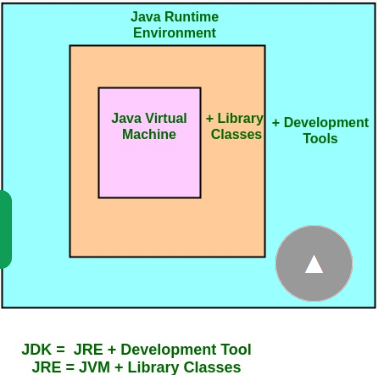
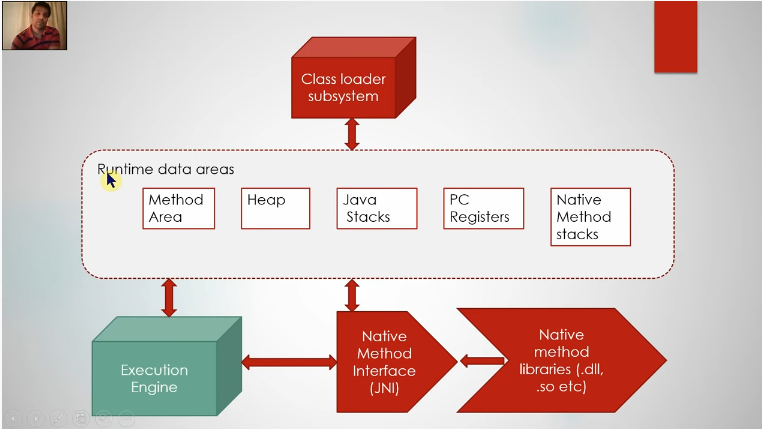
JDK + JRE + JVM



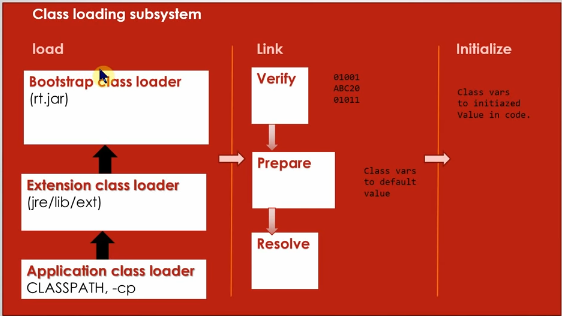
JVM Architecture:



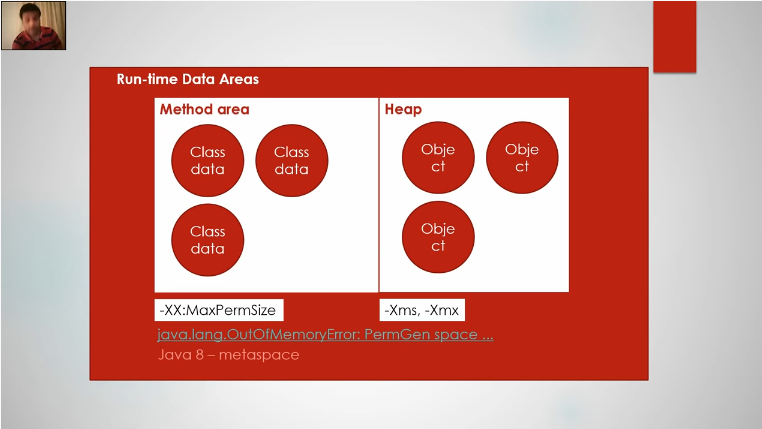
* Native methods is used to interact with host machine.



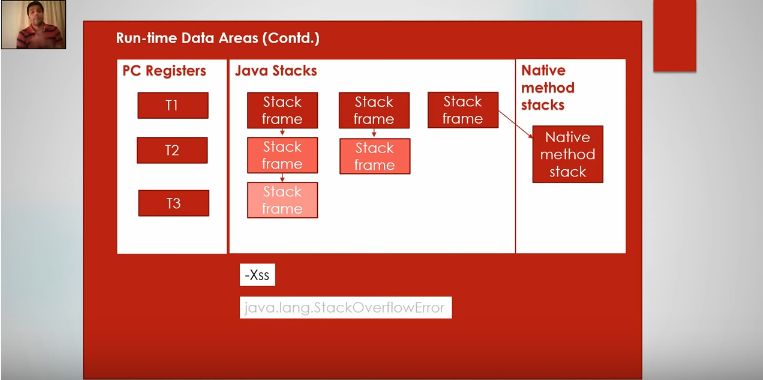
Class Loaders in Java programming language provides a way to load Java classes into the memory. The Class Loaders are important and powerful tool in JVM, which loads Java classes into memory when required. The Java class is loaded for the very first time when JVM calls public static void main(String [] args) method present in Java class file to load at runtime. All the classes referenced by the main class are than loaded when needed. A class loader than creates a namespace for the java classes loaded and already running into the JVM.



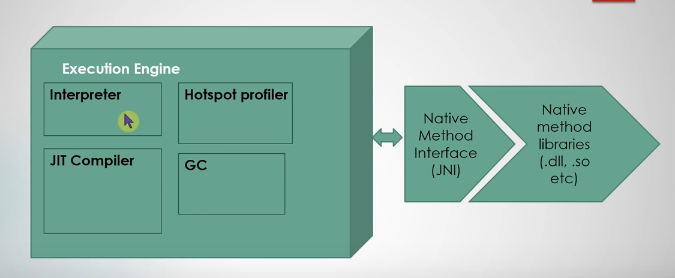
* ClassNotFoundException happens when class loader fails find byte code corresponding class we call.
* Types of class loader: Bootstrap, Extension and Application class Loader.



* Method area is stored in PermGen, max default is 64MB.
* Static, bytecode and final are stored in method area.
* Java-8 metaspace.
* Method area and Heap is created per JVM.
* Object are created in heap. 1/4th of the over total physical memory.



* Program Count (PC registers) will keep track of count of process call by Thread.
* Recursion causes StackOverFlowError.
* Above Data area is created per thread.



* Interpreter: interprets current instruction which is present in bytecode and executes it. Uses native method used for execution.
* Just In Time: what it does is if there are certain set of instruction which is executing repeatedly whose will not interpreted again and again. Which make machine code ready for execution.
* (Hotspot VM): Hotspot keep track of bytecode running and extract lot of statics. Helps JIT compiler.
* **HotSpot** works by running Java code in interpreted mode, while running a **profiler** in parallel. The **HotSpot profiler** looks for "**hot spots**" in the code, i.e. methods that the **JVM** spends a significant amount of time running, and then compiles those methods into native generated code.

// removing element using ArrayList's remove method during iteration

// This will not throw ConcurrentModificationException

**Iterator**<**String**> itr = loans.iterator();

**while** (itr.hasNext()) {

**String** loan = itr.next();

**if** (loan.equals("personal loan")) {

itr.remove();

}

}

wrapper classes – Integer, Double etc

[**What is Autoboxing and Unboxing in Java**](https://javarevisited.blogspot.com/2012/07/auto-boxing-and-unboxing-in-java-be.html)

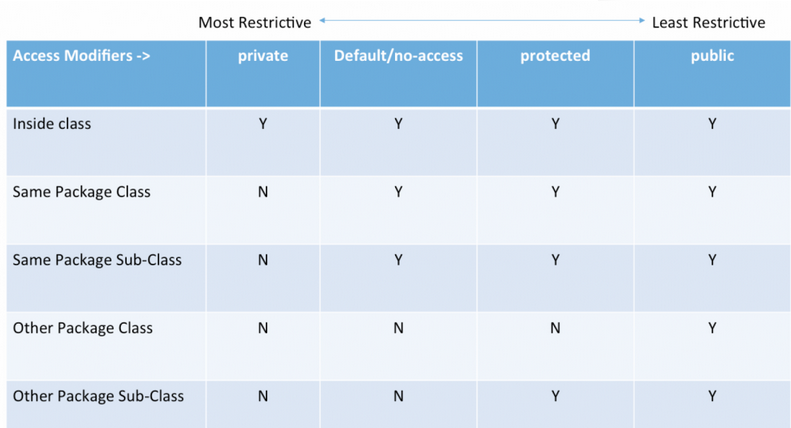
**Autoboxing and Unboxing**. **Autoboxing** is the automatic conversion that the **Java** compiler makes between the primitive types and their corresponding object wrapper classes. For **example**, converting an int to an Integer, a double to a Double, and so on. If the conversion goes the other way, this is called **unboxing**.

**Access Specifier**: This can be understood as the access you provide to your code in Java whether other classes can access your code or not.

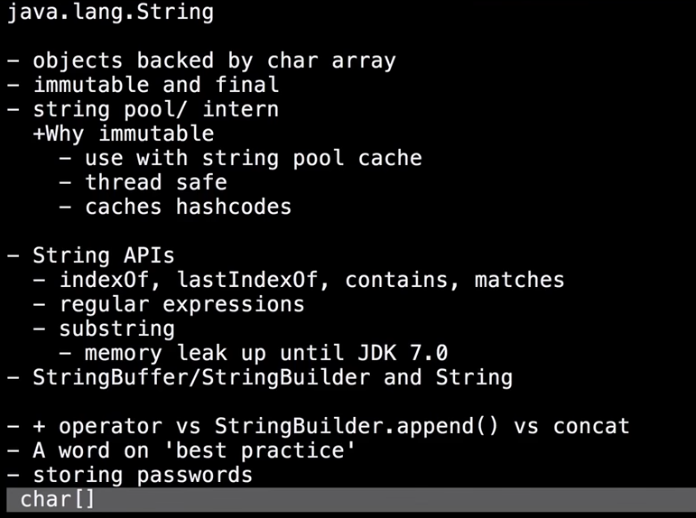
E.g. public, private, protected and default.

**Access Modifier**: Java provides both Access Specifier and Access Modifiers for creating access to your Java code for other classes. Here modifier is also used to do the same task but there are limitations.

1. **Class Modifier**:
   * **abstract**: This defines the restriction such that objects cannot be created.
   * **final**: This restricts a class from being inherited.
   * **strictfp**: it is related to the checking of floating point values irrespective of OS.
2. **Variable Modifier**:
   * **static**: no object creation required
   * **final**: cannot be reassigned
   * **transient**: it is not serialized
   * **volatile**: the values are liable for change



String:



There are two ways to create a String object:

1. **By string literal**: Java String literal is created by using double quotes.  
   For Example: String s=“Welcome”;
2. **By new keyword**: Java String is created by using a keyword “new”.  
   For example:  String s=new String(“Welcome”);    
   It creates two objects (in String pool and in heap) and one reference variable where the variable ‘s’ will refer to the object in the heap.

Now, let us understand the concept of Java String pool.

**Java String Pool:** Java String pool refers to collection of Strings which are stored in heap memory. In this, whenever a new object is created, String pool first checks whether the object is already present in the pool or not. If it is present, then same reference is returned to the variable else new object will be created in the String pool and the respective reference will be returned.

The String class is immutable, so that once it is created a String object cannot be changed. The String class has a number of methods, some of which will be discussed below, that appear to modify strings. Since strings are immutable, what these methods really do is create and return a new string that contains the result of the operation.

From **Java** 7 onwards, the **Java String Pool is stored** in the Heap space, which is garbage collected by the JVM. Before it was in permGen.

String obj = new String("Samuel");

String obj1 = new String("Samuel");

//vs

String obj = "Samuel";

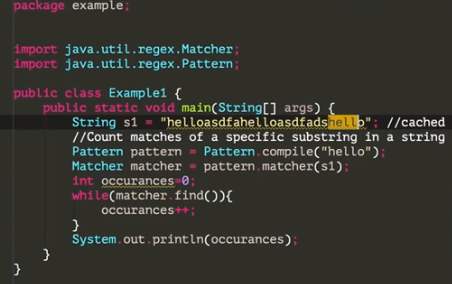
String obj1 = "Samuel";

in the first case obj==obj1 returns false

in the second case obj==obj1 returns true.

The reason for that is that in the first case you have two references to two different Objects. In the second case you have one object since Strings are immutable they are [interned](https://en.wikipedia.org/wiki/String_interning) and drawn from the same pool.

Finding the substring in String?



As String manipulations are resource consuming, Java provides two utility classes:

*StringBuffer* S*tringBuilder*.

Let us understand the difference between these two utility classes:

* StringBuffer and StringBuilder are mutable classes. StringBuffer operations are thread-safe and synchronized whereas StringBuilder operations are not thread-safe.
* StringBuffer is to be used when multiple threads are working on same String and StringBuilder in the single threaded environment.
* StringBuilder performance is faster when compared to StringBuffer because of no overhead of synchronized.
* **how-to-create-immutable-class-in-java?**

<https://www.journaldev.com/129/how-to-create-immutable-class-in-java>

**Immutable objects** are instances whose state doesn’t change after it has been initialized. For example, [String](https://www.journaldev.com/16928/java-string) is an immutable class and once instantiated its value never changes.

An immutable class is good for caching purpose because you don’t need to worry about the value changes. Other benefit of immutable class is that it is inherently [**thread-safe**](https://www.journaldev.com/1061/thread-safety-in-java), so you don’t need to worry about thread safety in case of multi-threaded environment.

To create an immutable class in java, you have to do following steps.

1. Declare the class as final so it can’t be extended.
2. Make all fields private so that direct access is not allowed.
3. Don’t provide setter methods for variables
4. Make all **mutable fields final** so that it’s value can be assigned only once.
5. Initialize all the fields via a constructor performing deep copy.
6. Perform cloning of objects in the getter methods to return a copy rather than returning the actual object reference.

public final class FinalClassExample {

private final int id;

private final String name;

private final HashMap<String,String> testMap;

public int getId() {

return id;

}

public String getName() {

return name;

}

/\*\*

\* Accessor function for mutable objects

\*/

public HashMap<String, String> getTestMap() {

//return testMap;

return (HashMap<String, String>) testMap.clone();

}

/\*\*

\* Constructor performing Deep Copy

\* @param i

\* @param n

\* @param hm

\*/

public FinalClassExample(int i, String n, HashMap<String,String> hm){

System.out.println("Performing Deep Copy for Object initialization");

this.id=i;

this.name=n;

HashMap<String,String> tempMap=new HashMap<String,String>();

String key;

Iterator<String> it = hm.keySet().iterator();

while(it.hasNext()){

key=it.next();

tempMap.put(key, hm.get(key));

}

this.testMap=tempMap;

}

/\*\*

\* Constructor performing Shallow Copy. We should not do shallow copy incase of immutable class.

\* @param i

\* @param n

\* @param hm

\*/

/\*\*

public FinalClassExample(int i, String n, HashMap<String,String> hm){

System.out.println("Performing Shallow Copy for Object initialization");

this.id=i;

this.name=n;

this.testMap=hm;

}

\*/

}

What about Date variable we can still change the date by writing  
d.getTime();

How can we avoid that?

public final class Sample  
{

private final Date dateField;

//Default private constructor will ensure no unplanned construction of class  
private Sample(Date date)  
{  
this.dateField = new Date(date.getTime());  
}

/\*\*  
\* Date class is mutable, so we need a little care here.  
\* We should not return the reference of original instance variable.  
\* Instead of a new Date object, with content copied to it, should be returned.  
\* \*/  
public Date getDateField() {  
return new Date(dateField.getTime());  
}

}

Volatile variable?

the volatile keyword in Java is used as an indicator to Java compiler and Thread that do not cache value of this variable and always read it from [main memory](http://javarevisited.blogspot.com/2011/05/java-heap-space-memory-size-jvm.html).

[double checked locking in Singleton](http://javarevisited.blogspot.com/2014/05/double-checked-locking-on-singleton-in-java.html)

**public** **class** **Singleton**{

**private** **static** **volatile** Singleton \_instance; //volatile variable

**public** **static** Singleton **getInstance**(){

**if**(\_instance == **null**){

**synchronized**(Singleton.class){

**if**(\_instance == **null**)

\_instance = **new** Singleton();

}

}

**return** \_instance;

}

Synchronized method(object level lock) and Synchronized block (class level lock(static))

**Synchronized method** is used to lock an object for any shared resource. When a thread invokes a **synchronized method**, it automatically acquires the lock for that object and releases it when the thread completes its task.

synchronized public void getLine()

    {

        for (int i = 0; i < 3; i++)

        {

            System.out.println(i);

            try

            {

                Thread.sleep(400);

            }

            catch (Exception e)

            {

                System.out.println(e);

            }

        }

    }

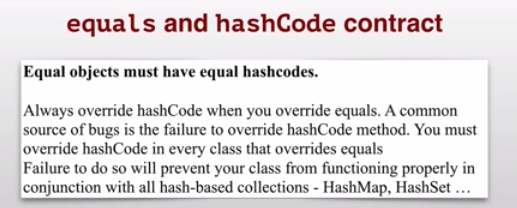
1. Synchronous:

If an API call is synchronous, it means that code execution will block (or wait) for the API call to return before continuing. This means that until a response is returned by the API, your application will not execute any further, which could be perceived by the user as latency or performance lag in your app. Making an API call synchronously can be beneficial, however, if there if code in your app that will only execute properly once the API response is received.

1. Asynchronous:

Asynchronous calls do not block (or wait) for the API call to return from the server. Execution continues on in your program, and when the call returns from the server, a "callback" function is executed.

**Equals and hashcode Contract:**

****

# [How Generics in Java works with Example of Collections, Best practices, Gotchas](https://javarevisited.blogspot.com/2011/09/generics-java-example-tutorial.html)

A generic type is a generic class or interface that is parameterized over types. The following Box class will be modified to demonstrate the concept.

public class Box {

private Object object;

public void set(Object object) { this.object = object; }

public Object get() { return object; }

}

The type parameter section, delimited by angle brackets (<>), follows the class name.

/\*\*

\* Generic version of the Box class.

\* @param <T> the type of the value being boxed

\*/

public class Box<T> {

// T stands for "Type"

private T t;

public void set(T t) { this.t = t; }

public T get() { return t; }

}

type parameter names are single, uppercase letters.

The most commonly used type parameter names are:

* E - Element (used extensively by the Java Collections Framework)
* K - Key
* N - Number
* T - Type
* V - Value
* S,U,V etc. - 2nd, 3rd, 4th types

Generic in Java is added to provide compile time **type-safety of code** and removing risk of ClassCastException at [runtime](http://javarevisited.blogspot.sg/2012/03/what-is-static-and-dynamic-binding-in.html) which was quite frequent error in Java code.

# Wildcards

In generic code, the question mark (?), called the wildcard, represents an unknown type. The wildcard can be used in a variety of situations: as the type of a parameter, field, or local variable; sometimes as a return type (though it is better programming practice to be more specific). The wildcard is never used as a type argument for a generic method invocation, a generic class instance creation, or a supertype.

# [Java: difference between strong/soft/weak/phantom reference](https://stackoverflow.com/questions/9809074/java-difference-between-strong-soft-weak-phantom-reference)

Java provides two different types/classes of *Reference Objects*: **strong** and **weak**. Weak Reference Objects can be further divided into *soft* and *phantom*. Let's go point by point.

**Strong Reference Object**

StringBuilder builder = new StringBuilder();

This is the default type/class of Reference Object, if not differently specified: builder is a strong Reference Object. This kind of reference makes the referenced object not eligible for GC. That is, whenever an object is referenced by a *chain of strong Reference Objects*, it cannot be garbage collected.

**Weak Reference Object**

WeakReference<StringBuilder> weakBuilder = new WeakReference<StringBuilder>(builder);

Weak Reference Objects are not the default type/class of Reference Object and to be used they should be explicitly specified like in the above example. This kind of reference makes the reference object eligible for GC. That is, in case the only reference reachable for the StringBuilder object in memory is, actually, the weak reference, then the GC is allowed to garbage collect the StringBuilder object. When an object in memory is reachable only by Weak Reference Objects, it becomes automatically eligible for GC.

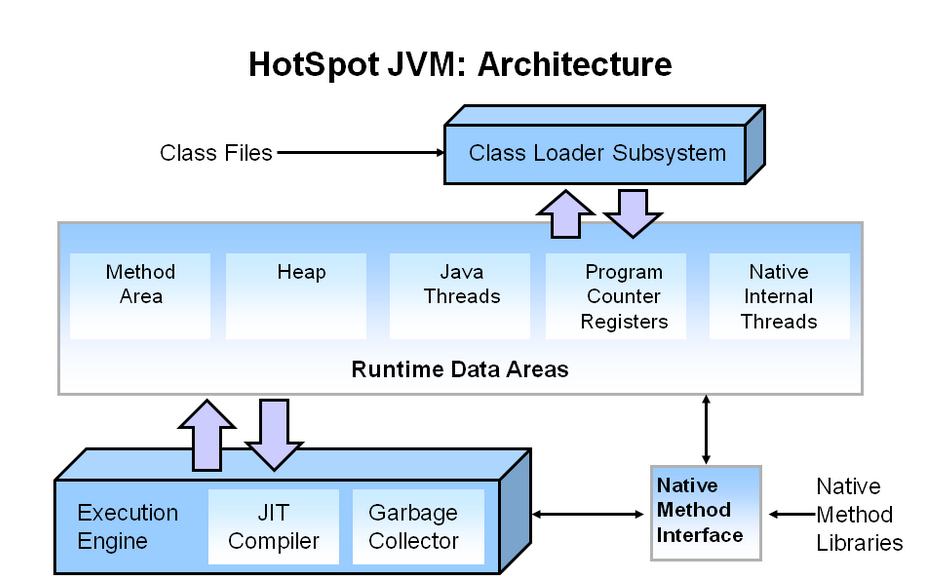
**Levels of Weakness**

Two different levels of weakness can be enlisted: soft and phantom.

A soft Reference Object is basically a weak Reference Object that remains in memory a bit more: normally, it resists GC cycle until memory is available and there is no risk of OutOfMemoryError (in that case, it can be removed).

On the other hand, a phantom Reference Object is useful only to know exactly when an object has been effectively removed from memory: normally they are used to fix weird finalize() revival/resurrection behavior, since they actually do not return the object itself but only help [in keeping track of their memory presence](https://community.oracle.com/blogs/enicholas/2006/05/04/understanding-weak-references).

<https://howtodoinjava.com/java/garbage-collection/all-garbage-collection-algorithms/>



<https://www.geeksforgeeks.org/mark-and-sweep-garbage-collection-algorithm/>

# Mark-and-Sweep: Garbage Collection Algorithm

[Garbage collection algorithms](https://howtodoinjava.com/java/garbage-collection/all-garbage-collection-algorithms/#gc-algorithms)

1. [Mark and sweep](https://howtodoinjava.com/java/garbage-collection/all-garbage-collection-algorithms/#mark-sweep)
2. [Concurrent mark sweep (CMS) garbage collection](https://howtodoinjava.com/java/garbage-collection/all-garbage-collection-algorithms/#cms)
3. [Serial garbage collection](https://howtodoinjava.com/java/garbage-collection/all-garbage-collection-algorithms/#serial-gc)
4. [Parallel garbage collection](https://howtodoinjava.com/java/garbage-collection/all-garbage-collection-algorithms/#parallel-gn)
5. [G1 garbage collection](https://howtodoinjava.com/java/garbage-collection/all-garbage-collection-algorithms/#g1-gc)

All the objects which are created dynamically (using new in C++ and Java) are allocated memory in the heap. If we go on creating objects we might get Out Of Memory error, since it is not possible to allocate heap memory to objects. So we need to clear heap memory by releasing memory for all those objects which are no longer referenced by the program (or the unreachable objects) so that the space is made available for subsequent new objects. This memory can be released by the programmer itself but it seems to be an overhead for the programmer, here garbage collection comes to our rescue, and it automatically releases the heap memory for all the unreferenced objects.

**Mark and Sweep Algorithm**  
Any garbage collection algorithm must perform 2 basic operations. One, it should be able to detect all the unreachable objects and secondly, it must reclaim the heap space used by the garbage objects and make the space available again to the program.  
The above operations are performed by Mark and Sweep Algorithm in two phases:  
1) Mark phase  
2) Sweep phase

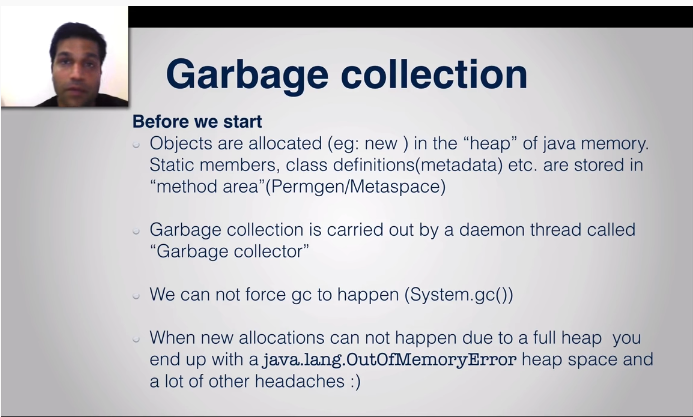
**Mark Phase**  
When an object is created, its mark bit is set to 0(false). In the Mark phase, we set the marked bit for all the reachable objects (or the objects which a user can refer to) to 1(true). Now to perform this operation we simply need to do a graph traversal, a [depth first search approach](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/) would work for us.

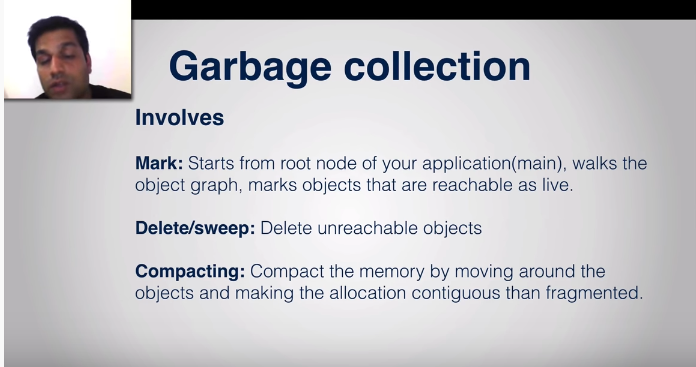
**Sweep Phase**  
As the name suggests it “sweeps” the unreachable objects i.e. it clears the heap memory for all the unreachable objects. All those objects whose marked value is set to false are cleared from the heap memory, for all other objects (reachable objects) the marked bit is set to true.

**Garbage Collection by Ranjith Ramachandran:**

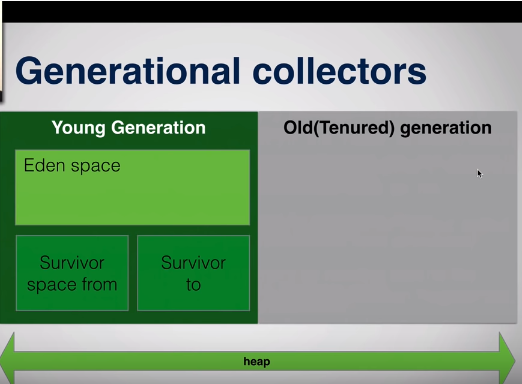
Basic hypothesis:

* Most objects soon become unreachable in application context.
* References from old objects to young object only exist in small number.
* Live object = reachable
* Dead object = unreachable



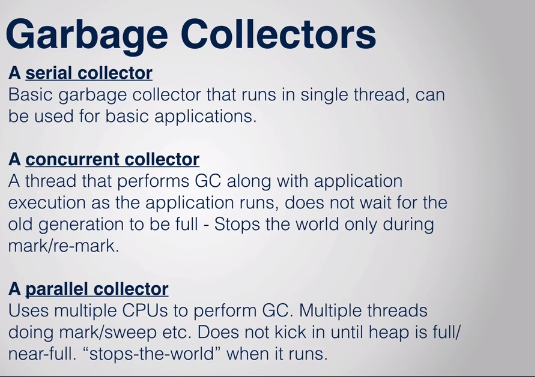


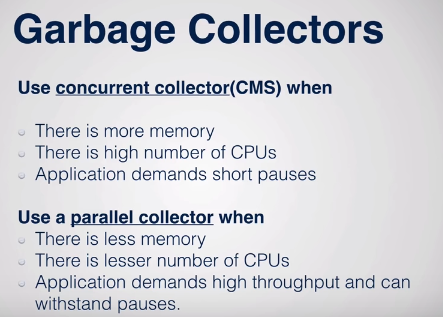
* Minor(run in Eden space) and major(whole heap) GC which are called as stop world GC.

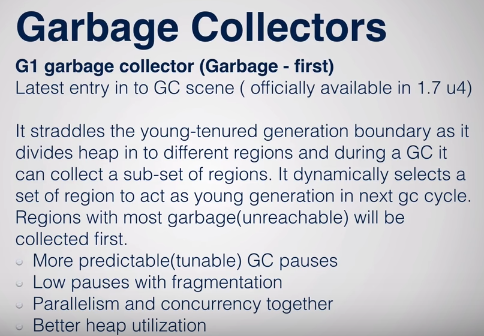


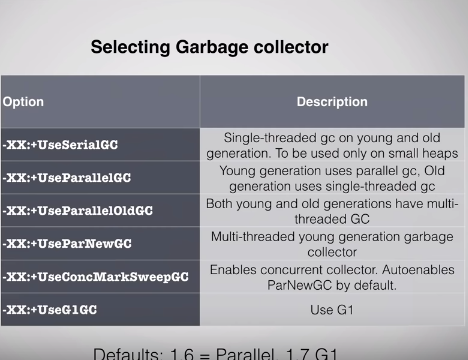


* -XX:MaxTenuringThreshold used to move object from YG to OG.
* Performances : 1. Latency/ Responses 2. Throughput





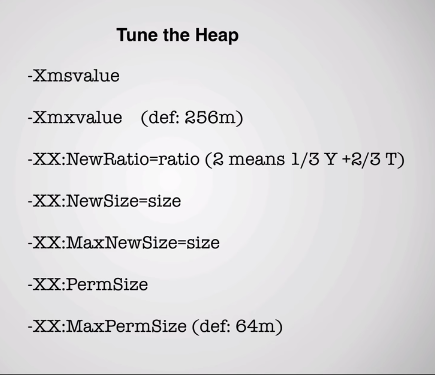


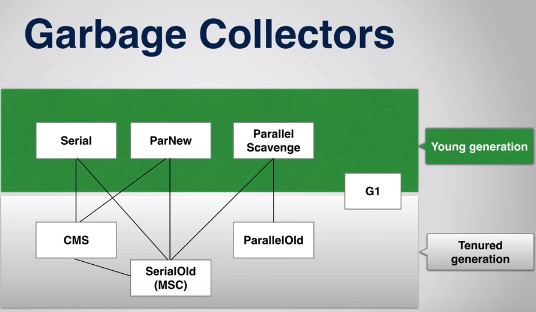


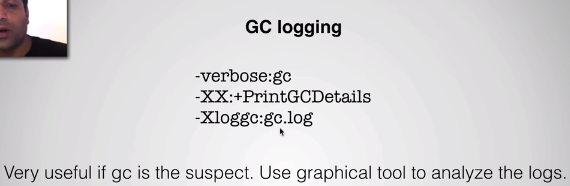
* Finalizer: protected void finalizer() throws Throwable . Is called when GC is called,

There is no granted when GC is called, so don’t use Finalizer.

Finalizer method is called only once for object. When we call new object creation in finalizer() method it created the object.







* GC Analyze tools: GC analyze tool from IBM(comes eclipse plugin), jvisualvm , Jhat and Jmap

H:\>jvisualvm

C:\Program Files\Java\jdk1.8.0\_111\bin>jmap -dump:file=C://sachin/dump.hprof 6620

Dumping heap to C:\sachin\dump.hprof ...

Heap dump file created

C:\Program Files\Java\jdk1.8.0\_111\bin>jhat C://sachin/dump.hprof

Reading from C://sachin/dump.hprof...

Dump file created Mon Dec 02 17:42:08 IST 2019

Snapshot read, resolving...

Resolving 5688267 objects...

Snapshot resolved.

Started HTTP server on port 7000

Server is ready.

Atomic in Java:

# The [java.util.concurrent.atomic](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html) 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. That is, a set has a happens-before relationship with any subsequent get on the same variable. The atomic compareAndSet method also has these memory consistency features, as do the simple atomic arithmetic methods that apply to integer atomic variables.

# AtomicInteger

private AtomicInteger counter = new AtomicInteger();

public int getNextUniqueIndex() {

return counter.getAndIncrement();

}

The AtomicInteger class uses CAS ([compare-and-swap](http://en.wikipedia.org/wiki/Compare-and-swap)) low-level CPU operations (no synchronization needed!) They allow you to modify a particular variable only if the present value is equal to something else (and is returned successfully). So when you execute getAndIncrement() it actually runs in a loop (simplified real implementation):

<https://docs.oracle.com/javase/tutorial/essential/concurrency/atomicvars.html>

import java.util.concurrent.atomic.AtomicInteger;

class AtomicCounter {

private AtomicInteger c = new AtomicInteger(0);

public void increment() {

c.incrementAndGet();

}

public void decrement() {

c.decrementAndGet();

}

public int value() {

return c.get();

}

}

# [Difference between Association, Composition and Aggregation in Java, UML and Object Oriented Programming](https://javarevisited.blogspot.com/2014/02/ifference-between-association-vs-composition-vs-aggregation.html)

In Object-oriented programming, one object is related to other to use functionality and service provided by that object. This relationship between two objects is known as the *association* in object oriented general software design and depicted by an arrow in Unified Modelling language or UML. Both Composition and Aggregation are the form of association between two objects, but there is a **subtle difference between composition and aggregation**

**Composition:** Since Engine is-part-of Car, the relationship between them is Composition. Here is how they are implemented between Java classes.

**public** **class** **Car** {

//final will make sure engine is initialized

**private** **final** Engine engine;

**public** **Car**(){

engine = **new** Engine();

}

}

**class** **Engine** {

**private** String type;

}

**Aggregation :** Since Organization has Person as employees, the relationship between them is Aggregation. Here is how they look like in terms of Java classes

**public** **class** **Organization** {

**private** List employees;

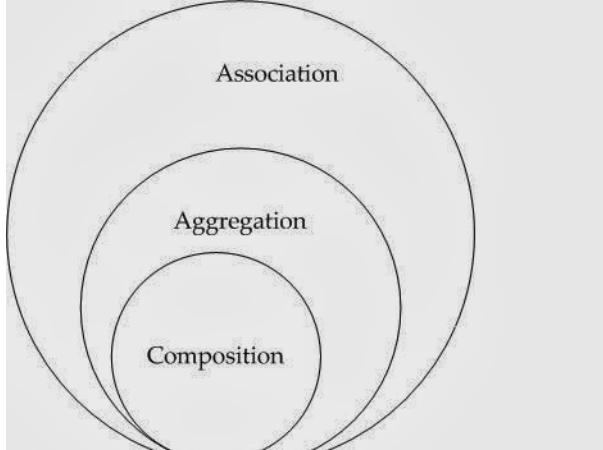
}

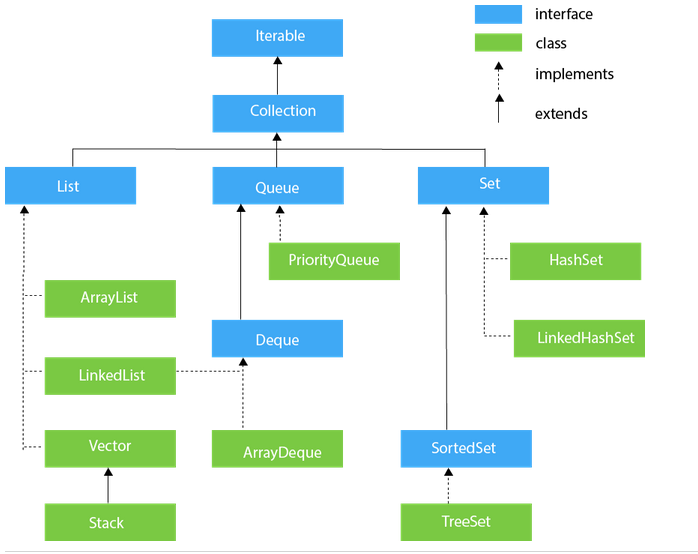
**public** **class** **Person** {

**private** String name;

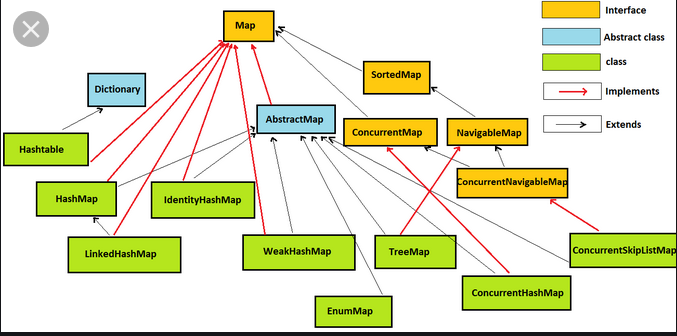
}

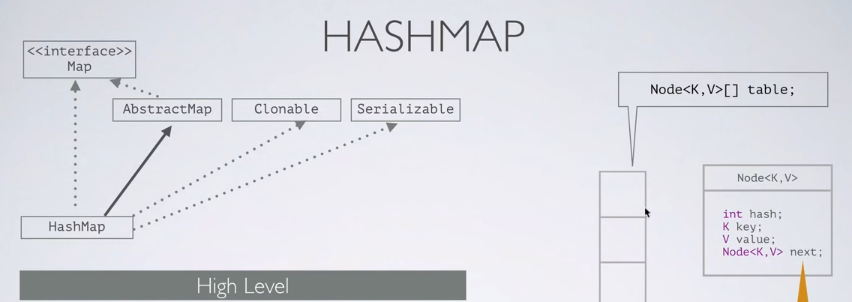
Association  A---->B  
Composition  A-----<filled>B  
Aggregation  A-----<>B





Main *difference between Array vs ArrayList in Java* is static nature of Array and dynamic nature of ArrayList. Once created you can not change size of Array but ArrayList can re-size itself when needed. Another notable difference between ArrayList and Array is that Array is part of core Java programming and has special syntax and semantics support in Java, While ArrayList is part of [Collection framework](http://javarevisited.blogspot.de/2011/11/collection-interview-questions-answers.html)

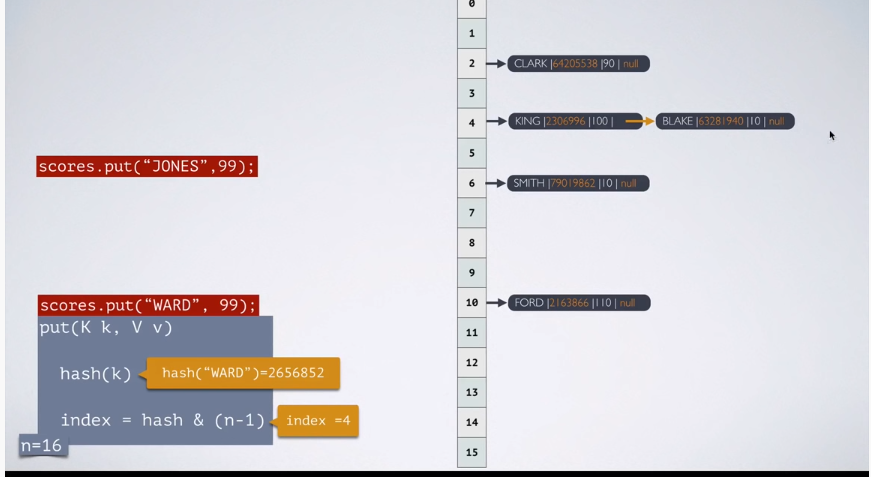




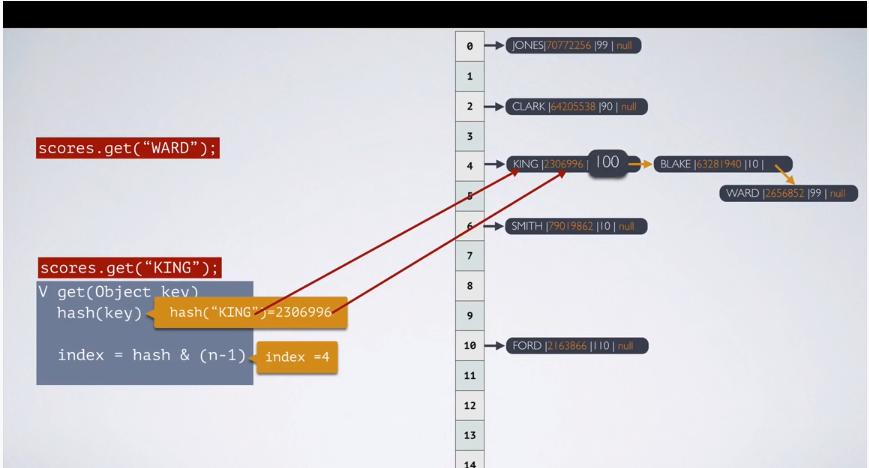
HashMap table is sized with 2^n buckets.

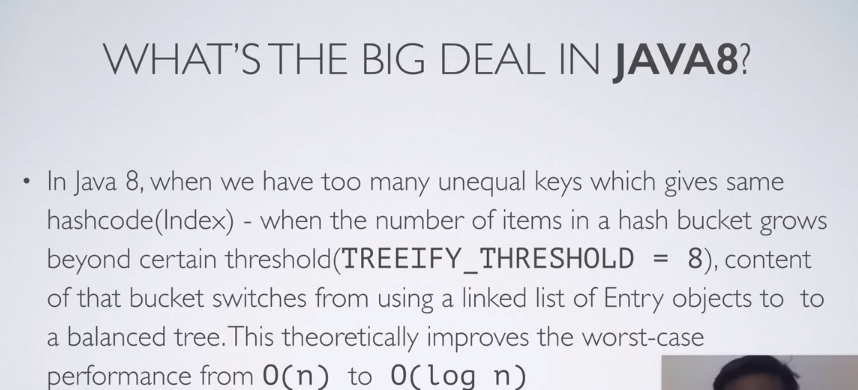
**HashMap** doesn't **allow duplicate keys** but **allows duplicate** values. That means A single **key** can't contain more than 1 value but more than 1 **key** can contain a single value. **HashMap allows** null **key** also but only once and multiple null values.

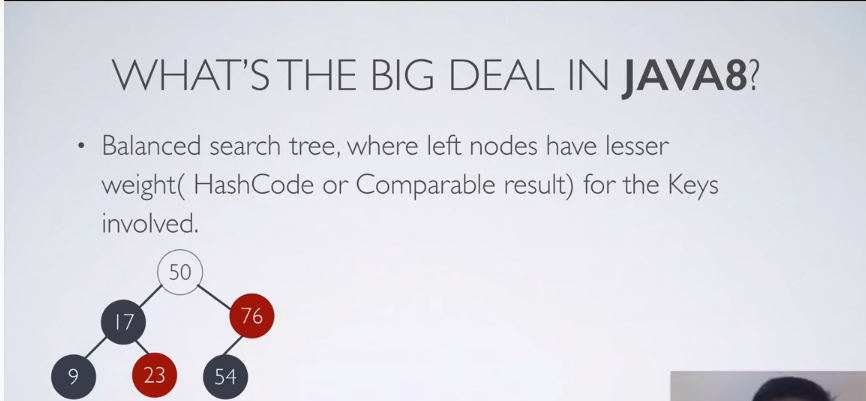
Put operation:

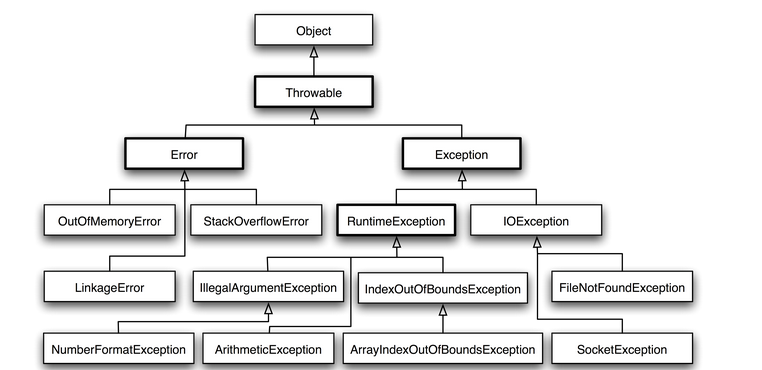


Get Operation:









public Circle(double radius) {

if (radius < 0.0) {

throw new IllegalArgumentException("Radius " + radius

+ " cannot be negative");

}

this.radius = radius;

}

Checked

extends Exception

Unchecked

extends RuntimeException

public class InsufficientBalanceException extends Exception {

private final double available;

private final double required;

/\*\*

\* Constructor.

\* @param available available account balance

\* @param required required account balance

\*/

public InsufficientBalanceException(double available, double required) {

super("Available $"+available+" but required $"+required);

this.available = available;

this.required = required;

}

/\*\*

\* Get available account balance

\* @return available account balance

\*/

public double getAvailable() {

return available;

}

/\*\*

\* Get required account balance

\* @return required account balance

\*/

public double getRequired() {

return required;

}

/\*\*

\* Get the difference between required and available account balances

\* @return required - available

\*/

public double getDifference() {

return required - available;

}

}

<https://www.geeksforgeeks.org/deep-shallow-lazy-copy-java-examples/>

* **deep-shallow-lazy-copy-java-examples**

There are several ways to copy an object, most commonly by a [copy constructor](https://www.geeksforgeeks.org/copy-constructor-in-java/) or [cloning](https://www.geeksforgeeks.org/clone-method-in-java-2/).

1. Copy Constructor:

Java doesn’t create a default copy constructor if you don’t write your own.

// copy constructor

    Complex(Complex c) {

        System.out.println("Copy constructor called");

        re = c.re;

        im = c.im;

    }

public static void main(String[] args) {

        Complex c1 = new Complex(10, 15);

        // Following involves a copy constructor call

        Complex c2 = new Complex(c1);

        // Note that following doesn't involve a copy constructor call as

        // non-primitive variables are just references.

        Complex c3 = c2;

**}**

1. **Cloning:**

# Clone() method in Java

Object cloning refers to creation of exact copy of an object. It creates a new instance of the class of current object and initializes all its fields with exactly the contents of the corresponding fields of this object.

We can define Cloning as **“create a copy of object”** Shallow, deep and lazy copy is related to cloning process

The class whose object’s copy is to be made must have a public clone method in it or in one of its parent class.

* Every class that implements clone() should call super.clone() to obtain the cloned object reference.
* The class must also implement java.lang.Cloneable interface whose object clone we want to create otherwise it will throw CloneNotSupportedException when clone method is called on that class’s object.
* Syntax:

protected Object clone() throws CloneNotSupportedException

**Usage of clone() method -Shallow Copy**

// An object reference of this class is

// contained by Test2

class Test

{

int x, y;

}

// Contains a reference of Test and implements

// clone with shallow copy.

class Test2 implements Cloneable

{

int a;

int b;

Test c = new Test();

public Object clone() throws

CloneNotSupportedException

{

return super.clone();

}

}

// Driver class

public class Main

{

public static void main(String args[]) throws

CloneNotSupportedException

{

Test2 t1 = new Test2();

t1.a = 10;

t1.b = 20;

t1.c.x = 30;

t1.c.y = 40;

Test2 t2 = (Test2)t1.clone();

// Creating a copy of object t1 and passing

// it to t2

t2.a = 100;

// Change in primitive type of t2 will not

// be reflected in t1 field

t2.c.x = 300;

// Change in object type field will be

// reflected in both t2 and t1(shallow copy)

System.out.println(t1.a + " " + t1.b + " " +

t1.c.x + " " + t1.c.y);

System.out.println(t2.a + " " + t2.b + " " +

t2.c.x + " " + t2.c.y);

}

}

**//code illustrating shallow copy**

**public class Ex {**

**private int[] data;**

**// makes a shallow copy of values**

**public Ex(int[] values) {**

**data = values;**

**}**

**public void showData() {**

**System.out.println( Arrays.toString(data) );**

**}**

**}**

* **Shallow copy** is method of copying an object and is followed by default in cloning. In this method the fields of an old object X are copied to the new object Y. While copying the object type field the reference is copied to Y i.e object Y will point to same location as pointed out by X. If the field value is a primitive type it copies the value of the primitive type.
* Therefore, any changes made in referenced objects in object X or Y will be reflected in other object.

**Usage of clone() method – Deep Copy**

* If we want to create a deep copy of object X and place it in a new object Y then new copy of any referenced objects fields are created and these references are placed in object Y. This means any changes made in referenced object fields in object X or Y will be reflected only in that object and not in the other. In below example, we create a deep copy of object.
* A deep copy copies all fields, and makes copies of dynamically allocated memory pointed to by the fields. A deep copy occurs when an object is copied along with the objects to which it refers.

class Test

{

    int x, y;

}

// Contains a reference of Test and implements

// clone with deep copy.

class Test2 implements Cloneable

{

    int a, b;

    Test c = new Test();

    public Object clone() throws

                CloneNotSupportedException

    {

        // Assign the shallow copy to new reference variable t

        Test2 t = (Test2)super.clone();

        t.c = new Test();

        // Create a new object for the field c

        // and assign it to shallow copy obtained,

        // to make it a deep copy

        return t;

    }

}

**// Code explaining deep copy**

**public class Ex {**

**private int[] data;**

**// altered to make a deep copy of values**

**public Ex(int[] values) {**

**data = new int[values.length];**

**for (int i = 0; i < data.length; i++) {**

**data[i] = values[i];**

**}**

**}**

**public void showData() {**

**System.out.println(Arrays.toString(data));**

**}**

**}**

**when to use what**  
There is no hard and fast rule defined for selecting between shallow copy and deep copy but normally we should keep in mind that if an object has only primitive fields, then obviously we should go for shallow copy, but if the object has references to other objects, then based on the requirement, shallow copy or deep copy should be done. If the references are not updated then there is no point to initiate a deep copy.

**Lazy Copy**  
A lazy copy can be defined as a combination of both shallow copy and deep copy. The mechanism follows a simple approach – at the initial state, shallow copy approach is used. A counter is also used to keep a track on how many objects share the data. When the program wants to modify the original object, it checks whether the object is shared or not. If the object is shared, then the deep copy mechanism is initiated.

**Comparable** is an [interface](https://www.edureka.co/blog/java-interface/) which defines a way to compare an object with other objects of the same type. It helps to sort the objects that have self-tendency to sort themselves, i.e., the objects must know how to order themselves.

@Override

public int compareTo(Student per) {

if(this.age == per.age)

return 0;

else

return this.age > per.age ? 1 : -1;

}

This method is used to compare the given object with the current object. The **compareTo()** method returns an int value. The value can be either positive, negative, or zero.

A Comparator interface is used to order the objects of a specific class. This interface is found in java.util package. It contains two methods;

* compare(Object obj1,Object obj2)
* equals(Object element).

## **Comparable v/s Comparator in Java**

|  |  |
| --- | --- |
| Comparable in Java | Comparator in Java |
| Comparable interface is used to sort the objects with natural ordering. | Comparator in Java is used to sort attributes of different objects. |
| Comparable interface compares “this” reference with the object specified. | Comparator in Java compares two different class objects provided. |
| Comparable is present in java.lang package. | A Comparator is present in the java.util package. |
| Comparable affects the original class, i.e., the actual class is modified. | Comparator doesn’t affect the original class |
| Comparable provides compareTo() method to sort elements. | Comparator provides compare() method, equals() method to sort elements. |