supplier.
<code>orElseThrow()</code>	Returns value if present, otherwise throws exception created by specified supplier.
<code>map()</code>	Applies given mapping function if value is present. Returns Optional containing the result. If result is null, returns empty Optional.
<code>flatMap()</code>	It is used when mapper function returns another Optional as a result and you don't want to wrap it in another Optional.
<code>filter()</code>	If value is present and matches with given predicate then it returns Optional containing the result. Otherwise returns empty Optional.

```
Optional<Anthology> findById(Integer id);

Anthology anthology = repo.findById(anthologyId).orElseThrow(
    () -> new ResourceNotFoundException("Anthology not found for this id :: " + anthologyId));
```

## Java 9 (2017)

### JShell: The Java Shell (REPL)

It is an interactive Java Shell tool; it allows us to execute Java code from the shell and shows output immediately. JShell is a REPL (Read Evaluate Print Loop) tool and run from the command line.

### Module System (Project Jigsaw)

- In earlier versions of Java, there was no concept of module to create modular Java applications. that why size of application increased and difficult to move around. Even JDK itself was too heavy in size, in Java 8, **rt.jar** file size is around 64MB.
- To deal with situation, **Java 9 restructured JDK into set of modules** so that we can use only required module for our project

- Create a file **module-info.java**, inside this file, declare a module by using **module** identifier and provide module name same as the directory name that contains it. In our case, our directory name is com.javatpoint.

### Interface Private Methods

In Java 9, we can create private methods inside an interface. Interface allows us to declare private methods that help to **share** common code between **non-abstract** methods / Default methods.

```
interface Sayable{
    default void say() {
        saySomething();
    }
    //Private method inside interface
    private void saySomething() {
        System.out.println("Hello... I'm private method");
    }
}
public class PrivateInterface implements Sayable {
    public static void main(String[] args) {
        Sayable s = new PrivateInterface();
        s.say();
    }
}
```

### Multi-Release JAR Files

Previously, you had to package all classes into a **jar** file and drop in the class path of another application, which wish to use it.

Using multi-release feature, now a jar can contain **different versions of a class – compatible to different JDK releases**. The information regarding different versions of a class, and in which JDK version which class shall be picked up by class loader, is stored in **MANIFEST.MF** file.

## Java10 (March 2018)

### Local Variable Type Inference

Java has now **var** style declarations. It allows you to declare a local variable without specifying its type. The type of variable will be inferred from type of actual object created. It claims to be the only real feature for developers in JDK 10

```
var str = "Hello world";
```

### Time-Based Release Versioning

On printing java version, it just shows version number only

```
E:\Users\Kaveti_S>java -version
java version "1.8.0_202-ea"
```

But in 10, it will show the Date of version release The new pattern of the Version number is:

```
$FEATURE. $INTERIM. $UPDATE. $PATCH
java version "10" 2018-03-20
```

### Garbage Collector Interface

Anyone wanting to implement a new GC would require knowledge about all these various places, as well as how to extend the various classes for their specific needs.

## Java 11 (September 2018)

They've changed the licensing and support model which means if you download the Java 11 Oracle JDK, it will be **paid for commercial use**

### Does that mean that I need to pay for Java from now on?

**NO.** Not necessarily unless you **download the Oracle JDK and use it in production**

You can use it in developing stages but to use it commercially, you need to buy a license. If you don't, you can get an invoice bill from Oracle any day!

**Java 10 was the last free Oracle JDK that could be downloaded.**

### Running Java File with single command

One major change is that you don't need to compile the java source file with `javac` tool first. You can directly run the file with java command, and it implicitly compiles

### Java String Methods

#### isBlank()

```
System.out.println(" ".isBlank()); //true

String s = "Anupam";
System.out.println(s.isBlank()); //false
```

#### repeat(int)

The repeat method simply repeats the string that many numbers of times as mentioned in the method in the form of an int.

```
String str = ".".repeat(2);
System.out.println(str); //prints ==
```

## Java 12 (March 2019)

Java 12 got a couple of new features and clean-ups, but the only ones worth mentioning here are **Unicode 11 support** and a preview of the new switch expression, which you will see covered in the next section.

## Java 13 (September 2019)

You can find a complete feature list [here](#), but essentially, you are getting **Unicode 12.1 support**, as well as **two new or improved preview features** (subject to change in the future):

### 1.Switch Expression (Preview)

Switch expressions can now **return a value**. And you can use a **lambda-style syntax** for your expressions,

```
//Old switch statements looked like this:
switch(status) {
    case SUBSCRIBER:
        // code block
        break;
    case FREE_TRIAL:
```

```

// code block
break;
default:
// code block
}

//Whereas with Java 13, switch statements can look like this:
boolean result = switch (status) {
    case SUBSCRIBER -> true;
    case FREE_TRIAL -> false;
    default -> throw new IllegalArgumentException("something is murky!");
};

```

## 2. Multiline Strings (Preview)

String should start with three Quotes """"

```

String htmlBeforeJava13 = "<html>\n" +
    "    <body>\n" +
    "        <p>Hello, world</p>\n" +
    "    </body>\n" +
    "</html>\n";

String htmlWithJava13 = """"
    <html>
    <body>
        <p>Hello, world</p>
    </body>
</html>
""";

```

## Java 14 (March 2020)

- [JEP 305 – Pattern Matching for instanceof \(Preview\)](#)
- [JEP 368 – Text Blocks \(Second Preview\)](#)
- [JEP 358 – Helpful NullPointerExceptions](#)
- [JEP 359 – Records \(Preview\)](#)
- [JEP 361 – Switch Expressions \(Standard\)](#)
- JEP 343 – Packaging Tool (Incubator)
- JEP 345 – NUMA-Aware Memory Allocation for G1
- JEP 349 – JFR Event Streaming
- JEP 352 – Non-Volatile Mapped Byte Buffers
- [JEP 363 – Remove the Concurrent Mark Sweep \(CMS\) Garbage Collector](#)
- JEP 367 – Remove the Pack200 Tools and API
- JEP 370 – Foreign-Memory Access API (Incubator)

## Java 15 (September 2020)

- [1. JEP 339: Edwards-Curve Digital Signature Algorithm \(EdDSA\)](#)
- [2. JEP 360: Sealed Classes \(Preview\)](#)  
Sealed Classes or Interfaces have the power to disallow themselves to be implemented or overridden by any object, which is not one of a given list of types

```
public sealed interface FrontEndDeveloper permits Javascripter, Htmler
```

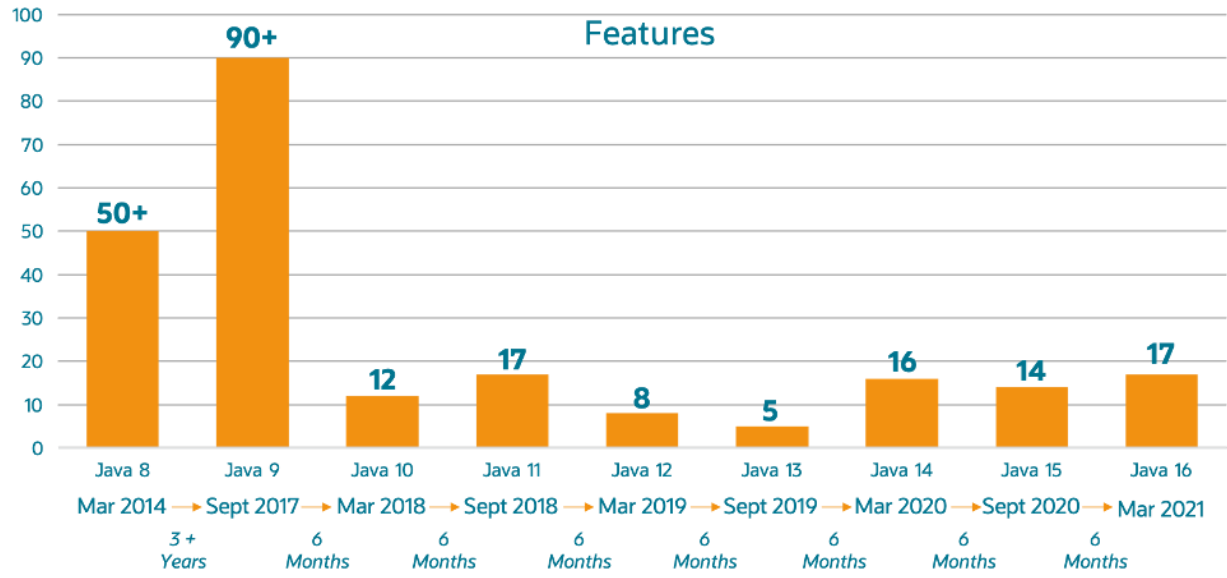
- **[JEP 371: Hidden Classes](#)**
- [JEP 372: Remove the Nashorn JavaScript Engine](#)
- [JEP 373: Reimplement the Legacy DatagramSocket API](#)
- [JEP 374: Disable and Deprecate Biased Locking](#)
- [JEP 375: Pattern Matching for instanceof \(Second Preview\)](#)
- **[JEP 377: ZGC: A Scalable Low-Latency Garbage Collector](#)**
- **[JEP 378: Text Blocks\(Finally Released\)](#)**
- [JEP 379: Shenandoah: A Low-Pause-Time Garbage Collector](#)
- [JEP 383: Foreign-Memory Access API \(Second Incubator\)](#)
- [JEP 384: Records \(Second Preview\)](#)
- [JEP 385: Deprecate RMI Activation for Removal](#)

## Java 16 (March 2021) – Latest

1. JEP 338: Vector API (Incubator)
2. **JEP 347: Enable C++14 Language Features**
3. JEP 357: Migrate from Mercurial to Git
4. **JEP 369: Migrate to GitHub**
5. JEP 376: ZGC: Concurrent Thread-Stack Processing
6. JEP 380: Unix-Domain Socket Channels
7. JEP 386: Alpine Linux Port
8. JEP 387: Elastic Metaspace
9. JEP 388: Windows/AArch64 Port
10. JEP 389: Foreign Linker API (Incubator)
11. JEP 390: Warnings for Value-Based Classes
12. **JEP 392: Packaging Tool**
13. JEP 393: Foreign-Memory Access API (Third Incubator)
14. JEP 394: Pattern Matching for instanceof
15. **JEP 395: Records**
16. JEP 396: Strongly Encapsulate JDK Internals by Default
17. **JEP 397: Sealed Classes (Second Preview)**

More .. [https://en.wikipedia.org/wiki/Java\\_version\\_history](https://en.wikipedia.org/wiki/Java_version_history)





Version	Release date	End of Free Public Updates <sup>[1][6][7][8]</sup>	Extended Support Until
JDK Beta	1995	?	?
JDK 1.0	January 1996	?	?
JDK 1.1	February 1997	?	?
J2SE 1.2	December 1998	?	?
J2SE 1.3	May 2000	?	?
J2SE 1.4	February 2002	October 2008	February 2013
J2SE 5.0	September 2004	November 2009	April 2015
Java SE 6	December 2006	April 2013	December 2018 December 2026, paid support for Azul Platform Core <sup>[9]</sup>
Java SE 7	July 2011	April 2015	July 2022
Java SE 8 (LTS)	March 2014	<b>January 2019 for Oracle (commercial)</b> December 2030 for Oracle (non-commercial) December 2030 for Azul At least May 2026 for AdoptOpenJDK At least May 2026 for Amazon Corretto	December 2030
Java SE 9	September 2017	March 2018 for OpenJDK	N/A
Java SE 10	March 2018	September 2018 for OpenJDK	N/A
Java SE 11 (LTS)	September 2018	September 2026 for Azul At least October 2024 for AdoptOpenJDK At least September 2027 for Amazon Corretto At least October 2024 for Microsoft <sup>[10][11]</sup>	September 2026 September 2028 for Azul <sup>[9]</sup>
Java SE 12	March 2019	September 2019 for OpenJDK	N/A
Java SE 13	September 2019	March 2020 for OpenJDK	N/A
Java SE 14	March 2020	September 2020 for OpenJDK	N/A
Java SE 15	September 2020	March 2021 for OpenJDK March 2023 for Azul <sup>[9]</sup>	N/A
<b>Java SE 16</b>	March 2021	September 2021 for OpenJDK	N/A
Java SE 17 (LTS)	September 2021	September 2029 for Azul At least September 2027 for Microsoft	September 2029 or later September 2031 for Azul Platform Prime
Java SE 18	March 2022	September 2022 for OpenJDK	N/A

**Legend:** ■ Old version ■ Older version, still maintained ■ Latest version ■ Future release

## Ref.

<https://srikarbandla.wordpress.com/2015/07/31/static-control-flow/>

<https://howtodoinjava.com/java-version-wise-features-history/>

[https://www.marcobehler.com/guides/a-guide-to-java-versions-and-features#\\_java\\_features\\_8\\_16](https://www.marcobehler.com/guides/a-guide-to-java-versions-and-features#_java_features_8_16)