

GENERAL QUESTIONS .....	2
OOPS CONCEPTS .....	5
PROGRAMMING FUNDAMENTALS.....	27
EXCEPTION HANDLING .....	65
GARBAGE COLLECTION .....	77
GENERICS .....	94
STRING.....	103
INNER CLASS .....	112
CLASS LOADER.....	127
COLLECTION .....	127
ANNOTATION.....	200
REFLECTION API .....	209
ENUM .....	214
ENUMERATION.....	219
CLONEABLE .....	220
MARKER INTERFACE .....	230
WRAPPER .....	231
SERILAZATION .....	238
RTTI IN JAVA.....	253
JAVA 7 .....	257
JAVA 8.....	257
THREAD .....	257
JDBC .....	260
IMPORTANT POINTS .....	262
javaProgrammingPointsInterview(SCJP) .....	<b>Error! Bookmark not defined.</b>
JAVA ARCHITECTURE.....	283
JVM INTERNALS .....	290
MEMORY MODEL .....	317
PROGRAMMING QUESTION.....	319

## GENERAL QUESTIONS

### 1) What is java?

Java is object oriented, platform independent, high-level programming language from the Sun Microsystem. It is released in the year 1995 and runs on multiple platforms like Windows, Mac and other variations of UNIX.

### 2) Difference between J2SDK 1.5 and J2SDK 5.0?

No difference, Sun has rebranded the version.

### 3) Difference between JDK AND JRE AND JVM?

**JVM (Java Virtual Machine):** It is an abstract machine. It is a specification that provides run-time environment in which java bytecode can be executed. It follows three notations:

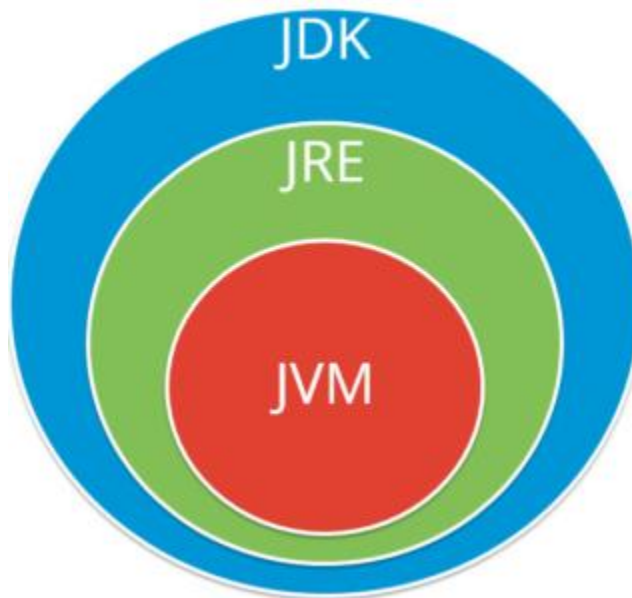
**Specification:** It is a document that describes the implementation of the Java virtual machine. It is provided by Sun and other companies.

**Implementation:** It is a program that meets the requirements of JVM specification.

**Runtime Instance:** An instance of JVM is created whenever you write a java command on the command prompt and run the class.

**JRE (Java Runtime Environment) :** JRE refers to a runtime environment in which java bytecode can be executed. It implements the JVM (Java Virtual Machine) and provides all the class libraries and other support files that JVM uses at runtime. So JRE is a software package that contains what is required to run a Java program. Basically, it's an implementation of the JVM which physically exists.

**JDK(Java Development Kit) :** It is the tool necessary to compile, document and package Java programs. The JDK completely includes JRE which contains tools for Java programmers. The Java Development Kit is provided free of charge. Along with JRE, it includes an interpreter/loader, a compiler (javac), an archiver (jar), a documentation generator (javadoc) and other tools needed in Java development. In short, it contains JRE + development tools. Refer to this below image and understand how exactly these components reside:



#### 4) Why java is platform neutral?

The main reason is because the compiler creates an architecture-natural object file format. This makes the compiled code to be executable on different other processors. Platform independence means that you can run the same Java Program in any Operating System. For example, you can write java program in Windows and run it in Mac OS.

#### 5) What is JVM and is it platform independent?

Java Virtual Machine (JVM) is the heart of java programming language. JVM is responsible for converting byte code into machine readable code. JVM is not platform independent, that's why you have different JVM for different operating systems. We can customize JVM with Java Options, such as allocating minimum and maximum memory to JVM. It's called virtual because it provides an interface that doesn't depend on the underlying OS.

## 6) Why Java is not pure Object Oriented language?

Java is not said to be pure object oriented because it support primitive types such as int, byte, short, long etc.

As we know, for all the primitive types we have wrapper classes such as Integer, Long etc that provides some additional methods.

## 7) Difference between PATH, CLASSPATH, JAVA\_HOME environment variables?

PATH is an environment variable used by operating system to locate the executables. That's why when we install Java or want any executable to be found by OS, we need to add the directory location in the PATH variable. If you work on Windows OS, read this post to learn [how to setup PATH variable on Windows](#).

Classpath is specific to java and used by java executables to locate class files. We can provide the classpath location while running java application and it can be a directory, ZIP files, JAR files etc.

## 8) Can you tell me the number of bits used to represent Unicode, ASCII, UTF-16, and UTF-8 characters?

For Unicode 16 bits and ASCII needs 7 bits. However, ASCII is usually represented as 8 bits. UTF-8 presents characters through 8, 16 and 18 bit pattern. UTF-16 will require 16-bit and larger bit patterns.

## 9) Java Compiler is stored in JDK, JRE or JVM?

The task of java compiler is to convert java program into bytecode, we have `javac` executable for that. So it must be stored in JDK, we don't need it in JRE and JVM is just the specs.

<https://www.quora.com/If-every-class-in-Java-extends-object-class-and-then-a-user-can-extend-it-with-any-other-class-doesnt-it-make-multiple-inheritance-applicable-in-Java>

## 10) Type of programming language?

High level and low level

Compiled vs interpreted

OOP vs functional programming

<https://www.codenewbie.org/blogs/object-oriented-programming-vs-functional-programming>

## 11) Compiled vs Interpreted Languages?

## OOPS CONCEPTS

### 12) What is OOP?

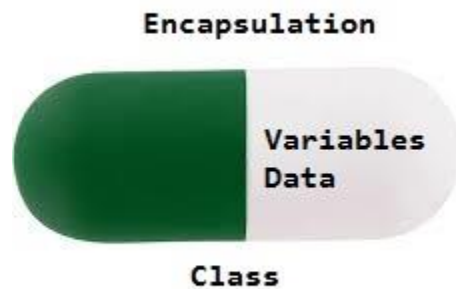
Object-oriented programming or popularly known as OOPs is a programming model or approach where the programs are organized around objects rather than logic and functions. In other words, OOP mainly focuses on the objects that are required to be manipulated instead of logic. This approach is ideal for the programs large and complex codes and needs to be actively updated or maintained.

### 13) Oops concepts:

- Abstraction
- Encapsulation
- Polymorphism
- Inheritance
- Association
- Aggregation
- Composition

### 14) What is encapsulation and how it is achieved in java?

Encapsulation simply means binding object state(fields) and behaviour(methods) together. it is a protective shield that prevents the data from being accessed by the code outside this shield. The whole idea behind encapsulation is to hide the implementation details from users. If a data member is private it means it can only be accessed within the same class. No outside class can access private data member (variable) of other class. Encapsulation is also known as DATA HIDING.



Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of own class in which they are declared.

As in encapsulation, the data in a class is hidden from other classes, so it is also known as **data-hiding**.

How to implement encapsulation in java:

- 1) Make the instance variables private so that they cannot be accessed directly from outside the class. You can only set and get values of these variables through the methods of the class.
- 2) Have getter and setter methods in the class to set and get the values of the fields.

Advantage:

- Ease of maintains
- We can make fields read only by giving only getter
- You can protect data to corrupt using set methods
- Hide the details from the user

## 15) What is abstraction and how it is used in java?

Data Abstraction is the property by virtue of which only the essential details are displayed to the user. The trivial or the non-essentials units are not displayed to the user. Ex: A car is viewed as a car rather than its individual components.

Data Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details. The properties and behaviors of an object differentiate it from other objects of similar type and also help in classifying/grouping the objects.

Abstraction is achieved by abstract classes and interface in java. We can achieve 100% abstraction using interfaces.

## 16) Abstraction vs Encapsulation

Abstraction is generalization, it tells the how to do and declare functionally.

Encapsulation is wrapping data and hiding the implementation to different levels as public, private, default and protected.

Abstraction hides complexity by giving you a more abstract picture, a sort of 10,000 feet view, while Encapsulation hides internal working so that you can change it later. In other words, Abstraction hides details at the **design** level, while Encapsulation hides details at the **implementation** level.

For example, when you first describe an object, you talk in more abstract term e.g. a Vehicle which can move, you don't tell how Vehicle will move, whether it will move by using tires or it will fly or it will sell. It just moves. This is called [Abstraction](#). We are talking about a most essential thing, which is moving, rather than focusing on details like moving in plane, sky, or water.

There are also the different levels of Abstraction and it's good practice that classes should interact with other classes with the same level of abstraction or higher level of abstraction. As you increase the level of Abstraction, things start getting simpler and simpler because you leave out details.

On the other hand, Encapsulation is all about implementation. Its sole purpose is to hide internal working of objects from outside world so that you can change it later without impacting outside clients.


For example, we have a HashMap which allows you to store the object using `put()` method and retrieve the object using the `get()` method. How HashMap implements this method (see [here](#)) is an internal detail of HashMap, the client only cares that `put` stores the object and `get` return it back, they are not concerned whether HashMap is using an array, how it is resolving the collision, whether it is using [linked list](#) or [binary tree](#) to store object landing on same bucket etc.

# JAVA INTERVIEW QUESTIONS GENERIC, INNERCLAS,ENUMSET,EMUMAP,QUEUE, DEQUEUE

abstraction vs encapsulation - G... x Difference between Abstraction - x Difference between Abstraction - x +

https://www.guru99.com/difference-between-abstraction-and-encapsulation.html#7

## Abstraction Vs. Encapsulation




Parameter	Abstraction	Encapsulation
Use for	Abstraction solves the problem and issues that arise at the design stage.	Encapsulation solves the problem and issue that arise at the implementation stage.
Focus	Abstraction allows you to focus on what the object does instead of how it does it	Encapsulation enables you to hide the code and data into a single unit to secure the data from the outside world.
Implementation	You can use abstraction using Interface and Abstract Class.	You can implement encapsulation using Access Modifiers (Public, Protected & Private.)
Focuses	Focus mainly on what should be done.	Focus primarily on how it should be done.
Application	During design level.	During the Implementation level.

### Summary

- Abstraction is an OOP concept that focuses only on relevant data of an object. It hides the background details and emphasizes the essential data points for reducing the complexity and increase efficiency
- Encapsulation is a method of making a complex system more comfortable to handle for end users. The user need not worry about internal details and complexities of the system
- Abstraction helps you to partition the program into many independent notions.

Get a **student-exclusive offer** today.



Learn More  
Online only.

Type here to search

10:12 AM  
7/29/2019



Abstraction	Encapsulation
1. Abstraction solves the problem in the design level.	1. Encapsulation solves the problem in the implementation level.
2. Abstraction is used for hiding the unwanted data and giving relevant data.	2. Encapsulation means hiding the code and data into a single unit to protect the data from outside world.
3. Abstraction lets you focus on what the object does instead of how it does it	3. Encapsulation means hiding the internal details or mechanics of how an object does something.
<p>4. <b>Abstraction</b>- Outer layout, used in terms of design.</p> <p>For Example:-</p> <p>Outer Look of a Mobile Phone, like it has a display screen and keypad buttons to dial a number.</p>	<p>4. <b>Encapsulation</b>- Inner layout, used in terms of implementation.</p> <p>For Example:- Inner Implementation detail of a Mobile Phone, how keypad button and Display Screen are connect with each other using circuits.</p>

## 17) Association, Aggregation, composition

Association - Multiple students can associate with single teacher and single student can associate with multiple teachers but there is no ownership between the objects and both have their own lifecycle. Both can create and delete independently. It can be one to one, one to many, many to many, many to one.

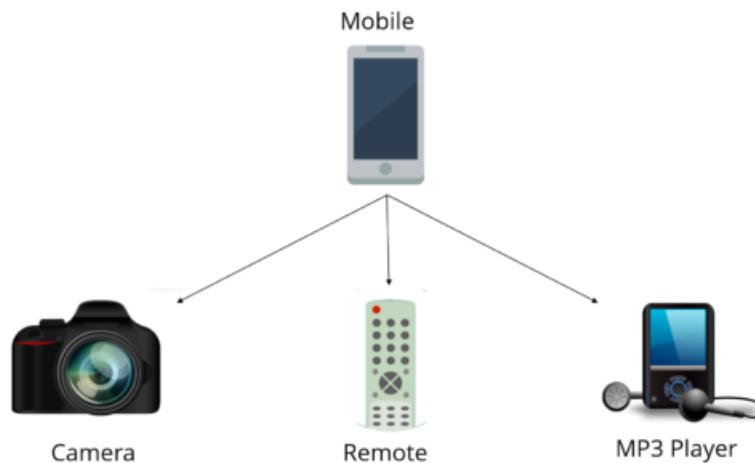
Aggregation - HAS-A (special form of association) ; Aggregation is a specialized form of Association where all object have their own lifecycle but there is ownership and child object cannot belongs to another parent object. Let's take an example of Department and teacher. A single teacher cannot belongs to multiple departments, but if we delete the department teacher object will not destroy.

Composition - LinkedList and Node class (special form of aggregation)  
Composition is again specialized form of Aggregation and we can call this as a "death" relationship. It is a strong type of Aggregation. Child object dose not have their lifecycle and if parent object deletes all child object will also be deleted. Let's take again an example of relationship between House and rooms. House can contain multiple rooms there is no independent life of room and any room can not belongs to two different house if we delete the house room will automatically delete.

Composition is the design technique to implement has-a relationship in classes. We can use Object composition for code reuse. Java composition is achieved by using instance variables that refers to other objects. Benefit of using composition is that we can control the visibility of other object to client classes and reuse only what we need.

Association - > Aggregation -> Composition

## 18) What is Polymorphism?



Polymorphism is briefly described as “one interface, many implementations”. Polymorphism is a characteristic of being able to assign a different meaning or usage to something in different contexts – specifically, to allow an entity such as a variable, a function, or an object to have more than one form.

There are two types of polymorphism:

- Compile time polymorphism/overloading
- Run time polymorphism/overriding/dynamic dispatch

Compile time polymorphism is method overloading whereas Runtime time polymorphism is done using inheritance and interface.

```
package misc;
```

```
class A{
```

```
    public A(){
```

```
        System.out.println("A's Constructor");
```

```
        show();
```

```
        System.out.println("A's Constructor End");
```

```
    }
```

```
    void show(){
```

```
        System.out.println("A");
```

```
    }
```

```
void print(){  
    System.out.println("P");  
}  
}
```

```
class B extends A{
```

```
    int radius;
```

```
    public B(int r){  
        radius = r;  
        System.out.println("B's constructor "+radius);  
    }
```

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

```
    void print(){  
        System.out.println("BC");  
    }
```

```
}
```

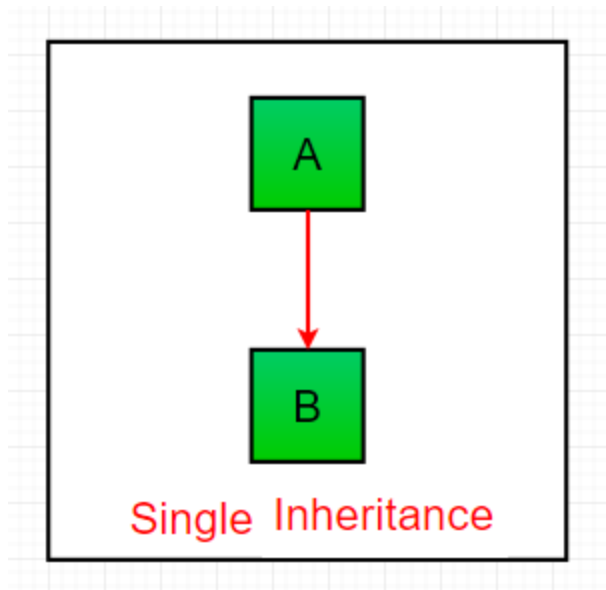
```
public class LateBinding {
```

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

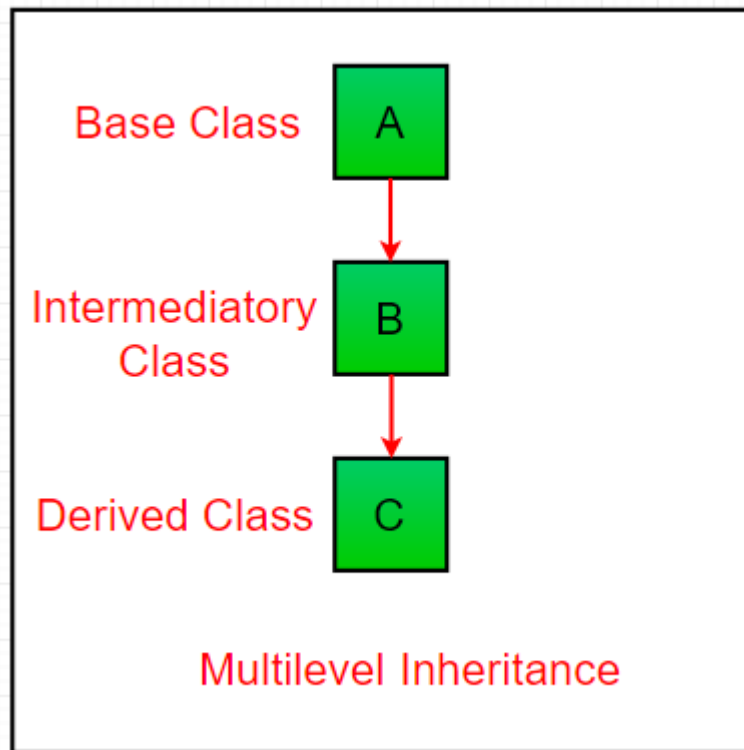
```
        A a = new B(5);  
        a.show();  
        a.print();  
    }  
  
}
```

## 19) Type of inheritance in java?

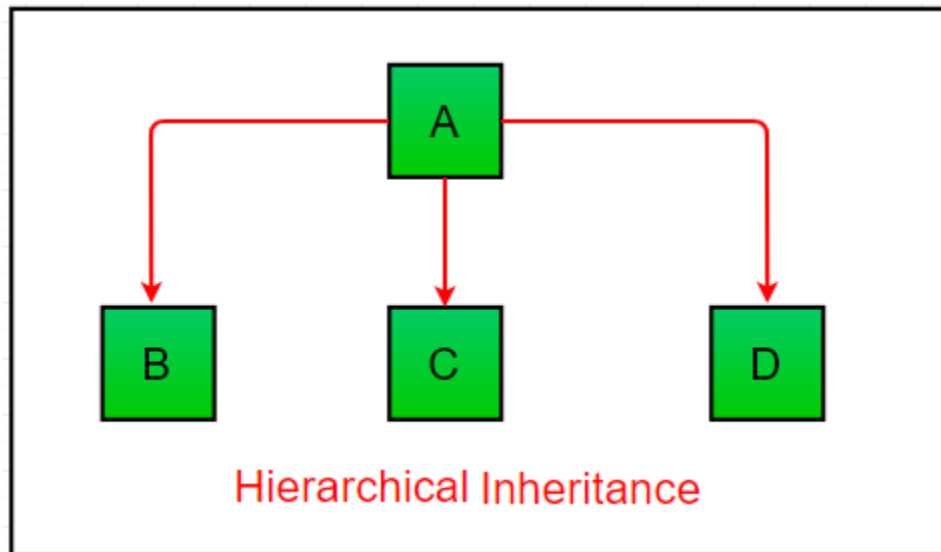
Single Inheritance:



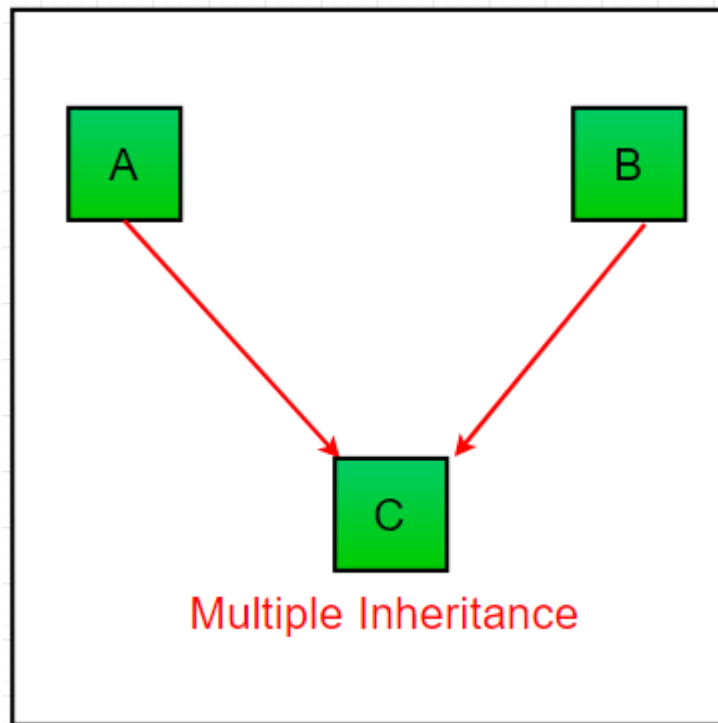
Multilevel Inheritance:



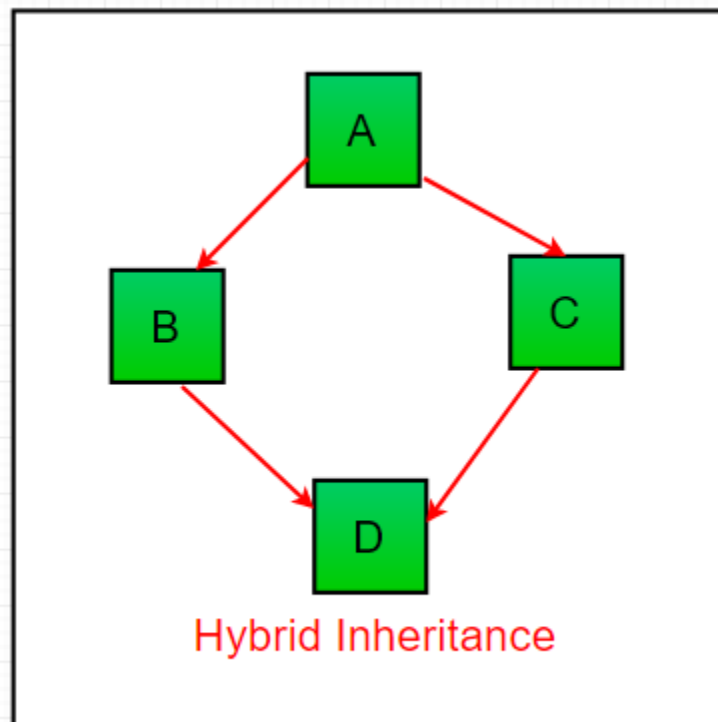
**Hierarchical Inheritance:**



Multiple Inheritance (Through Interfaces) :



Hybrid Inheritance:



<https://www.geeksforgeeks.org/inheritance-in-java/>

## 20) What is method overloading and method overriding?

Method Overloading :

In Method Overloading, Methods of the same class shares the same name but each method must have different number of parameters or parameters having different types and order.

- Method Overloading is to “add” or “extend” more to method’s behavior.
- It is a compile time polymorphism.
- The methods must have different signature.
- It may or may not need inheritance in Method Overloading.

Let’s take a look at the example below to understand it better.

```
class Adder {  
  
    Static int add(int a, int b)  
  
    {  
  
        return a+b;  
  
    }  
  
    Static double add( double a, double b)  
  
    {  
  
        return a+b;  
  
    }  
  
    public static void main(String args[])  
  
    {  
  
        System.out.println(Adder.add(11,11));  
  
        System.out.println(Adder.add(12.3,12.6));  
  
    }  
}
```

Method Overriding:

- In Method Overriding, sub class have the same method with same name and exactly the same number and type of parameters and same return type as a super class.
- Method Overriding is to “Change” existing behavior of method.
- It is a run time polymorphism.
- The methods must have same signature.
- It always requires inheritance in Method Overriding.

Let’s take a look at the example below to understand it better.

```
class Car {  
  
    void run() {
```



```
System.out.println("car is running");  
}  
Class Audi extends Car{  
void run()  
{  
System.out.println("Audi is running safely with 100km");  
}  
public static void main( String args[])  
{  
Car b=new Audi();  
b.run();  
}  
}
```

## 21) Difference between Overloading and overriding

When we have more than one method with same name in a single class but the arguments are different, then it is called as method overloading.

Overriding concept comes in picture with inheritance when we have two methods with same signature, one in parent class and another in child class. We can use @Override annotation in the child class overridden method to make sure if parent class method is changed, so as child class.

- main difference comes from the fact that method overloading is resolved during compile time, while method overriding is resolved at runtime.
- For overriding both name and signature of method must remain same, but in for overloading method signature must be different.
- Last but not the least difference between them is that call to overloaded methods are resolved using static binding while call to overridden method is resolved using dynamic binding in Java.
- Overriding method can not throw higher Exception than original or overridden method. means if original method throws IOException than overriding method cannot throw super class of IOException e.g. Exception but it can throw any sub class of IOException or simply does not throw any Exception.
- This rule only applies to checked Exception in Java, overridden method is free to throw any unchecked Exception.
- A subclass constructor always invokes its parent constructor with a call to super(..). In this case, the parent constructor is declared as throwing a checked exception of type MyException. Your subclass constructor must be able to handle that (with a throws since super(..) has to be the first statement in the constructor body).
- Overriding method cannot reduce accessibility of overridden method, means if original or overridden method is public than overriding method cannot make it protected. Because every instance of the subclass still needs to be a valid instance of the base class
- Methods cant be overloaded on the basis of accessibility. aslo overloading cant be done on the basis of return type.
- Second major difference between method overloading vs overriding in Java is that You can overload method in one class but overriding can only be done on subclass.
- You cannot override static, final and private method in Java but you can overload static, final or private method in Java. Static methods cannot be overridden because method overriding only occurs in the context of dynamic lookup of methods. Static methods are looked up statically.
- One of the rule of method overriding is that return type of overriding method must be same as overridden method but this restriction is relaxed little bit from Java 1.5 and now overridden method can return sub class of return type of original method.

This relaxation is known as co-variant method overriding and it allows you to remove casting at client end

<http://stackoverflow.com/questions/5875414/why-cant-overriding-methods-throw-exceptions-broader-than-the-overriden-method>

## 22) What is runtime polymorphism or dynamic method dispatch?

In Java, runtime polymorphism or dynamic method dispatch is a process in which a call to an overridden method is resolved at runtime rather than at compile-time. In this process, an overridden method is called through the reference variable of a superclass. Let's take a look at the example below to understand it better.

```
class Car {

    void run()

    {

        System.out.println("car is running");

    }

}

class Audi extends Car {

    void run()

    {

        System.out.println("Audi is running safely with 100km");

    }

    public static void main(String args[])

    {

        Car b= new Audi();    //upcasting

        b.run();

    }

}
```

## 23) Can we overload main method?

Yes, we can have multiple methods with name "main" in a single class. However if we run the class, java runtime environment will look for main method with syntax as public static void main(String args[]).

## 24) What is an interface?

In the Java programming language, an *interface* is a reference type, similar to a class, that can contain *only* constants, method signatures, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods. Interfaces cannot be instantiated—they can only be *implemented* by classes or *extended* by other interfaces.

Interface body –

Variable – public, static, final

Methods – abstract, default, static

Methods are always public.

We cannot initialize the instance variable in the interface.

Interfaces are good for starting point to define Type and create top level hierarchy in our code. Since a java class can implements multiple interfaces, it's better to use interfaces as super class in most of the cases. Read more at [java interface](#).

## 25) What is an abstract class?

Abstract classes are used in java to create a class with some default method implementation for subclasses. An abstract class can have abstract method without body and it can have methods with implementation also.

abstract keyword is used to create a abstract class. Abstract classes can't be instantiated and mostly used to provide base for sub-classes to extend and implement the abstract methods and override or use the implemented methods in abstract class. Read important points about abstract classes at [java abstract class](#).

- Abstract keyword is used to define abstract classes.
- Abstract method example – abstract void print();
- If any method is declared as abstract that class has to be declared as abstract where as vice versa is not true.

## 26) Can an interface implement or extend another interface?

Interfaces don't implement another interface, they extend it. Since interfaces can't have method implementations, there is no issue of diamond problem. That's why we have multiple inheritance in interfaces i.e an interface can extend multiple interfaces.

From Java 8 onwards, interfaces can have default method implementations. So to handle diamond problem when a common default method is present in multiple interfaces, it's mandatory to provide implementation of the method in the class implementing them. For more details with examples, read [Java 8 interface changes](#).

## 27) What is the benefit of Composition over Inheritance?

- One of the best practices of java programming is to "favor composition over inheritance". Some of the possible reasons are:
- Any change in the superclass might affect subclass even though we might not be using the superclass methods. For example, if we have a method test() in subclass and suddenly somebody introduces a method test() in superclass, we will get compilation errors in subclass. Composition will never face this issue because we are using only what methods we need.
- Inheritance exposes all the super class methods and variables to client and if we have no control in designing superclass, it can lead to security holes. Composition allows us to provide restricted access to the methods and hence more secure.
- We can get runtime binding in composition where inheritance binds the classes at compile time. So composition provides flexibility in invocation of methods.

You can read more about above benefits of composition over inheritance at [java composition vs inheritance](#).

## 28) What is the difference between abstract classes and interfaces?

Difference between interface and abstract class:

1. when we don't have any knowledge of implementation use interface  
when limited knowledge abstract
2. methods are public and abstract in interface and methods can't be final, private, protected , native , synchronized and strcitfp in interface.In java 8, we can have default and static methods.
3. variables are always public , static and final in Interface they cannot be variables can't be private, protected, volatile and transient.
4. initialization required for variables in Interface because -- The reason why you are getting the "not initialized" error is because the field on the interface is, as I said, automatically public static final regardless of what modifiers you place on it. Since you did not initialize the static final field, the compiler complains at you.
5. A class can extend only one abstract class but it can implement multiple interfaces.
- 6.abstract keyword is used to create abstract class whereas interface is the keyword for interfaces.

5. no instance or static block in Interface.

Workaround:

```
interface ITest {
    public static final String hello = Hello.hello();
}

// You can have non-public classes in the same file.
class Hello {
    static {
        System.out.println("Static Hello");
    }
    public static String hello() {
        System.out.println("Hello again");
        return "Hello";
    }
}

public class InterfaceTester implements ITest{

    public static void main(String[] args) {

        ITest obj = new InterfaceTester();
        System.out.println(obj.hello);

    }

}
```

9. constructor not allowed in Interface.

29) Why abstract methods have constructors ?

Imagine that your abstract class has fields x and y, and that you always want them to be initialized in a certain way, no matter what actual concrete subclass is eventually created. So you create a constructor and initialize these fields. Now, if you have two different subclasses of your abstract class, when you instantiate them their constructors will be called, and then the parent constructor will be called and the fields will be initialized. If you don't do anything, the default constructor of the parent will be called. However, you can use the super keyword to invoke specific constructor on the parent class.

### 30) Difference between abstract and interface?

Abstract Class	Interfaces
An abstract class can provide complete, default code and/or just the details that have to be overridden.	An interface cannot provide any code at all, just the signature.
In case of abstract class, a class may extend only one abstract class.	A Class may implement several interfaces.
An abstract class can have non-abstract methods.	All methods of an Interface are abstract.
An abstract class can have instance variables.	An Interface cannot have instance variables
An abstract class can have any visibility: public, private, protected.	An Interface visibility must be public (or) none.
If we add a new method to an abstract class then we have the option of providing default implementation and therefore all the existing code might work properly	If we add a new method to an Interface then we have to track down all the implementations of the interface and define implementation for the new method
An abstract class can contain constructors	An Interface cannot contain constructors
Abstract classes are fast	Interfaces are slow as it requires extra indirection to find corresponding method in the actual class



### 31) When should we use interface and abstract class?

Consider using abstract classes if any of these statements apply to your situation:

- You want to share code among several closely related classes. An example of an abstract class in the JDK is `AbstractMap`, which is part of the Collections Framework. Its subclasses (which include `HashMap`, `TreeMap`, and `ConcurrentHashMap`) share many methods (including `get`, `put`, `isEmpty`, `containsKey`, and `containsValue`) that `AbstractMap` defines.
- You expect that classes that extend your abstract class have many common methods or fields, or require access modifiers other than `public` (such as `protected` and `private`).
- You want to declare non-static or non-final fields. This enables you to define methods that can access and modify the state of the object to which they belong.

Abstract class is IS-A relationship.

Consider using interfaces if any of these statements apply to your situation:

- You expect that unrelated classes would implement your interface. For example, the interfaces `Comparable` and `Cloneable` are implemented by many unrelated classes.
- You want to specify the behavior of a particular data type, but not concerned about who implements its behavior.
- You want to take advantage of multiple inheritance of type.
- An example of a class in the JDK that implements several interfaces is `HashMap`, which implements the interfaces `Serializable`, `Cloneable`, and `Map<K, V>`. By reading this list of interfaces, you can infer that an instance of `HashMap` (regardless of the developer or company who implemented the class) can be cloned, is serializable (which means that it can be converted into a byte stream; see the section [Serializable Objects](#)), and has the functionality of a map. In addition, the `Map<K, V>` interface has been enhanced with many default methods such as `merge` and `forEach` that older classes that have implemented this interface do not have to define.
- Note that many software libraries use both abstract classes and interfaces; the `HashMap` class implements several interfaces and also extends the abstract class `AbstractMap`.

Interface is CAN-DO-THIS relationship.

### 32) Why java 8 introduced default method in interface.

If we add any new method in the interface, all subclasses need to implement that method, compiler will give error in all class. To prevent this, java 8 has concept of default method, so that we can modify the base class without changing the subclass. In java 8, lot of interface has been modified for adding new methods to collection api.

### 33) Why java 8 introduced static method in interface?

Need to search.

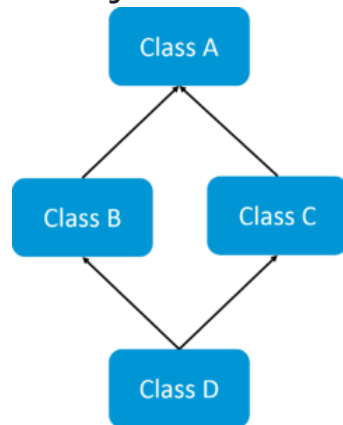
### 34) Can you override a private or static method in Java?

You cannot override a private or static method in Java. If you create a similar method with same return type and same method arguments in child class then it will hide the super class method; this is known as method hiding. Similarly, you cannot override a private method in sub class because it's not accessible there. What you can do is create another private method with the same name in the child class. Let's take a look at the example below to understand it better.

```
class Base {  
  
    private static void display() {  
        System.out.println("Static or class method from Base");  
    }  
  
    public void print() {  
        System.out.println("Non-static or instance method from Base");  
    }  
  
    class Derived extends Base {  
        private static void display() {  
            System.out.println("Static or class method from Derived");  
        }  
  
        public void print() {  
            System.out.println("Non-static or instance method from Derived");  
        }  
  
        public class test {  
            public static void main(String args[])  
            {
```

```
Base obj= new Derived();  
obj1.display();  
obj1.print();  
}  
}
```

**35)** What is multiple inheritance? Is it supported by Java?  
Prev to java 8.



If a child class inherits the property from multiple classes is known as multiple inheritance. Java does not allow to extend multiple classes.

The problem with multiple inheritance is that if multiple parent classes have a same method name, then at runtime it becomes difficult for the compiler to decide which method to execute from the child class.

Therefore, Java doesn't support multiple inheritance. The problem is commonly referred as Diamond Problem.

But with java8, we can have default and static methods in interface, so in the implementation class, it is must to override the default methods.

## PROGRAMMING FUNDAMENTALS

### 36) What are methods of Object class?

Five methods are not final:

Equals  
hashCode  
finalize --- protected.  
toString  
clone -- protected

As an additional check, clone checks that the class implements Cloneable, only to ensure you don't clone non-cloneables by accident.

generic abstract Object it is unclear what to do if user wants to clone it for example, or finalize. That's why we have a chance to override this methods and create our own implementation.

You can't call the clone method from the object reference of base class object.

### 37) What is the use of Java Package? Which java package is imported by default?

Java Package is useful for organizing projects containing multiple modules and protecting them from unauthorized access. Java.lang package is imported by default.

### 38) While working in the JVM, do we need to import java.lang package?

No, by default it is loaded in the JVM

### 39) Can a .java file support more than one java classes?

Yes, it can support more than one Java classes in a condition where one of them is a public class.

### 40) Does java Support default arguments?

NO

#### 41) What is the importance of main method in Java?

main() method is the entry point of any standalone java application. The syntax of main method is `public static void main(String args[]).main` method is public and static so that java can access it without initializing the class. The input parameter is an array of String through which we can pass runtime arguments to the java program. Check this post to learn [how to compile and run java program](#).

**public** : Public is an access modifier, which is used to specify who can access this method. Public means that this Method will be accessible by any Class.

**static** : It is a keyword in java which identifies it is class based i.e it can be accessed without creating the instance of a Class.

**void** : It is the return type of the method. Void defines the method which will not return any value.

**main**: It is the name of the method which is searched by JVM as a starting point for an application with a particular signature only. It is the method where the main execution occurs.

**String args[]** : It is the parameter passed to the main method.

#### 42) What are constructors in Java?

Answer: In Java, constructor refers to a block of code which is used to initialize an object. It must have the same name as that of the class. Also, it has no return type and it is automatically called when an object is created.

There are two types of constructors:

- Default constructor
- Parameterized constructor

#### 43) Difference between stack and heap?

What are the differences between Heap and Stack Memory in Java?

The major difference between Heap and Stack memory are:

Features	Stack	Heap
<b>Memory</b>	Stack memory is used only by one thread of execution.	Heap memory is used by all the parts of the application.
<b>Access</b>	Stack memory can't be accessed	Objects stored in the heap are globally accessible.

	by other threads.	
<b>Memory Management</b>	Follows LIFO manner to free memory.	Memory management is based on the generation associated with each object.
<b>Lifetime</b>	Exists until the end of execution of the thread.	Heap memory lives from the start till the end of application execution.
<b>Usage</b>	Stack memory only contains local primitive and reference variables to objects in heap space.	Whenever an object is created, it's always stored in the Heap space.

#### 44) Why pointers are not used in JAVA?

Java doesn't use pointers because they are unsafe and increases the complexity of the program. Since, Java is known for its simplicity of code, adding the concept of pointers will be contradicting. Moreover, since JVM is responsible for implicit memory allocation, thus in order to avoid direct access to memory by the user, pointers are discouraged in Java.

#### 45) What is bytecode in java?

**Bytecode** is the compiled format for **Java** programs. Once a **Java** program has been converted to **bytecode**, it can be transferred across a network and executed by **Java**Virtual Machine (JVM). **Bytecode files** generally have a .class extension.

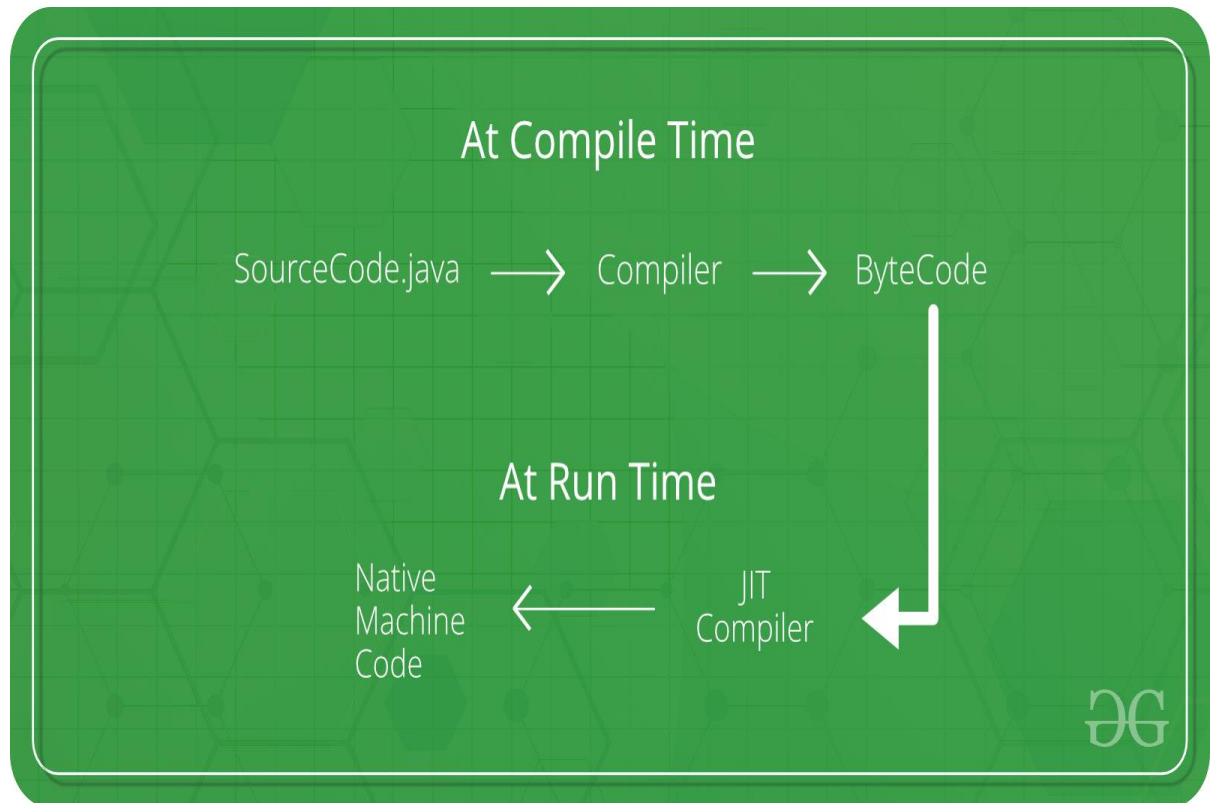
#### 46) What is JIT compiler?

The Just-In-Time (JIT) compiler is a an essential part of the JRE i.e. Java Runtime Environment, that is responsible for performance optimization of java based applications at run time. Compiler is one of the key aspects in deciding performance of an application for both parties i.e. the end user and the application developer.

##### **Java JIT Compiler : General Overview**

Bytecode is one of the most important features of java that aids in cross-platform execution. Way of converting bytecode to native machine language for execution has a huge impact on the speed of it. These Bytecode have to be interpreted or compiled to proper machine instructions depending on the instruction set architecture. Moreover these can be directly executed if the instruction architecture is bytecode based. Interpreting the bytecode affects the speed of execution.

In order to improve performance, JIT compilers interact with the Java Virtual Machine (JVM) at run time and compile suitable bytecode sequences into native machine code. While using a JIT compiler, the hardware is able to execute the native code, as compared to having the JVM interpret the same sequence of bytecode repeatedly and incurring an overhead for the translation process. This subsequently leads to performance gains in the execution speed, unless the compiled methods are executed less frequently. The JIT compiler is able to perform certain simple optimizations while compiling a series of bytecode to native machine language. Some of these optimizations performed by JIT compilers are data-analysis, reduction of memory accesses by register allocation, translation from stack operations to register operations, elimination of common sub-expressions etc. The greater is the degree of optimization done, the more time a JIT compiler spends in the execution stage. Therefore it cannot afford to do all the optimizations that a static compiler is capable of, because of the extra overhead added to the execution time and moreover it's view of the program is also restricted.



### Working of JIT Compiler

Java follows object oriented approach, as a result it consists of classes. These constitute of bytecode which are platform neutral and are executed by the JVM across diversified architectures.

At run time, the JVM loads the class files, the semantic of each is determined and appropriate computations are performed. The additional processor and memory usage during interpretation makes a Java application perform slowly as compared to a native application.

The JIT compiler aids in improving the performance of Java programs by compiling bytecode into native machine code at run time.

The JIT compiler is enabled throughout, while it gets activated, when a method is invoked. For a compiled method, the JVM directly calls the compiled code, instead of interpreting it. Theoretically speaking, If compiling did not require any processor time or memory usage, the speed of a native compiler and that of a Java compiler would have been same.

JIT compilation requires processor time and memory usage. When the java virtual machine first starts up, thousands of methods are invoked. Compiling all these methods can significantly affect startup time, even if the end result is a very good performance optimization.



#### 47) What is constructor chaining?

In Java, constructor chaining is the process of calling one constructor from another with respect to the current object. Constructor chaining is possible only through legacy where a subclass constructor is responsible for invoking the superclass' constructor first. There could be any number of classes in the constructor chain. Constructor chaining can be achieved in two ways:

- Within the same class using this()
- From base class using super()

#### 48) Use of super and this keyword?

In Java, super() and this(), both are special keywords that are used to call the constructor.

this()	super()
1. this() represents the current instance of a class	1. super() represents the current instance of a parent/base class
2. Used to call the default constructor of the same class	2. Used to call the default constructor of the parent/base class
3. Used to access methods of the current class	3. Used to access methods of the base class
4. Used for pointing the current class instance	4. Used for pointing the superclass instance
5. Must be the first line of a block	5. Must be the first line of a block

#### 49) Float f = 23.6, why it won't compile?

Reason is by default this is double in java, so when we write without using f, compiler warns for loosing precision and we need to write the f in the last 23.6f.

#### 50) Difference between b = b+5 and b+=5

That brings us to the compound assignment operators. The following will compile,

```
byte b = 3;
```

```
b += 7; // No problem - adds 7 to b (result is 10)
```

and is equivalent to

```
byte b = 3;
```

```
b = (byte) (b + 7); // Won't compile without the
```

```
// cast, since b + 7 results in an int
```

The compound assignment operator += lets you add to the value of b, without putting in an explicit cast. In fact, +=, -=, \*=, and /= will all put in an implicit cast

## 51) Order of static block, init block and constructor?

A static initialization block runs once, when the class is first loaded. An instance initialization block runs once every time a new instance is created. Remember when we talked about the order in which constructor code executed? Instance init block code runs rightafter the call to super() in a constructor, in other words, after all super-constructors have run. You can have many initialization blocks in a class. It is important to note that unlike methods or constructors, the order in which initialization blocks appear in a class matters. When it's time for initialization blocks to run, if a class has more than one, they will run in the order in which they appear in the class file...in other words, from the top down. Based on the rules we just discussed, can you determine the

output of the following program?

```
class Init {

    Init(int x) { System.out.println("1-arg const"); }

    Init() { System.out.println("no-arg const"); }

    static { System.out.println("1st static init"); }

    { System.out.println("1st instance init"); }

    { System.out.println("2nd instance init"); }

    static { System.out.println("2nd static init"); }

    public static void main(String [] args) {

        new Init();

        new Init(7);

    }

}
```

To figure this out, remember these rules:

- Init blocks execute in the order they appear.
- Static init blocks run once, when the class is first loaded.
- Instance init blocks run every time a class instance is created.
- Instance init blocks run after the constructor's call to super().

With those rules in mind, the following output should make sense:

```
1st static init

2nd static init
```

```
1st instance init
2nd instance init
no-arg const
1st instance init
2nd instance init
1-arg const
```

As you can see, the instance init blocks each ran twice. Instance init blocks are often used as a place to put code that all the constructors in a class should share. That way, the code doesn't have to be duplicated across constructors.

## 52) What happens if there is any error in the static block?

Finally, if you make a mistake in your static init block, the JVM can throw an `ExceptionInInitializationError`. Let's look at an example,

```
class InitError {
    static int [] x = new int[4];
    static { x[4] = 5; } // bad array index!
    public static void main(String [] args) { }
}
```

which produces something like:

```
Exception in thread "main" java.lang.ExceptionInInitializerError
Caused by: java.lang.ArrayIndexOutOfBoundsException: 4
at InitError.<clinit>(InitError.java:3)
```

## 53) What is tail recursive?

A recursive function is tail recursive when recursive call is the last thing executed by the function. For example the following C++ function `print()` is tail recursive.

```
// An example of tail recursive function
void print(int n)
{
    if (n < 0) return;
    cout << " " << n;
```

```

// The last executed statement is recursive call
print(n-1);
}

```

The tail recursive functions considered better than non tail recursive functions as tail-recursion can be optimized by compiler. The idea used by compilers to optimize tail-recursive functions is simple, since the recursive call is the last statement, there is nothing left to do in the current function, so saving the current function's stack frame is of no use

Modern compiler basically do **tail call elimination** to optimize the tail recursive code.

If we take a closer look at above function, we can remove the last call with goto. Below are examples of tail call elimination.

```

// Above code after tail call elimination
void print(int n)
{
start:
    if (n < 0)
        return;
    cout << " " << n;

    // Update parameters of recursive call
    // and replace recursive call with goto
    n = n-1;
    goto start;
}

```

## 54) Immutable objects?

Immutable objects greatly simplify your program, since they:

- | are simple to construct, test, and use
- | are automatically thread-safe and have no synchronization issues
- | don't need a copy constructor
- | don't need an implementation of clone
- | allow hashCode to use lazy initialization, and to cache its return value
- | don't need to be copied defensively when used as a field
- | make good Map keys and Set elements (these objects must not change state while in the collection)
- | have their class invariant established once upon construction, and it never needs to be checked again
- | always have "failure atomicity" (a term used by Joshua Bloch):  
if an immutable object throws an exception, it's never left in an undesirable or indeterminate state

Class can be made immutable by making it final

or

make fields private and final

Immutable class means that once an object is created, we cannot change its content. In Java, all the wrapper classes (like String, Boolean, Byte, Short) and String class is immutable. We can create our own immutable class as well. Immutable means that once the constructor for an object has completed execution that instance can't be altered.

Following are the requirements:

- Class must be declared as final (So that child classes can't be created)
- Data members in the class must be declared as final (So that we can't change the value of it after object creation)
- A parameterized constructor
- Getter method for all the variables in it
- No setters (To not have option to change the value of the instance variable)

- Reflection can break immutability. <https://avaldes.com/hacking-immutable-class-using-java-reflection/>
- If variables are not primitive, then extra are need to be taken.

<https://stackoverflow.com/questions/40845750/is-a-class-with-only-static-members-immutable>

// An immutable class

```
public final class Student
```

```
{
```

```
    final String name;
```

```
final int regNo;

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

public String getName()
{
    return name;
}

public int getRegNo()
{
    return regNo;
}
}

// Driver class
class Test
{
    public static void main(String args[])
    {
        Student s = new Student("ABC", 101);
        System.out.println(s.getName());
        System.out.println(s.getRegNo());

        // Uncommenting below line causes error
        // s.regNo = 102;
    }
}
```

## 55) How to prevent immutability from reflection API?

The JVM has security mechanisms built into it that allow you to define restrictions to code through a Java security policy file. The Java security manager uses the Java security policy file to enforce a set of permissions granted to classes. The permissions allow specified classes running in that instance of the JVM to permit or not permit certain runtime operations. If you enable the Java security manager but do not specify a security policy file, the Java security manager uses the default security policies defined in the java.security and java.policy files in the \$JAVA\_HOME/jre/lib/security directory. Defining your policy file can be found here <http://docs.oracle.com/javase/7/docs/technotes/guides/security/PolicyFiles.html>

OR

Extend the SecurityManager class and override this method to restrict reflection access

```
@Override
public void checkPackageAccess(String pkg){

    // don't allow the use of the reflection package
    if(pkg.equals("java.lang.reflect")){
        throw new SecurityException("Reflection is not allowed!");
    }
}
```

## 56) What is lazy initialization of class?

The Java runtime has built-in lazy instantiation for classes. Classes load into memory only when they're first referenced.

public final class MyFrame extends Frame

```
{

    private MessageBox mb_ ; //null, implicit

    //private helper used by this class

    private void showMessage(String message)

    {

        if(mb_==null)//first call to this method

            mb_=new MessageBox();

        //set the message text

        mb_.setMessage( message );

    }

}
```

```
mb_.pack();  
mb_.show();  
}  
}
```

## 57) Interface with java 8?

Now in Java 8 we can have default and static methods in an interface.

If a class implements two interfaces and both interfaces have default method with same signature then it has to override that method with public keyword.

Functional Interfaces - An interface with exactly one abstract method is known as Functional Interface.

A new annotation `@FunctionalInterface` has been introduced to mark an interface as Functional Interface.

`@FunctionalInterface` annotation is a facility to avoid accidental addition of abstract methods in the functional interfaces.

It's optional but good practice to use it.

Functional interfaces are long awaited and much sought out feature of Java 8 because it enables us to use lambda expressions to instantiate them.

A new package `java.util.function` with bunch of functional interfaces are added to provide target types for lambda expressions and method references.

Nested Interface in an interface cant be private.

Inner class can be present in an interface but it would automatically be static and public.

```
Runnable r1 = () -> System.out.println("My Runnable");
```

<http://www.journaldev.com/2763/java-8-lambda-expressions-and-functional-interfaces-example-tutorial>



## 58) Describe static keyword in java?

Static can be used in four ways: static variables, static methods, static classes and it can be used across a block of code in any class in order to indicate code that runs when a virtual machine starts and before the instances are created.

- Local variable- We cannot declare local variables as static it leads to compile time error "illegal start of expression". Because being static variable it must get memory at the time of class loading, which is not possible to provide memory to local variable at the time of class loading.
- Static methods can be overloaded. static methods cant be overridden.
- static members belong to the class instead of a specific instance. It means that only one instance of a static field exists[1] even if you create a million instances of the class or you don't create any. It will be shared by all instances.
- Since static methods also do not belong to a specific instance, they can't refer to instance members static members can only refer to static members. Instance members can, of course access static members.

Side note: Of course, static members can access instance members through an object reference.

Example:

```
public class Example {  
    private static boolean staticField;  
    private boolean instanceField;  
    public static void main(String[] args) {  
        // a static method can access static fields  
        staticField = true;  
  
        // a static method can access instance fields through an object reference  
        Example instance = new Example();
```

```
instance.instanceField = true;  
}
```

[1]: Depending on the runtime characteristics, it can be one per ClassLoader or AppDomain or thread, but that is beside the point.

Java has static nested classes but it sounds like you're looking for a top-level static class.

Java has no way of making a top-level class static but you can simulate a static class like this:

Declare your class final - Prevents extension of the class since extending a static class makes no sense

Make the constructor private - Prevents instantiation by client code as it makes no sense to instantiate a static class

Make all the members and functions of the class static - Since the class cannot be instantiated no instance methods can be called or instance fields accessed

Note that the compiler will not prevent you from declaring an instance (non-static) member. The issue will only show up if you attempt to call the instance member

## 59) In Java, what is the default value of Float and Double?

Answer: Default value of Float is 0.0f while 0.0d for Double.

## 60) Is it possible to import same package or class twice? Will the JVM load the package twice at runtime?

Answer: It is possible to import the same package or class more than one time. Also, it won't have any effect on compiler or JVM. JVM will load the class for one time only, irrespective of the number of times you import the same class.

## 61) Why Java doesn't support multiple inheritance? What is changed with java 8?

Answer: Java doesn't support multiple inheritance in classes because of "Diamond Problem". To know more about diamond problem with example, read [Multiple Inheritance in Java](#).

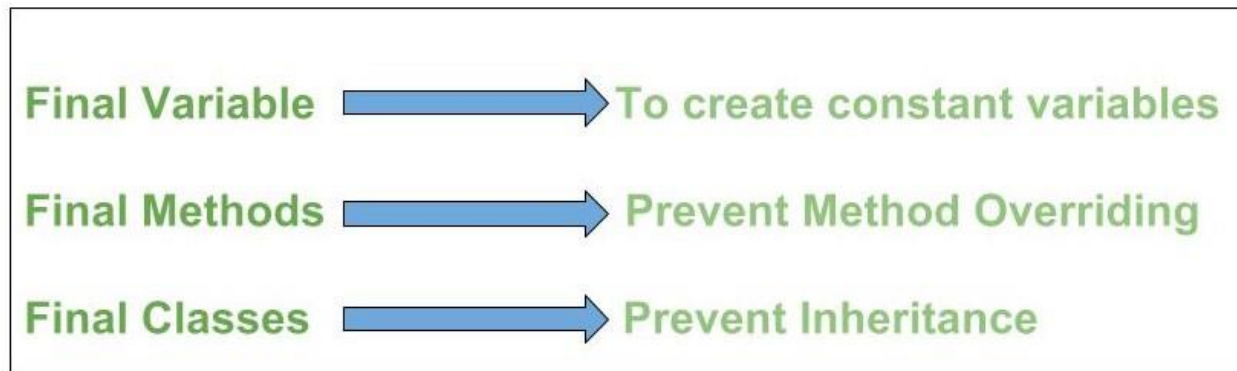
However multiple inheritance is supported in interfaces. An interface can extend multiple interfaces because they just declare the methods and implementation will be present in the implementing class. So there is no issue of diamond problem with interfaces.

## 62) Access Modifier

Java provides access control through public, private and protected access modifier keywords. When none of these are used, it's called default access modifier. A java class can only have public or default access modifier.

Read [Java Access Modifiers](#) to learn more about these in detail

## 63) Final Keyword



**Class:** final keyword is used with Class to make sure no other class can extend it, for example String class is final and we can't extend it. Wrapper classes are final. If we extend such class, we will get compiler error. Final classes are useful in immutable class and to prevent inheritance.

**Method:** We can use final keyword with methods to make sure child classes can't override it. Sometimes we don't need to prohibit a class extension entirely, but only prevent overriding of some methods. A good example of this is the Thread class. It's legal to extend it and thus create a custom thread class. But its `isAlive()` method is final. This method checks if a thread is alive. It's impossible to override the `isAlive()` method correctly for many reasons. One of them is that this method is native. Native code is implemented in another programming language and is often specific to the operating system and hardware it's running on. If we try to override final method, we will get compiler error.

**Variables:** final keyword can be used with variables to make sure that it can be assigned only once. Variables marked as final can't be reassigned. Once a final variable is initialized, it can't be altered.

When a variable is declared with *final* keyword, its value can't be modified, essentially, a constant. This also means that you must initialize a final variable. If the final variable is a reference, this means that the variable cannot be re-bound to reference another object, but internal state of the object pointed by that reference variable can be changed i.e. you can add or remove elements from [final array](#) or final collection. It is good practice to represent final variables in all uppercase, using underscore to separate words.

- Final Primitive Variables
- Final reference variable
- Static final variable

```
// a final variable
final int THRESHOLD = 5;
// a blank final variable
final int THRESHOLD;
// a final static variable PI
static final double PI = 3.141592653589793;
// a blank final static variable
static final double PI;
// final reference variable
final StringBuffer sb;
```

Initialization of final variables:

We must initialize a final variable, otherwise compiler will throw compile-time error. A final variable can only be initialized once, either via an [initializer](#) or an assignment statement

*static final* fields, this means that we can initialize them:

- upon declaration as shown in the above example
- in the static initializer block

For instance *final* fields, this means that we can initialize them:

- upon declaration
- in the instance initializer block
- in the constructor

Otherwise, the compiler will give us an error.

### Reference final variable :

When a final variable is a reference to an object, then this final variable is called reference final variable. For example, a final StringBuffer variable looks like

```
final StringBuffer sb;
```

As you know that a final variable cannot be re-assign. But in case of a reference final variable, internal state of the object pointed by that reference variable can be changed. Note that this is not re-assigning. This property of *final* is called *non-transitivity*. To understand what is mean by internal state of the object, see below example :

```
// Java program to demonstrate
```

```
// reference final variable
```

```
class Gfg
{
    public static void main(String[] args)
    {
        // a final reference variable sb
        final StringBuilder sb = new StringBuilder("Geeks");

        System.out.println(sb);

        // changing internal state of object
        // reference by final reference variable sb
        sb.append("ForGeeks");

        System.out.println(sb);
    }
}
```

```
}
```

Output: **GeeksForGeeks**

#### Final Arguments

The final keyword is also legal to put before method arguments. A final argument can't be changed inside a method:

```
public void methodWithFinalArguments(final int x) {  
    x=1;  
}
```

The above assignment causes the compiler error:

The final local variable x cannot be assigned. It must be blank and not using a  
1 compound assignment

<https://www.geeksforgeeks.org/final-keyword-java/>

### 64) Difference between finally and finalize

Answer: finally block is used with try-catch to put the code that you want to get executed always, even if any exception is thrown by the try-catch block. finally block is mostly used to release resources created in the try block. finalize() is a special method in Object class that we can override in our classes. This method get's called by garbage collector when the object is getting garbage collected. This method is usually overridden to release system resources when object is garbage collected.

### 65) Can we declare class as static?

Answer: We can't declare a top-level class as static however an inner class can be declared as static. If inner class is declared as static, it's called static nested class. Static nested class is same as any other top-level class and is nested for only packaging convenience.

### 66) What is static import?

Answer: If we have to use any static variable or method from other class, usually we import the class and then use the method/variable with class name.

```
import java.lang.Math;
```

```
//inside class
```

```
double test = Math.PI * 5;
```

We can do the same thing by importing the static method or variable only and then use it in the class as if it belongs to it.

```
import static java.lang.Math.PI;
```

```
//no need to refer class now
```

```
double test = PI * 5;
```

Use of static import can cause confusion, so it's better to avoid it. Overuse of static import can make your program unreadable and unmaintainable.

## 67) Static block usage?

Java static block is the group of statements that gets executed when the class is loaded into memory by Java ClassLoader. It is used to initialize static variables of the class. Mostly it's used to create static resources when class is loaded.

## 68) How to run a JAR file through command prompt?

We can run a jar file using java command but it requires Main-Class entry in jar manifest file. Main-Class is the entry point of the jar and used by java command to execute the class. Learn more at [java jar file](#).

## 69) Command prompt – how to check java is installed, compile a class, run a class?

## 70) What is the use of System class?

Java System Class is one of the core classes. One of the easiest way to log information for debugging is System.out.print() method.

System class is final so that we can't subclass and override it's behavior through inheritance. System class doesn't provide any public constructors, so we can't instantiate this class and that's why all of it's methods are static.

Some of the utility methods of System class are for array copy, get current time, reading environment variables. Read more at [Java System Class](#).

## 71) What is instanceof keyword?

We can use instanceof keyword to check if an object belongs to a class or not.

We should avoid it's usage as much as possible. Sample usage is:

```
public static void main(String args[]){  
    Object str = new String("abc");  
  
    if(str instanceof String){
```

```
        System.out.println("String value:"+str);  
    }  
  
    if(str instanceof Integer){  
        System.out.println("Integer value:"+str);  
    }  
}
```

Since str is of type String at runtime, first if statement evaluates to true and second one to false.

## 72) Can we use String with switch case?

One of the Java 7 feature was improvement of switch case of allow Strings. So if you are using Java 7 or higher version, you can use String in switch-case statements. Read more at [Java switch-case String example](#).

## 73) Java is Pass by Value or Pass by Reference?

Java is pass by value. This is a very confusing question, we know that object variables contain reference to the Objects in heap space. When we invoke any method, a copy of these variables is passed and gets stored in the stack memory of the method. We can test any language whether it's pass by reference or pass by value through a simple generic swap method, to learn more read [Java is Pass by Value and Not Pass by Reference](#).



The two most prevalent modes of passing arguments to methods are “passing-by-value” and “passing-by-reference”. Different programming languages use these concepts in different ways. **As far as Java is concerned, everything is strictly *Pass-by-Value*.**

In this tutorial, we’re going to illustrate how Java passes arguments for various types.

#### Pass-by-Value vs Pass-by-Reference

Let’s start with some of the different mechanisms for passing parameters to functions:

- value
- reference
- result
- value-result
- name

The two most common mechanisms in modern programming languages are “Pass-by-Value” and “Pass-by-Reference”. Before we proceed, let’s discuss these first:

#### Pass-by-Value

When a parameter is pass-by-value, the caller and the callee method operate on two different variables which are copies of each other. Any changes to one variable don’t modify the other.

It means that while calling a method, parameters passed to the callee method will be clones of original parameters. Any modification done in callee method will have no effect on the original parameters in caller method.

#### Pass-by-Reference

When a parameter is pass-by-reference, the caller and the callee operate on the same object.

It means that when a variable is pass-by-reference, the unique identifier of the object is sent to the method. Any changes to the parameter’s instance members will result in that change being made to the original value.

## Parameter Passing in Java

The fundamental concepts in any programming language are “values” and “references”. In Java, Primitive variables store the actual values, whereas Non-Primitives store the reference variables which point to the addresses of the objects they’re referring to. Both values and references are stored in the stack memory.

Arguments in Java are always passed-by-value. During method invocation, a copy of each argument, whether its a value or reference, is created in stack memory which is then passed to the method.

In case of primitives, the value is simply copied inside stack memory which is then passed to the callee method; in case of non-primitives, a reference in stack memory points to the actual data which resides in the heap. When we pass an object, the reference in stack memory is copied and the new reference is passed to the method.

Let’s now see this in action with the help of some code examples.

## Passing Primitive Types

The Java Programming Language features eight primitive data types. Primitive variables are directly stored in stack memory. Whenever any variable of primitive data type is passed as an argument, the actual parameters are copied to formal arguments and these formal arguments accumulate their own space in stack memory.

The lifespan of these formal parameters lasts only as long as that method is running, and upon returning, these formal arguments are cleared away from the stack and are discarded.

Let’s try to understand it with the help of a code example:

```
1          public class PrimitivesUnitTest {
2
3          @Test
4          public void whenModifyingPrimitives_thenOriginalValuesNotModi
5
6          int x = 1;
7          int y = 2;
8
9          // Before Modification
10         assertEquals(x, 1);
11         assertEquals(y, 2);
12
13         modify(x, y);
```

```

14
15          // After Modification
16          assertEquals(x, 1);
17          assertEquals(y, 2);
18      }
19
20      public static void modify(int x1, int y1) {
21          x1 = 5;
22          y1 = 10;
23      }
24  }

```

Let's try to understand the assertions in the above program by analyzing how these values are stored in memory:

The variables "x" and "y" in the main method are primitive types and their values are directly stored in the stack memory

When we call method *modify()*, an exact copy for each of these variables is created and stored at a different location in the stack memory

Any modification to these copies affects only them and leaves the original variables unaltered

Initial Stack space	Stack space when <i>modify()</i> method called	Stack space after <i>modify()</i> method call												
<table><tr><td>x = 1</td></tr><tr><td>y = 2</td></tr><tr><td></td></tr><tr><td></td></tr></table>	x = 1	y = 2			<table><tr><td>x = 1</td></tr><tr><td>y = 2</td></tr><tr><td>x1 = 1</td></tr><tr><td>y1 = 2</td></tr></table>	x = 1	y = 2	x1 = 1	y1 = 2	<table><tr><td>x = 1</td></tr><tr><td>y = 2</td></tr><tr><td>x1 = 5</td></tr><tr><td>y1 = 10</td></tr></table>	x = 1	y = 2	x1 = 5	y1 = 10
x = 1														
y = 2														
x = 1														
y = 2														
x1 = 1														
y1 = 2														
x = 1														
y = 2														
x1 = 5														
y1 = 10														

### Passing Object References

In Java, all objects are dynamically stored in Heap space under the hood. These objects are referred from references called reference variables.

A Java object, in contrast to Primitives, is stored in two stages. The reference variables are stored in stack memory and the object that they're referring to, are stored in a Heap memory.

Whenever an object is passed as an argument, an exact copy of the reference variable is created which points to the same location of the object in heap memory as the original reference variable.

As a result of this, whenever we make any change in the same object in the method, that change is reflected in the original object. However, if we allocate a new object to the passed reference variable, then it won't be reflected in the original object.

Let's try to comprehend this with the help of a code example:

```
1      public class NonPrimitivesUnitTest {
2
3          @Test
4          public void whenModifyingObjects_thenOriginalObjectChanged()
5              Foo a = new Foo(1);
6              Foo b = new Foo(1);
7
8              // Before Modification
9              assertEquals(a.num, 1);
10             assertEquals(b.num, 1);
11
12             modify(a, b);
13
14             // After Modification
15             assertEquals(a.num, 2);
16             assertEquals(b.num, 1);
17         }
18
19         public static void modify(Foo a1, Foo b1) {
20             a1.num++;
21         }
```

```

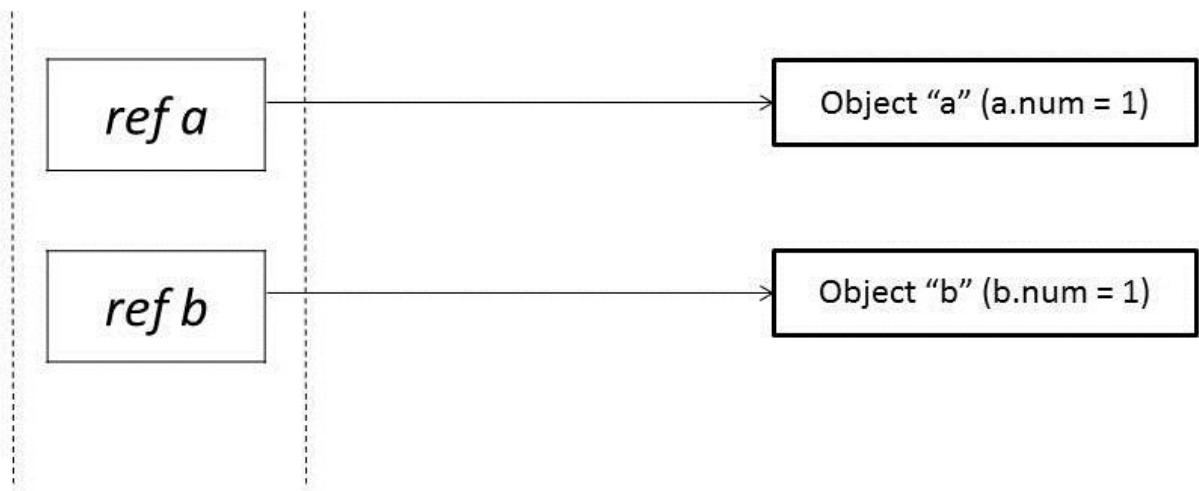
22         b1 = new Foo(1);
23         b1.num++;
24     }
25 }
26
27 class Foo {
28     public int num;
29
30     public Foo(int num) {
31         this.num = num;
32     }
33 }

```

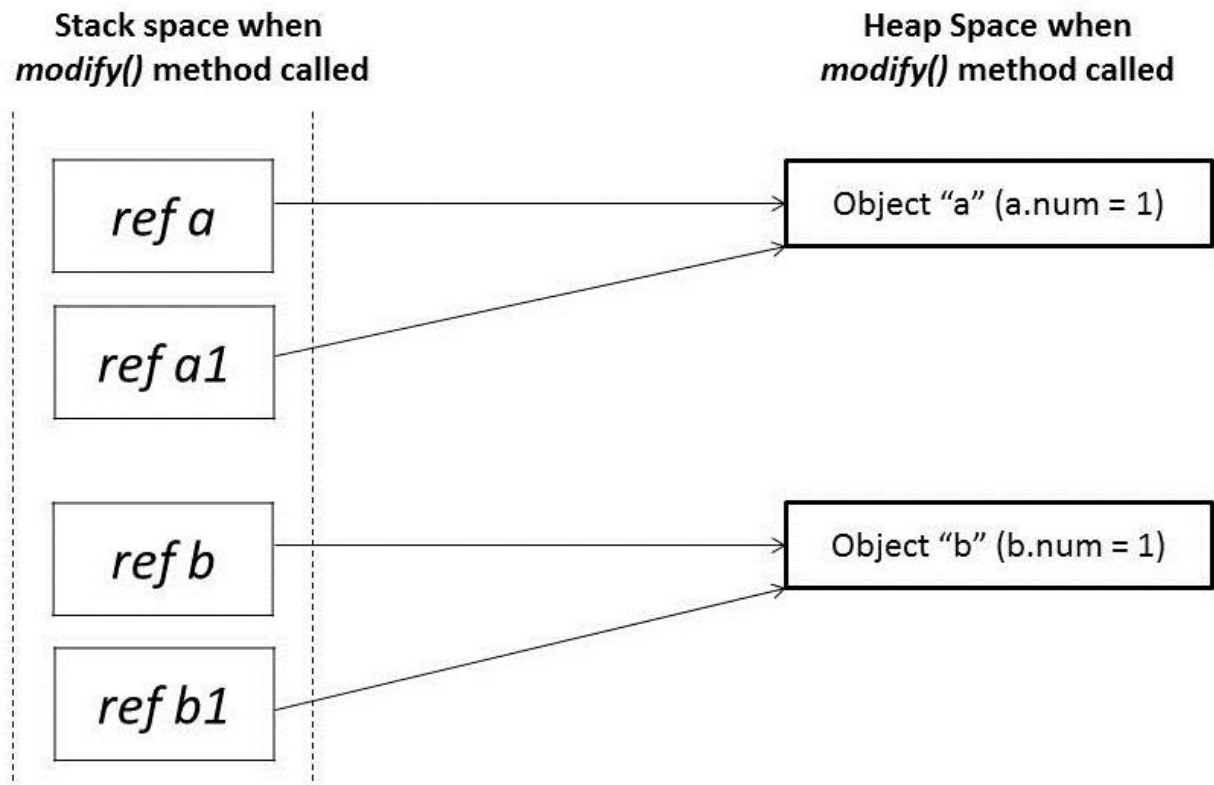
Let's analyze the assertions in the above program. We have passed objects *a* and *b* in *modify()* method that has the same value 1. Initially, these object references are pointing to two distinct object locations in a heap space:

**Initial Stack space**

**Initial Heap Space**

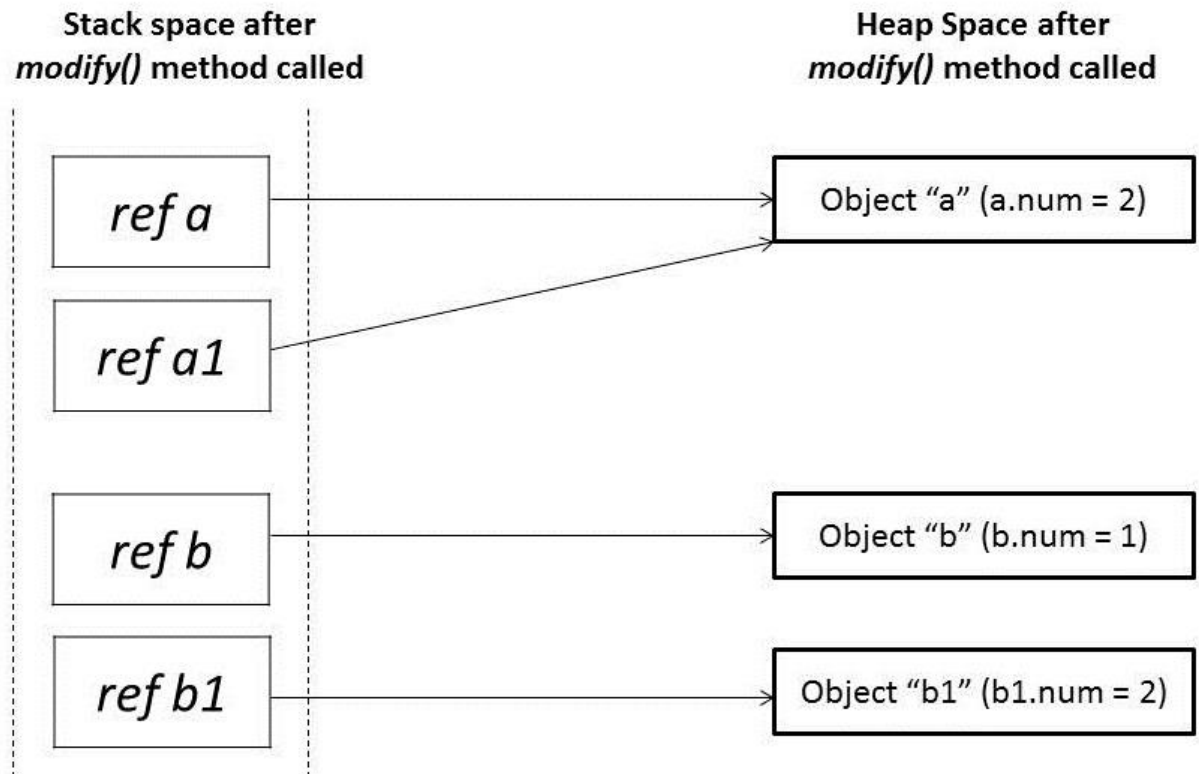


When these references *a* and *b* are passed in the *modify()* method, it creates mirror copies of those references *a1* and *b1* which point to the same old objects:



In the *modify()* method, when we modify reference *a1*, it changes the original object. However, for a reference *b1*, we have assigned a new object. So it's now pointing to a new object in heap memory.

Any change made to *b1* will not reflect anything in the original object:



We learned that parameter passing in Java is always Pass-by-Value. However, the context changes depending upon whether we're dealing with Primitives or Objects:

- For Primitive types, parameters are pass-by-value
- For Object types, the object reference is pass-by-value

<https://www.baeldung.com/java-pass-by-value-or-pass-by-reference>

## 74) Arrays class method

The **Arrays** class in **java.util package** is a part of the **Java Collection Framework**. This class provides static methods to dynamically create and access **Java arrays**. It consists of only static methods and the methods of Object class. The methods of this class can be used by the class name itself.

- Fill an array with a particular value.
- Sort an Arrays.
- Search in an Arrays.
- Compare array
- Copy array

## 75) Collections class method

Collections class in java is a useful utility class to work with collections in java. The java.util.Collections class directly extends the Object class and exclusively consists of the static methods that operate on Collections or return them.

The important points about Java Collections class are:

- Java Collection class supports the **polymorphic algorithms** that operate on collections.
- Java Collection class throws a **NullPointerException** if the collections or class objects provided to them are null.

SN	Modifier & Type	Methods	Descriptions
1)	static <T> boolean	<a href="#"><u>addAll()</u></a>	It is used to adds all of the specified elements to the specified collection.
2)	static <T> Queue<T>	<a href="#"><u>asLifoQueue()</u></a>	It returns a view of a Deque as a Last-in-first-out (LIFO) Queue.
3)	static <T> int	<a href="#"><u>binarySearch()</u></a>	It searches the list for the specified object and returns their position in a sorted list.
4)	static <E> Collection<E>	<a href="#"><u>checkedCollection()</u></a>	It is used to returns a dynamically typesafe view of the specified collection.
5)	static <E> List<E>	<a href="#"><u>checkedList()</u></a>	It is used to returns a dynamically typesafe view of the specified list.



JAVA INTERVIEW QUESTIONS GENERIC,  
INNERCLAS,ENUMSET,EMUMAP,QUEUE, DEQUEUE

6)	static <K,V> Map<K,V>	<a href="#"><u>checkedMap()</u></a>	It is used to returns a dynamically typesafe view of the specified map.
7)	static <K,V> NavigableMap<K,V>	<a href="#"><u>checkedNavigableMap()</u></a>	It is used to returns a dynamically typesafe view of the specified navigable map.
8)	static <E> NavigableSet<E>	<a href="#"><u>checkedNavigableSet()</u></a>	It is used to returns a dynamically typesafe view of the specified navigable set.
9)	static <E> Queue<E>	<a href="#"><u>checkedQueue()</u></a>	It is used to returns a dynamically typesafe view of the specified queue.
10)	static <E> Set<E>	<a href="#"><u>checkedSet()</u></a>	It is used to returns a dynamically typesafe view of the specified set.
11)	static <K,V> SortedMap<K,V>	<a href="#"><u>checkedSortedMap()</u></a>	It is used to returns a dynamically typesafe view of the specified sorted map.
12)	static <E> SortedSet<E>	<a href="#"><u>checkedSortedSet()</u></a>	It is used to returns a dynamically typesafe view of the specified sorted set.
13)	static <T> void	<a href="#"><u>copy()</u></a>	It is used to copy all the elements from one list into another list.
14)	static boolean	<a href="#"><u>disjoint()</u></a>	It returns true if the two specified collections have no elements in common.

JAVA INTERVIEW QUESTIONS GENERIC,  
INNERCLAS,ENUMSET,EMUMAP,QUEUE, DEQUEE

15)	static <T> Enumeration<T>	<a href="#"><u>emptyEnumeration()</u></a>	It is used to get an enumeration that has no elements.
16)	static <T> Iterator<T>	<a href="#"><u>emptyIterator()</u></a>	It is used to get an Iterator that has no elements.
17)	static <T> List<T>	<a href="#"><u>emptyList()</u></a>	It is used to get a List that has no elements.
18)	static <T> ListIterator<T>	<a href="#"><u>emptyListIterator()</u></a>	It is used to get a List Iterator that has no elements.
19)	static <K,V> Map<K,V>	<a href="#"><u>emptyMap()</u></a>	It returns an empty map which is immutable.
20)	static <K,V> NavigableMap<K,V>	<a href="#"><u>emptyNavigableMap()</u></a>	It returns an empty navigable map which is immutable.
21)	static <E> NavigableSet<E>	<a href="#"><u>emptyNavigableSet()</u></a>	It is used to get an empty navigable set which is immutable in nature.
22)	static <T> Set<T>	<a href="#"><u>emptySet()</u></a>	It is used to get the set that has no elements.
23)	static <K,V> SortedMap<K,V>	<a href="#"><u>emptySortedMap()</u></a>	It returns an empty sorted map which is immutable.
24)	static <E> SortedSet<E>	<a href="#"><u>emptySortedSet()</u></a>	It is used to get the sorted set that has no elements.
25)	static <T> Enumeration<T>	<a href="#"><u>enumeration()</u></a>	It is used to get the enumeration over the specified collection.

26)	static <T> void	<a href="#"><u>fill()</u></a>	It is used to replace all of the elements of the specified list with the specified elements.
27)	static int	<a href="#"><u>frequency()</u></a>	It is used to get the number of elements in the specified collection equal to the specified object.
28)	static int	<a href="#"><u>indexOfSubList()</u></a>	It is used to get the starting position of the first occurrence of the specified target list within the specified source list. It returns -1 if there is no such occurrence in the specified list.
29)	static int	<a href="#"><u>lastIndexOfSubList()</u></a>	It is used to get the starting position of the last occurrence of the specified target list within the specified source list. It returns -1 if there is no such occurrence in the specified list.
30)	static <T> ArrayList<T>	<a href="#"><u>list()</u></a>	It is used to get an array list containing the elements returned by the specified enumeration in the order in which they are returned by the enumeration.

JAVA INTERVIEW QUESTIONS GENERIC,  
INNERCLAS,ENUMSET,EMUMAP,QUEUE, DEQUEUE

31)	static <T extends Object & Comparable<? super T>> T	<a href="#"><u>max()</u></a>	It is used to get the maximum value of the given collection, according to the natural ordering of its elements.
32)	static <T extends Object & Comparable<? super T>> T	<a href="#"><u>min()</u></a>	It is used to get the minimum value of the given collection, according to the natural ordering of its elements.
33)	static <T> List<T>	<a href="#"><u>nCopies()</u></a>	It is used to get an immutable list consisting of <b>n</b> copies of the specified object.
34)	static <E> Set<E>	<a href="#"><u>newSetFromMap()</u></a>	It is used to return a set backed by the specified map.
35)	static <T> boolean	<a href="#"><u>replaceAll()</u></a>	It is used to replace all occurrences of one specified value in a list with the other specified value.
36)	static void	<a href="#"><u>reverse()</u></a>	It is used to reverse the order of the elements in the specified list.
37)	static <T> Comparator<T>	<a href="#"><u>reverseOrder()</u></a>	It is used to get the comparator that imposes the reverse of the natural ordering on a collection of objects which implement the Comparable interface.

JAVA INTERVIEW QUESTIONS GENERIC,  
INNERCLAS,ENUMSET,EMUMAP,QUEUE, DEQUEUE

38)	static void	<a href="#"><u>rotate()</u></a>	It is used to rotate the elements in the specified list by a given distance.
39)	static void	<a href="#"><u>shuffle()</u></a>	It is used to randomly reorders the specified list elements using a default randomness.
40)	static <T> Set<T>	<a href="#"><u>singleton()</u></a>	It is used to get an immutable set which contains only the specified object.
41)	static <T> List<T>	<a href="#"><u>singletonList()</u></a>	It is used to get an immutable list which contains only the specified object.
42)	static <K,V> Map<K,V>	<a href="#"><u>singletonMap()</u></a>	It is used to get an immutable map, mapping only the specified key to the specified value.
43)	static <T extends Comparable<? super T>>void	<a href="#"><u>sort()</u></a>	It is used to sort the elements presents in the specified list of collection in ascending order.
44)	static void	<a href="#"><u>swap()</u></a>	It is used to swap the elements at the specified positions in the specified list.
45)	static <T> Collection<T>	<a href="#"><u>synchronizedCollection()</u></a>	It is used to get a synchronized (thread-safe) collection backed by the specified collection.

46)	static <T> List<T>	<a href="#"><u>synchronizedList()</u></a>	It is used to get a synchronized (thread-safe) collection backed by the specified list.
47)	static <K,V> Map<K,V>	<a href="#"><u>synchronizedMap()</u></a>	It is used to get a synchronized (thread-safe) map backed by the specified map.
48)	static <K,V> NavigableMap<K,V>	<a href="#"><u>synchronizedNavigableMap()</u></a>	It is used to get a synchronized (thread-safe) navigable map backed by the specified navigable map.
49)	static <T> NavigableSet<T>	<a href="#"><u>synchronizedNavigableSet()</u></a>	It is used to get a synchronized (thread-safe) navigable set backed by the specified navigable set.
50)	static <T> Set<T>	<a href="#"><u>synchronizedSet()</u></a>	It is used to get a synchronized (thread-safe) set backed by the specified set.
51)	static <K,V> SortedMap<K,V>	<a href="#"><u>synchronizedSortedMap()</u></a>	It is used to get a synchronized (thread-safe) sorted map backed by the specified sorted map.
52)	static <T> SortedSet<T>	<a href="#"><u>synchronizedSortedSet()</u></a>	It is used to get a synchronized (thread-safe) sorted set backed by the specified sorted set.

53)	static <T> Collection<T>	<a href="#"><u>unmodifiableCollection()</u></a>	It is used to get an unmodifiable view of the specified collection.
54)	static <T> List<T>	<a href="#"><u>unmodifiableList()</u></a>	It is used to get an unmodifiable view of the specified list.
55)	static <K,V> Map<K,V>	<a href="#"><u>unmodifiableMap()</u></a>	It is used to get an unmodifiable view of the specified map.
56)	static <K,V> NavigableMap<K,V>	<a href="#"><u>unmodifiableNavigableMap()</u></a>	It is used to get an unmodifiable view of the specified navigable map.
57)	static <T> NavigableSet<T>	<a href="#"><u>unmodifiableNavigableSet()</u></a>	It is used to get an unmodifiable view of the specified navigable set.
58)	static <T> Set<T>	<a href="#"><u>unmodifiableSet()</u></a>	It is used to get an unmodifiable view of the specified set.
59)	static <K,V> SortedMap<K,V>	<a href="#"><u>unmodifiableSortedMap()</u></a>	It is used to get an unmodifiable view of the specified sorted map.
60)	static <T> SortedSet<T>	<a href="#"><u>unmodifiableSortedSet()</u></a>	It is used to get an unmodifiable view of the specified sorted set.

## 76) What is ternary operator in java?

Java ternary operator is the only conditional operator that takes three operands. It's a one liner replacement for if-then-else statement and used a lot in java programming. We can use ternary operator if-else conditions or even switch conditions using nested ternary operators. An example can be found at [java ternary operator](#).

## 77) What does super keyword do?

super keyword can be used to access super class method when you have overridden the method in the child class.

We can use super keyword to invoke super class constructor in child class constructor but in this case it should be the first statement in the constructor method.

```
package com.journaldev.access;

public class SuperClass {

    public SuperClass(){

    }

    public SuperClass(int i){}

    public void test(){
        System.out.println("super class test method");
    }

}
```

Use of super keyword can be seen in below child class implementation.

```
package com.journaldev.access;

public class ChildClass extends SuperClass {

    public ChildClass(String str){
        //access super class constructor with super keyword
        super();

        //access child class method
        test();

        //use super to access super class method
        super.test();
    }

    @Override
    public void test(){
        System.out.println("child class test method");
    }

}
```



## 78) What is break and continue statement?

We can use break statement to terminate for, while, or do-while loop. We can use break statement in switch statement to exit the switch case. You can see the example of break statement at [java break](#). We can use break with label to terminate the nested loops.

The continue statement skips the current iteration of a for, while or do-while loop. We can use continue statement with label to skip the current iteration of outermost loop.

## 79) What is this keyword?

this keyword provides reference to the current object and it's mostly used to make sure that object variables are used, not the local variables having same name.

```
//constructor
public Point(int x, int y) {
    this.x = x;
    this.y = y;
}
```

We can also use this keyword to invoke other constructors from a constructor.

```
public Rectangle() {
    this(0, 0, 0, 0);
}
public Rectangle(int width, int height) {
    this(0, 0, width, height);
}
public Rectangle(int x, int y, int width, int height) {
    this.x = x;
    this.y = y;
    this.width = width;
    this.height = height;
}
```

## 80) What is default constructor?

No argument constructor of a class is known as default constructor. When we don't define any constructor for the class, java compiler automatically creates the default no-args constructor for the class. If there are other constructors defined, then compiler won't create default constructor for us.

# EXCEPTION HANDLING

## 81) Exception handling and error?

<http://www.codeproject.com/Articles/175482/Compiler-Internals-How-Try-Catch-Throw-are-Interpr>

All the exceptions are subclasses of `java.lang.Exception`. When you throw an exception, several things happen.

First, the exception object is created in the same way that any Java object is created: on the heap, with `new`.

Then the current path of execution is stopped and the reference for the exception object is ejected from the current context.

At this point the exception-handling mechanism takes over and begins to look for an appropriate place to continue executing the program.

This appropriate place is the exception handler, whose job is to recover from the problem so the program can either try another tack or just continue.

The object is, in effect, “returned” from the method, even though that object type isn’t normally what the method is designed to return.

A simplistic way to think about exception handling is as an alternate return mechanism, although you get into trouble if you take that analogy too far.

You can also exit from ordinary scopes by throwing an exception. But a value is returned, and the method or scope exits.

There are two constructors in all standard exceptions:

The first is the default constructor, and the second takes a string argument so that you can place pertinent information in the exception.

There are two basic models in exception-handling theory.

Java supports termination, in which you assume that the error is so critical that there’s no way to get back to where the exception occurred.

Whoever threw the exception decided that there was no way to salvage the situation, and they don’t want to come back.

The alternative is called resumption.

It means that the exception handler is expected to do something to rectify the situation, and then the faulting method is retried, presuming success the second time.

If you want resumption, it means you still hope to continue execution after the exception is handled.

If you want resumption-like behavior in Java, don't throw an exception when you encounter an error.

Instead, call a method that fixes the problem.

Alternatively, place your try block inside a while loop that keeps reentering the try block until the result is satisfactory.

If a RuntimeException gets all the way out to main( ) without being caught, printStackTrace( ) is called for that exception as the program exits

The catch block is only executed if an exception is thrown in the try block.

The finally block is executed always after the try(-catch) block, if an exception is thrown or not.

We can loose an exception thrown by a try block if we throw any other exception from finally block

An even simpler way to lose an exception is just to return from inside a finally clause

When you override a method, you can throw only the exceptions that have been specified in the base-class version of the method.

This is a useful restriction, since it means that code that works with the base class will automatically work with any object derived from the base class

(a fundamental OOP concept, of course), including exceptions.

The overridden event( ) method shows that a derived-class version of a method may choose not to throw any exceptions, even if the base-class version does

The restriction on exceptions does not apply to constructors. A constructor can throw anything it wants, regardless of what the base-class constructor throws.

However, since a base-class constructor must always be called one way or another (here, the default constructor is called automatically),

the derived-class constructor must declare any base-class constructor exceptions in its exception specification

catch(Annoyance a) will catch an Annoyance or any class derived from it.

This is useful because if you decide to add more derived exceptions to a method,

then the client programmer's code will not need changing as long as the client catches the base-class exceptions

A ClassNotFoundException is thrown when the reported class is not found by the ClassLoader in the CLASSPATH.

It could also mean that the class in question is trying to be loaded from another class which was loaded in a parent classloader

and hence the class from the child classloader is not visible.

Consider if NoClassDefFoundError occurs which is something like

```
java.lang.NoClassDefFoundError
```

```
src/com/TestClass
```

does not mean that the TestClass class is not in the CLASSPATH.

It means that the class TestClass was found by the ClassLoader however when trying to load the class, it ran into an error reading the class definition.

This typically happens when the class in question has static blocks or members which use a Class that's not found by the ClassLoader.

So to find the culprit, view the source of the class in question (TestClass in this case) and look for code using static blocks or static members.

The StackOverflowError is an Error Object thrown by the Runtime System when it Encounters that your application/code has ran out of the memory.

It may occur in case of recursive methods or a large amount of data is fetched from the server and stored in some object. This error is generated by JVM.

tips for avoiding null pointer exception:

1. use primitives whenever possible
2. equals method
3. valueOf instead of toString
4. avoid method chaining

JDK7 has introduced two major feature which is related to Error and Exception handling,

one is ability to handle multiple exception in one catch block, popularly known as multi catch block

and other is ARM blocks in Java 7 for automatic resource management, also known as try with resource.

ARM:

Whatever resource we are using should be subtypes of AutoCloseable otherwise will get compile time error.

The resources which we are using are closed in reverse order

Read more: <http://javarevisited.blogspot.com/2011/09/arm-automatic-resource-management-in.html#ixzz44yUny64S>

<http://tutorials.jenkov.com/java-exception-handling/try-with-resources.html>

\*\*\*\*\*

<http://javarevisited.blogspot.in/2011/09/javalangoutofmemoryerror-permgen-space.html>

<http://javarevisited.blogspot.in/2011/07/jdk7-multi-cache-block-example-tutorial.html>

<http://javarevisited.blogspot.in/2011/09/arm-automatic-resource-management-in.html>

Interview Questions:

<http://javarevisited.blogspot.in/2013/06/10-java-exception-and-error-interview-questions-answers-programming.html>

<http://www.journaldev.com/2167/java-exception-interview-questions-and-answers>

## 82) What is difference between Throw and Throws?

Answer: While Throw is used to trigger an exception, Throws is used in the declaration of exception. It is not possible to handle checked exception without Throws.

### 83) What is the significance of the order in which catch statements for FileNotFoundException and IOException are written?

Answer: It is crucial to consider the order as the FileNotFoundException is inherited from the IOException. Therefore, it is important that exception's subclasses caught first.

### 84) What is multi catch?

Java 7 one of the improvement was multi-catch block where we can catch multiple exceptions in a single catch block. This makes are code shorter and cleaner when every catch block has similar code.

If a catch block handles multiple exception, you can separate them using a pipe (|) and in this case exception parameter (ex) is final, so you can't change it.

### 85) Difference between ClassNotFoundException and classNotFound

**ClassNotFoundException** is an Exception and will be thrown when Java program dynamically tries to load a Java class at Runtime and don't find the corresponding class file on the classpath. Two keywords here "dynamically" and "runtime".

A classic example of these errors is whey you try to load JDBC driver by using Class.forName("driver name") and greeted with java.lang.ClassNotFoundException: com.mysql.jdbc.Driver.

So this error essentially comes when Java try to load a class using forName() or by loadClass() method of ClassLoader.

The key thing to note is that presence of that class on Java classpath is not checked on compile time. So even if those classes are not present on Java classpath your program will compile successfully and only fail when you try to run.

On the other hand, NoClassDefFoundError is an Error and more critical than ClassNotFoundException which is an exception and recoverable.

NoClassDefFoundError comes when a particular class was present in Java Classpath during compile time but not available during run-time.

A classic example of this error is using log4j.jar for logging purpose and forgot to include log4j.jar on the classpath in java during run-time.

The keyword here is, the class was present at compile time but not available at run-time.

This is normally occurring due to any method invocation on a particular class which is part of the library and not available on the classpath in Java.

By the way, NoClassDefFoundError can also come due to various other reason like static initializer failure or class not visible to Classloaders in the J2EE environment.

\*If two classes with the same name exist in Java Classpath then the class which comes earlier in Classpath will be picked by Java Virtual Machine

## 86) Can we have try without catch block?

Yes, we can have try-finally statement and hence avoiding catch block.

## 87) Do we need finally or catch with try with resource as well?

No, this will work fine.

```
try (PrintWriter writer = new PrintWriter(new File("test.txt"))) {  
    writer.println("Hello World");  
}
```



### 88) What is a finally block? Is there a case when finally will not execute?

Finally block is a block which always execute a set of statements. It is always associated with a try block regardless of any exception that occurs or not. Yes, finally will not be executed if the program exits either by calling `System.exit()` or by causing a fatal error that causes the process to abort.

### 89) What is difference between Error and Exception?

An error is an irrecoverable condition occurring at runtime. Such as `OutOfMemory` error. These JVM errors you can not repair them at runtime. Though error can be caught in catch block but the execution of application will come to a halt and is not recoverable.

While exceptions are conditions that occur because of bad input or human error etc. e.g. `FileNotFoundException` will be thrown if the specified file does not exist. Or a `NullPointerException` will take place if you try using a null reference. In most of the cases it is possible to recover from an exception (probably by giving user a feedback for entering proper values etc.

### 90) How can you handle Java exceptions?

There are five keywords used to handle exceptions in java:

- try
- catch
- finally
- throw
- throws

### 91) What are the differences between Checked Exception and Unchecked Exception?

Checked Exception

- The classes that extend `Throwable` class except `RuntimeException` and `Error` are known as checked exceptions.
- Checked exceptions are checked at compile-time.
- Example: `IOException`, `SQLException` etc.

Unchecked Exception

- The classes that extend `RuntimeException` are known as unchecked exceptions.
- Unchecked exceptions are not checked at compile-time.
- Example: `ArithmeticException`, `NullPointerException` etc.

## 92) What purpose does the keywords final, finally, and finalize fulfill?

final - modifier applicable for classes(can't be inherited), methods(can't be overridden) and variables(constant)

finally - block associated with try/catch - cleanup activities like closing db connection

finalize - IS a method present in Object Class. Is called before GC claim the object to perform last-minute clean-up activities.

Protected access specifier

can be overridden but its our responsibility to call It

finalize gets called only once by GC thread if object revives itself from finalize method than finalize will not be called again

Any Exception is thrown by finalize method is ignored by GC thread and it will not be propagated further, in fact, I doubt if you find any trace of it.

There is one way to increase the probability of running of finalize method by calling System.runFinalization() and

Runtime.getRuntime().runFinalization().

These methods put more effort that JVM call finalize() method of all object which are eligible for garbage collection and whose finalize has not yet called. It's not guaranteed, but JVM tries its best.

final program concurrency

### Final:

Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed. Let's take a look at the example below to understand it better.

```
class FinalVarExample {
    public static void main( String args[])
    {
        final int a=10;    // Final variable
        a=50;              //Error as value can't be changed
    }
}
```

### Finally

Finally is used to place important code, it will be executed whether exception is handled or not. Let's take a look at the example below to understand it better.

```
class FinallyExample {
    public static void main(String args[]){
        try {
            int x=100;
        }
        catch(Exception e) {
            System.out.println(e);
        }
        finally {
            System.out.println("finally block is executing");}
    }
}
```

### Finalize

Finalize is used to perform clean up processing just before object is garbage collected. Let's take a look at the example below to understand it better.

```
class FinalizeExample {
    public void finalize() {
        System.out.println("Finalize is called");
    }
    public static void main(String args[])
    {
        FinalizeExample f1=new FinalizeExample();
        FinalizeExample f2=new FinalizeExample();
        f1= NULL;
        f2=NULL;
        System.gc();
    }
}
```

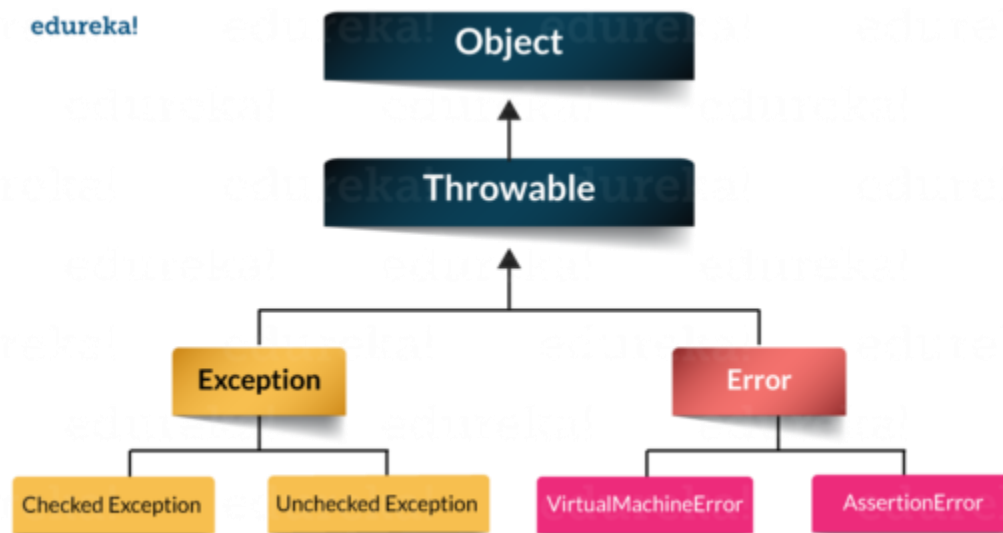
### 93) What are the differences between throw and throws?

throw keyword	throws keyword
Throw is used to explicitly throw an exception.	Throws is used to declare an exception.
Checked exceptions can not be propagated with throw only.	Checked exception can be propagated with throws.
Throw is followed by an instance.	Throws is followed by class.
Throw is used within the method.	Throws is used with the method signature.
You cannot throw multiple exception	You can declare multiple exception e.g. public void method()throws IOException,SQLException.

### 94) What is exception hierarchy in java?

The hierarchy is as follows:

Throwable is a parent class of all Exception classes. There are two types of Exceptions: Checked exceptions and UncheckedExceptions or RunTimeExceptions. Both type of exceptions extends Exception class whereas errors are further classified into Virtual Machine error and Assertion error.



### 95) Different type of error

[https://www.tutorialspoint.com/java/lang/java\\_lang\\_errors](https://www.tutorialspoint.com/java/lang/java_lang_errors)

<https://airbrake.io/blog/java-exception-handling/the-java-exception-class-hierarchy>

## 96) How to create a custom Exception?

To create you own exception extend the Exception class or any of its subclasses.

- class New1Exception extends Exception { } // this will create Checked Exception
- class NewException extends IOExcpetion { } // this will create Checked exception
- class NewException extends NullPonterExcpetion { } // this will create UnChecked exception

## 97) What are the important methods of Java Exception Class?

Exception and all of it's subclasses doesn't provide any specific methods and all of the methods are defined in the base class Throwable.

**String getMessage()** – This method returns the message String of Throwable and the message can be provided while creating the exception through it's constructor.

**String getLocalizedMessage()** – This method is provided so that subclasses can override it to provide locale specific message to the calling program. Throwable class implementation of this method simply use getMessage() method to return the exception message.

**Synchronized Throwable getCause()** – This method returns the cause of the exception or null id the cause is unknown.

**String toString()** – This method returns the information about Throwable in String format, the returned String contains the name of Throwable class and localized message.

**void printStackTrace()** – This method prints the stack trace information to the standard error stream, this method is overloaded and we can pass PrintStream or PrintWriter as argument to write the stack trace information to the file or stream.

## GARBAGE COLLECTION

## 98) What is garbage collection?

Garbage Collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects. In Java, process of deallocating memory is handled automatically by the garbage collector.

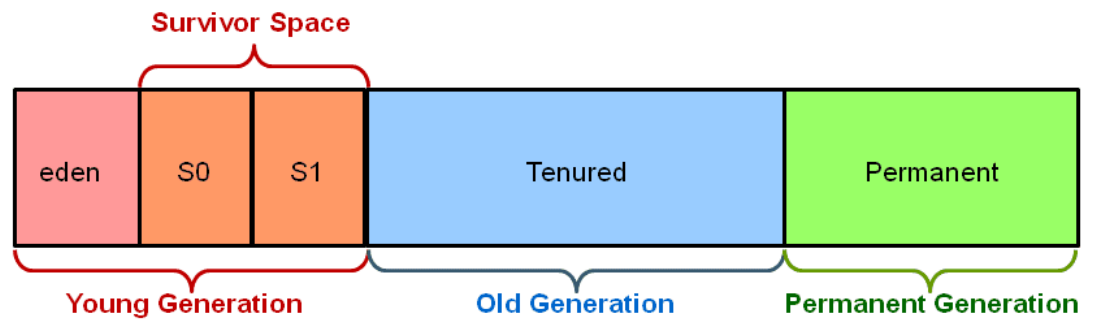
We can run the garbage collector with code `Runtime.getRuntime().gc()` or use utility method `System.gc()`. For a detailed analysis of Heap Memory and Garbage Collection, please read [Java Garbage Collection](#).

- When Garbage Collector calls `finalize()` method on an object, it **ignores** all the exceptions raised in the method and program will terminate normally.
- Garbage Collector i.e. it will call `finalize()` method on a particular object exactly **one** time.
- Objects created inside method is eligible for gc after method completion until they are not returned/referenced to some other object.

## 99) What is heap ?

When a Java program started Java Virtual Machine gets some memory from Operating System. Java Virtual Machine or JVM uses this memory for all its need and part of this memory is call java heap memory. Heap in Java generally located at bottom of address space and move upwards. 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 in Java

## Hotspot Heap Structure



### 100) How do you identify minor and major garbage collection in Java?

Minor collection prints "GC" if garbage collection [logging](#) is enable using `-verbose:gc` or `-XX:PrintGCDetails`, while Major collection prints "Full GC".

### 101) What is the meaning of the term "stop-the-world"?

When the garbage collector thread is running, other threads are stopped, meaning the application is stopped momentarily. This is analogous to house cleaning or fumigation where occupants are denied access until the process is complete. Depending on the needs of an application, "stop the world" garbage collection can cause an unacceptable freeze. This is why it is important to do garbage collector tuning and JVM optimization so that the freeze encountered is at least acceptable.

### 102) What are stack and heap? What is stored in each of these memory structures, and how are they interrelated?

The stack(LIFO) is a part of memory that contains information about nested method calls down to the current position in the program. It also contains all local variables and references to objects on the heap defined in currently executing methods. This structure allows the runtime to return from the method knowing the address whence it was called, and also clear all local variables after exiting the method. Every thread has its own stack. The heap is a large bulk of memory intended for allocation of objects. When you create an object with the `new` keyword, it gets allocated on the heap. However, the reference to this object lives on the stack.

### 94) What is generational garbage collection and what makes it a popular garbage collection approach?



Generational garbage collection can be loosely defined as the strategy used by the garbage collector where the heap is divided into a number of sections called generations, each of which will hold objects according to their "age" on the heap. Whenever the garbage collector is running, the first step in the process is called marking. This is where the garbage collector identifies which pieces of memory are in use and which are not. This can be a very time-consuming process if all objects in a system must be scanned. As more and more objects are allocated, the list of objects grows and grows leading to longer and longer garbage collection time. However, empirical analysis of applications has shown that most objects are short-lived. With generational garbage collection, objects are grouped according to their "age" in terms of how many garbage collection cycles they have survived. This way, the bulk of the work spread across various minor and major collection cycles. Today, almost all garbage collectors are generational. This strategy is so popular because, over time, it has proven to be the optimal solution.

### 103) Describe in detail how generational garbage collection works?

To properly understand how generational garbage collection works, it is important to first remember how Java heap is structured to facilitate generational garbage collection. The heap is divided up into smaller spaces or generations. These spaces are Young Generation, Old or Tenured Generation, and Permanent Generation.

The young generation hosts most of the newly created objects. An empirical study of most applications shows that majority of objects are quickly short lived and therefore, soon become eligible for collection. Therefore, new objects start their journey here and are only "promoted" to the old generation space after they have attained a certain "age".

The term "age" in generational garbage collection refers to the number of collection cycles the object has survived.

The young generation space is further divided into three spaces: an Eden space and two survivor spaces such as Survivor 1 (s1) and Survivor 2 (s2).

The old generation hosts objects that have lived in memory longer than a certain "age". The objects that survived garbage collection from the young generation are promoted to this space. It is generally larger than the young generation. As it is bigger in size, the garbage collection is more expensive and occurs less frequently than in the young generation.

The permanent generation or more commonly called, PermGen, contains metadata required by the JVM to describe the classes and methods used in the application. It also contains the string pool for storing interned strings. It is populated by the JVM at runtime based on classes in use by the application. In addition, platform library classes and methods may be stored here.

First, any new objects are allocated to the Eden space. Both survivor spaces start out empty. When the Eden space fills up, a minor garbage collection is triggered. Referenced objects are moved to the first survivor space. Unreferenced objects are deleted.

During the next minor GC, the same thing happens to the Eden space. Unreferenced objects are deleted and referenced objects are moved to a survivor space. However, in this case, they are moved to the second survivor space (S2).

In addition, objects from the last minor GC in the first survivor space (S1) have their age incremented and are moved to S2. Once all surviving objects have been moved to S2, both S1 and Eden space are cleared. At this point, S2 contains objects with different ages.

At the next minor GC, the same process is repeated. However this time the survivor spaces switch. Referenced objects are moved to S1 from both Eden and S2. Surviving objects are aged. Eden and S2 are cleared.

After every minor garbage collection cycle, the age of each object is checked. Those that have reached a certain arbitrary age, for example, 8, are promoted from the young generation to the old or tenured generation. For all subsequent minor GC cycles, objects will continue to be promoted to the old generation space.

This pretty much exhausts the process of garbage collection in the young generation. Eventually, a major garbage collection will be performed on the old generation which cleans up and compacts that space. For each major GC, there are several minor GCs.

#### 104) When does an object become eligible for garbage collection? Describe how the GC collects an eligible object?

An object becomes eligible for Garbage collection or GC if it is not reachable from any live threads or by any static references.

The most straightforward case of an object becoming eligible for garbage collection is if all its references are null. Cyclic dependencies without any live external reference are also eligible for GC. So if object A references object B and object B references Object A and they don't have any other live reference then both Objects A and B will be eligible for Garbage collection.

Another obvious case is when a parent object is set to null. When a kitchen object internally references a fridge object and a sink object, and the kitchen object is set to null, both fridge and sink will become eligible for garbage collection alongside their parent, kitchen.

#### 105) How do you trigger garbage collection from Java code?

You, as Java programmer, can not force garbage collection in Java; it will only trigger if JVM thinks it needs a garbage collection based on Java heap size. Before removing an object from memory garbage collection thread invokes `finalize()` method of that object and gives an opportunity to perform any sort of cleanup required. You can also invoke this method of an object code, however, there is no guarantee that garbage collection will occur when you call this method.

Additionally, there are methods like `System.gc()` and `Runtime.gc()` which is used to send request of Garbage collection to JVM but it's not guaranteed that garbage collection will happen.

#### 106) What happens when there is not enough heap space to accommodate storage of new objects?

If there is no memory space for creating a new object in Heap, Java Virtual Machine throws `OutOfMemoryError` or more specifically `java.lang.OutOfMemoryError` heap space.

### 107) Is it possible to «resurrect» an object that became eligible for garbage collection?

When an object becomes eligible for garbage collection, the GC has to run the finalize method on it. The finalize method is guaranteed to run only once, thus the GC flags the object as finalized and gives it a rest until the next cycle.

In the finalize method you can technically “resurrect” an object, for example, by assigning it to a static field. The object would become alive again and non-eligible for garbage collection, so the GC would not collect it during the next cycle.

The object, however, would be marked as finalized, so when it would become eligible again, the finalize method would not be called. In essence, you can turn this “resurrection” trick only once for the lifetime of the object. Beware that this ugly hack should be used only if you really know what you’re doing — however, understanding this trick gives some insight into how the GC works.

## 108) Strong vs weak vs phantom vs soft reference

Much as memory is managed in Java, an engineer may need to perform as much optimization as possible to minimize latency and maximize throughput, in critical applications. Much as it is impossible to explicitly control when garbage collection is triggered in the JVM, it is possible to influence how it occurs as regards the objects we have created.

Java provides us with reference objects to control the relationship between the objects we create and the garbage collector.

By default, every object we create in a Java program is strongly referenced by a variable:

```
1
```

```
StringBuilder sb = new StringBuilder();
```

In the above snippet, the new keyword creates a new StringBuilder object and stores it on the heap. The variable sb then stores a strong reference to this object. What this means for the garbage collector is that the particular StringBuilder object is not eligible for collection at all due to a strong reference held to it by sb. The story only changes when we nullify sb like this:

```
1
```

```
sb = null;
```

After calling the above line, the object will then be eligible for collection.

We can change this relationship between the object and the garbage collector by explicitly wrapping it inside another reference object which is located inside java.lang.ref package.

A soft reference can be created to the above object like this:

```
1
```

```
2
```

```
3
```

```
StringBuilder sb = new StringBuilder();
```

```
SoftReference<StringBuilder> sbRef = new SoftReference<>(sb);
```

```
sb = null;
```

In the above snippet, we have created two references to the StringBuilder object. The first line creates a strong reference sb and the second line creates a soft reference sbRef. The third line should make the object eligible for collection but the garbage collector will postpone collecting it because of sbRef.

The story will only change when memory becomes tight and the JVM is on the brink of throwing an OutOfMemory error. In other words, objects with only soft references are collected as a last resort to recover memory.

A weak reference can be created in a similar manner using WeakReference class. When sb is set to null and the StringBuilder object only has a weak reference, the JVM's garbage collector will have absolutely no compromise and immediately collect the object at the very next cycle.

A phantom reference is similar to a weak reference and an object with only phantom references will be collected without waiting. However, phantom references are enqueued as soon as their objects are collected. We can poll the reference queue to know exactly when the object was collected.

<https://medium.com/@ramtop/weak-soft-and-phantom-references-in-java-and-why-they-matter-c04bfc9dc792>

-----

Present in java.lang.ref Package

```
Counter prime = new Counter(); // prime holds a strong reference -  
line 2  
SoftReference<Counter> soft = new SoftReference<Counter>(prime)  
; //soft reference variable has SoftReference to Counter Object created at line  
2  
prime = null; // now Counter object is eligible for garbage collection  
but only be collected when JVM absolutely needs memory
```

A SoftReference can be used to implement a cache.

An object that is not reachable by a strong reference (that is, not strongly reachable) but is referenced by a soft reference is called softly reachable.

A softly reachable object may be garbage collected at the discretion of the garbage collector.

This generally means that softly reachable objects will only be garbage collected when free memory is low,

but again, it is at the discretion of the garbage collector. Semantically, a soft reference means "keep this object unless the memory is needed."

```
Counter counter = new Counter(); // strong reference - line 1  
WeakReference<Counter> weakCounter = new  
WeakReference<Counter>(counter); //weak reference  
counter = null;
```

we can use weakCounter.get() to get the counter object.

A WeakReference is used to implement weak maps. An object that is not strongly or softly reachable, but is referenced by a weak reference is called weakly reachable.

A weakly reachable object will be garbage collected during the next collection cycle.

This behavior is used in the class java.util.WeakHashMap.

A weak map allows the programmer to put key/value pairs in the map and not worry about the objects taking up memory when the key is no longer reachable anywhere else.

Another possible application of weak references is the string intern pool.

It's actually quite often a bad idea to use weak hashmaps. For one it's easy to get wrong, but even worse it's usually used to implement some kind of cache.

What this does mean is the following: Your program runs fine with good performance for some time, under stress we allocate more and more memory (

more requests = more memory pressure = probably more cache entries) which then leads to a GC.

Now suddenly while your system is under high stress you not only get the GC, but also lose your whole cache, just when you'd need it the most.

Not fun this problem, so you at least have to use a reasonably sized hard referenced LRU cache to mitigate that problem - you can still use the weakrefs then

but only as an additional help.

A PhantomReference is enqueued after finalization of the object. A WeakReference is enqueued before.

A PhantomReference is used to reference objects that have been marked for garbage collection and have been finalized, but have not yet been reclaimed.

An object that is not strongly, softly or weakly reachable, but is referenced by a phantom reference is called phantom reachable.

This allows for more flexible cleanup than is possible with the finalization mechanism alone.

Semantically, a phantom reference means "this object is no longer needed and has been finalized in preparation for being collected."

Mandatory supply a ReferenceQueue instance while creating any WeakReference, SoftReference or PhantomReference as shown in following code :

the phantom references are enqueued once the referenced objects becomes "phantom reachable"

```
ReferenceQueue refQueue = new ReferenceQueue(); //reference will  
be stored in this queue for cleanup
```

```
DigitalCounter digit = new DigitalCounter();
```

```
PhantomReference<DigitalCounter> phantom = new  
PhantomReference<DigitalCounter>(digit, refQueue);
```



Reference of instance will be appended to ReferenceQueue and you can use it to perform any clean-up by polling ReferenceQueue.

Phantom references can be used to perform pre-garbage collection actions such as freeing resources.

Instead, people usually use the finalize() method for this which is not a good idea.

Finalizers have a horrible impact on the performance of the garbage collector and can break data integrity of your application

if you're not very careful since the "finalizer" is invoked in a random thread, at a random time.

The main advantage of using a PhantomReference over finalize() is that finalize() is called by a garbage-collector thread,

meaning it introduces concurrency even in a single-threaded program, with all the potential issues (like correctly synchronizing shared state).

With a PhantomReference, you choose the thread that dequeues references from your queue (in a single-threaded program, that thread could periodically do this job).

### 109) Suppose we have a circular reference (two objects that reference each other). Could such pair of objects become eligible for garbage collection and why?

Yes, a pair of objects with a circular reference can become eligible for garbage collection. This is because of how Java's garbage collector handles circular references. It considers objects live not when they have any reference to them, but when they are reachable by navigating the object graph starting from some garbage collection root (a local variable of a live thread or a static field). If a pair of objects with a circular reference is not reachable from any root, it is considered eligible for garbage collection.

### 110) What is the difference between Serial and Throughput Garbage collector ?

The throughput garbage collector uses a parallel version of the young generation collector and is meant to be used with applications that have medium to large data sets. On the other hand, the serial collector is usually adequate for most small applications (those requiring heaps of up to approximately 100MB on modern processors).

Serial Garbage collector is a stop the world GC which stops application thread from running during both minor and major collection. Serial Garbage collector can be enabled using JVM option `-XX:UseSerialGC` and it's designed for Java application which doesn't have pause time requirement and have client configuration. Serial Garbage collector was also default GC in JDK 1.4 before ergonomics was introduced in JDK 1.5. Serial GC is most suited for small application with less number of thread while throughput GC is more suited for large applications. On the other hand Throughput garbage collector is parallel collector where minor and major collection happens in parallel taking full advantage of all the system resources available like multiple processor. Though both major and minor collection runs on stop-the-world fashion and introduced pause in application. Throughput Garbage collector can be enable using `-XX:UseParallelGC` or `-XX:UseOldParallelGC`. It increases overall throughput of application by minimizing time spent in Garbage collection but still has long pauses during full GC.

### 111) If an object reference is set to null, will the Garbage Collector immediately free the memory held by that object ?

No, the object will be available for garbage collection in the next cycle of the garbage collector.

<https://snowdream.github.io/115-Java-Interview-Questions-and-Answers/115-Java-Interview-Questions-and-Answers/en/collectors.html>

### 112) What is difference between ParNew and DefNew Young Generation Garbage collector?

ParNew and DefNew is two young generation garbage collector. ParNew is a multi-threaded GC used along with concurrent Mark Sweep while DefNew is single threaded GC used along with Serial Garbage Collector.

### 113) How do you find GC resulted due to calling `System.gc()`?

Another GC interview question which is based on GC output. Similar to major and minor collection, there will be a word "System" included in Garbage collection output.

## 114) Does Garbage collection occur in permanent generation space in JVM?

This is a tricky Garbage collection interview question as many programmers are not sure whether PermGen space is part of Java heap space or not and since it maintains class Meta data and String pool, whether its eligible for garbage collection or not. By the way Garbage Collection does occur in PermGen space and if PermGen space is full or cross a threshold, it can trigger Full GC. If you look at output of GC you will find that PermGen space is also garbage collected. This is why correct sizing of PermGen space is important to avoid frequent full GC. You can control size of PermGen space by JVM options -XX:PermGenSize and -XX:MaxPermGenSize.

## 115) How to monitor garbage collection log and what is that?

To diagnose any memory problems, the [Garbage Collection](#) log file is the best place to start. It provides several interesting statistics:

- When the scavenge (or Young generation) GC ran?
- When the full GC ran?
- How many scavenge GCs and Full GCs ran? Did they run repeatedly? In what interval?
- After the GC process ran, how much memory was reclaimed in Young, Old, and Permanent/Metaspace generations?
- How long did the GC run?
- How long did JVM pause when Full GC run?
- What was the total allocated memory in each generation?
- How many objects were promoted to old generation?

## 116) How to generate GC log and what they mean?

```
Command to generate log - -XX:+PrintGCDetails -XX:+PrintGCDateSta  
mps -Xloggc:<file-path>
```

### Example :

```
-XX:+PrintGCDetails -XX:+PrintGCDateStamps -  
Xloggc:/opt/app/gc.log
```

-XX:+PrintGCDateStamps would print the absolute time stamp in the log statement (i.e. "2014-11-18T16:39:25.303-0800").

-XX:+PrintGCDetails property would print the details of how much memory is reclaimed in each generation.

Passing the above system properties would generate a Garbage Collection log file that would look like:

+++++

```
2014-11-18T16:39:25.512-0800: 76.592: [Full GC [PSYoungGen: 26560K->0K(233024
K)] [PSOldGen: 632024K->658428K(699072K)] 658584K->658428K(932096K) [PSPermGe
n: 2379K->2379K(21248K)], 3.0978612 secs] [Times: user=3.09 sys=0.00, real=3.
10 secs]
2014-11-18T16:39:31.536-0800: 82.616: [Full GC [PSYoungGen: 116544K->0K(23302
4K)] [PSOldGen: 658428K->684832K(699072K)] 774972K->684832K(932096K) [PSPermG
en: 2379K->2379K(21248K)], 3.2582136 secs] [Times: user=3.23 sys=0.03, real=3
.26 secs]
2014-11-18T16:39:37.728-0800: 88.808: [Full GC [PSYoungGen: 116544K->12164K(2
33024K)] [PSOldGen: 684832K->699071K(699072K)] 801376K->711236K(932096K) [PSP
ermGen: 2379K->2379K(21248K)], 3.4230220 secs] [Times: user=3.40 sys=0.02, re
al=3.42 secs]
:
:
Heap
  PSYoungGen total 233024K, used 116544K [0x00000000eaab0000, 0x00000001000000
00, 0x0000000100000000)
    eden space 116544K, 100% used [0x00000000eaab0000,0x00000000f1c80000,0x00000
000f1c80000)
    from space 116480K, 0% used [0x00000000f1c80000,0x00000000f1c80000,0x0000000
0f8e40000)
    to space 116480K, 0% used [0x00000000f8e40000,0x00000000f8e40000,0x000000010
0000000)
  PSOldGen total 699072K, used 699071K [0x00000000c0000000, 0x00000000eaab0000
, 0x00000000eaab0000)
    object space 699072K, 99% used [0x00000000c0000000,0x00000000eaaafff0,0x0000
0000eaab0000)
  PSPermGen total 21248K, used 2409K [0x00000000bae00000, 0x00000000bc2c0000,
0x00000000c0000000)
    object space 21248K, 11% used [0x00000000bae00000,0x00000000bb05a740,0x00000
000bc2c0000)
```

+++++

## 117) Antanomy of GC LOG

<https://dzone.com/articles/understanding-garbage-collection-log>

2014-11-18T16:39:37.728-0800: 88.808: [Full GC [PSYoungGen: 116544K->12164K(233024K)] [PSOldGen: 684832K->699071K(699072K)] 801376K->711236K(932096K) [PSPermGen: 2379K->2379K(21248K)], 3.4230220 secs] [Times: user=3.40 sys=0.02, real=3.42 secs]

Let's pick apart this log statement and examine each field in it:

**2014-11-18T16:39:37.728-0800** – Time stamp at which GC ran.

**Full GC** – Type of GC. It could be either 'Full GC' or 'GC'.

**[PSYoungGen: 116544K->12164K(233024K)]** – After the GC ran the young generation, space came down from 116544k (i.e. 113mb) to 12164k (i.e. 12mb). Total allocated young generation space is 233024k (i.e.227mb).

**[PSOldGen: 684832K->699071K(699072K)]** – After the GC ran the old generation space increased from 684832k (i.e. 669mb) to 699071k (i.e. 682mb) and total allocated old generation space is 669072k (i.e. 682mb). In this case after the GC event, old generation's space increased and didn't decrease, which isn't the case always. Here size has increased *because all objects in Old generation are actively referenced + objects from young generation are promoted to old generation. Thus you are seeing the increase in the old generation size.*

**801376K->711236K(932096K)** – After the GC ran, overall memory came down from 801376k to 711236k. Total allocated memory space is 932096k (i.e. 910mb).

**[PSPermGen: 2379K->2379K(21248K)]** – After the GC ran, there was no drop in perm generation space.

**3.4230220 secs** – the GC took 3.42 seconds to complete.

**[Times: user=3.40 sys=0.02, real=3.42 secs]** – **Real** is wall clock time (time from start to finish of the call). This is all elapsed time including time slices used by other processes and time the process spends blocked (for example if it is waiting for I/O to complete).

**User** is the amount of CPU time spent in user-mode code (outside the kernel) within the process. This is only actual CPU time used in executing the process. Other processes, and the time the process spends blocked, do not count towards this figure.

**Sys** is the amount of CPU time spent in the kernel within the process. This means executing CPU time spent in system calls within the kernel, as opposed to library code, which is still running in user-space. Like **user**, this is only CPU time used by the process.

In your case, if the CPU time (3.4 sec) is considerably higher than the real time passed (3.42 Sec), we can conclude that the GC was run using multiple threads. To learn more about the difference between each of these Times, please [check out this article](#).

## GENERIC

### 118) Can you explain Type Erasure?

Answer: It is nothing but a JVM phenomenon, which means that the runtime has no idea about the types of generic objects like List<Integer>.

### 119) What Is a Generic Type Parameter?

Type is the name of a class or interface. As implied by the name, a generic type parameter is when a type can be used as a parameter in a class, method or interface declaration.

Let's start with a simple example, one without generics, to demonstrate this

```
public interface Consumer {  
    public void consume(String parameter)  
}
```

In this case, the method parameter type of the consume() method is String. It is not parameterized and not configurable. Now let's replace our String type with a generic type that we will call T. It is named like this by convention:

```
public interface Consumer<T> {  
    public void consume(T parameter)  
}
```

When we implement our consumer, we can provide the type that we want it to consume as an argument. This is a generic type parameter:

```
public class IntegerConsumer implements Consumer<Integer> {  
    public void consume(Integer parameter)  
}
```

In this case, now we can consume integers. We can swap out this type for whatever we require.

## 120) What Are Some Advantages of Using Generic Types?

One advantage of using generics is avoiding costs and provide type safety. This which is particularly useful when working with collections. Let's demonstrate this:

Let's demonstrate this:

```
List list = new ArrayList();
```

```
list.add("foo");
```

```
Object o = list.get(1);
```

```
String foo = (String) foo;
```

In our example, the element type in our list is unknown to the compiler. This means that the only thing that can be guaranteed is that it is an object. So when we retrieve our element, an object is what we get back. As the authors of the code, we know it's a String, but we have to cast our object to one to fix the problem explicitly. This produces a lot of noise and boilerplate.

Next, if we start to think about the room for manual error, the casting problem gets worse. What if we accidentally had an integer in our list?

```
list.add(1)
```

```
Object o = list.get(1);
```

```
String foo = (String) foo;
```

In this case, we would get a ClassCastException at runtime, as an Integer cannot be cast to String. Now, let's try repeating ourselves, this time using generics:

```
List<String> list = new ArrayList<>();
```

```
list.add("foo");
```

```
String o = list.get(1); // No cast
```

```
Integer foo = list.get(1); // Compilation error
```

As we can see, by using generics we have a compile type check which prevents ClassExceptions and removes the need for casting.

The other advantage is to avoid code duplication. Without generics, we have to copy and paste the same code but for different types. With generics, we do not have to do this. We can even implement algorithms which apply to generic types.

## 121) What Is Type Erasure?

It's important to realize that generic type information is only available to the compiler, not the JVM. In other words, type erasure means that generic type information is not available to the JVM at runtime, only compile time.

The reasoning behind major implementation choice is simple – preserving backward compatibility with older versions of Java. When a generic code is compiled into bytecode, it will be as if the generic type never existed. This means that compilation will:

Replace generic types with objects

Replace bounded types (More on these in a later question) with the first bound class

Insert the equivalent of casts when retrieving generic objects.

It's important to understand type erasure. Otherwise, a developer might get confused and think they'd be able to get the type at runtime:

```
public foo(Consumer<T> consumer) {  
    Type type = consumer.getGenericTypeParameter()  
}
```

The above example is a pseudo code equivalent of what things might look like without type erasure, but unfortunately, it is impossible. Once again, the generic type information is not available at runtime.

## 122) If a Generic Type Is Omitted When Instantiating an Object, Will the Code Still Compile?

As generics did not exist before Java 5, it is possible not to use them at all. For example, generics were retrofitted to most of the standard Java classes such as collections. If we look at our list from question one, then we will see that we already have an example of omitting the generic type:

```
List list = new ArrayList();
```

Despite being able to compile, it's still likely that there will be a warning from the compiler. This is because we are losing the extra compile time check that we get from using generics.

The point to remember is that while backward compatibility and type erasure make it possible to omit generic types, it is bad practice.



## 123) What is generic method?

Generic methods are those methods that are written with a single method declaration and can be called with arguments of different types. The compiler will ensure the correctness of whichever type is used. These are some properties of generic methods:

Generic methods have a type parameter (the diamond operator enclosing the type) before the return type of the method declaration

Type parameters can be bounded (bounds are explained later in the article)

Generic methods can have different type parameters separated by commas in the method signature

Method body for a generic method is just like a normal method

An example of defining a generic method to convert an array to a list:

```
public <T> List<T> fromArrayToList(T[] a) {
    return Arrays.stream(a).collect(Collectors.toList());
}
```

In the previous example, the <T> in the method signature implies that the method will be dealing with generic type T. This is needed even if the method is returning void.

As mentioned above, the method can deal with more than one generic type, where this is the case, all generic types must be added to the method signature, for example, if we want to modify the above method to deal with type T and type G, it should be written like this:

```
1
2         public static <T, G> List<G> fromArrayToList(T[] a, Function<T, G> mapper) {
3             return Arrays.stream(a)
4                 .map(mapperFunction)
5                 .collect(Collection)}
```

## 124) How Does a Generic Method Differ From a Generic Type?

A generic method is where a type parameter is introduced to a method, living within the scope of that method. Let's try this with an example:

```
public static <T> T returnType(T argument) {
    return argument;
}
```

We've used a static method but could have also used a non-static one if we wished. By leveraging type inference (covered in the next question), we can

invoke this like any ordinary method, without having to specify any type arguments when we do so.

## 125) What Is Type Inference?

Type inference is when the compiler can look at the type of a method argument to infer a generic type. For example, if we passed in T to a method which returns T, then the compiler can figure out the return type. Let's try this out by invoking our generic method from the previous question:

```
Integer inferredInteger = returnType(1);
```

```
String inferredString = returnType("String");
```

As we can see, there's no need for a cast, and no need to pass in any generic type argument. The argument type only infers the return type.

## 126) What is a Bounded Type Parameter?

So far all our questions have covered generic types arguments which are unbounded. This means that our generic type arguments could be any type that we want.

When we use bounded parameters, we are restricting the types that can be used as generic type arguments.

As an example, let's say we want to force our generic type always to be a subclass of animal:

```
public abstract class Cage<T extends Animal> {  
    abstract void addAnimal(T animal)  
}
```

By using extends, we are forcing T to be a subclass of animal. We could then have a cage of cats:

```
Cage<Cat> catCage;
```

But we could not have a cage of objects, as an object is not a subclass of an animal:

```
Cage<Object> objectCage; // Compilation error
```

One advantage of this is that all the methods of animal are available to the compiler. We know our type extends it, so we could write a generic algorithm which operates on any animal. This means we don't have to reproduce our method for different animal subclasses:

```
public void firstAnimalJump() {  
    T animal = animals.get(0);  
    animal.jump();  
}
```

## 127) Is It Possible to Declared a Multiple Bounded Type Parameter?

Declaring multiple bounds for our generic types is possible. In our previous example, we specified a single bound, but we could also specify more if we wish:

```
public abstract class Cage<T extends Animal & Comparable>
```

In our example, the animal is a class and comparable is an interface. Now, our type must respect both of these upper bounds. If our type were a subclass of animal but did not implement comparable, then the code would not compile. It's also worth remembering that if one of the upper bounds is a class, it must be the first argument.

## 128) What Is a Wildcard Type?

A wildcard type represents an unknown type. It's denoted with a question mark as follows:

```
public static consumeListOfWildcardType(List<?> list)
```

Here, we are specifying a list which could be of any type. We could pass a list of anything into this method.

## 129) What is an Upper Bounded Wildcard?

An upper bounded wildcard is when a wildcard type inherits from a concrete type. This is particularly useful when working with collections and inheritance.

Let's try demonstrating this with a farm class which will store animals, first without the wildcard type:

```
public class Farm {  
    private List<Animal> animals;  
    public void addAnimals(Collection<Animal> newAnimals) {  
        animals.addAll(newAnimals);  
    }  
}
```

```
}  
}
```

If we had multiple subclasses of animal, such as cat and dog, we might make the incorrect assumption that we can add them all to our farm:

```
farm.addAnimals(cats); // Compilation error
```

```
farm.addAnimals(dogs); // Compilation error
```

This is because the compiler expects a collection of the concrete type animal, not one of its subclasses.

Now, let's introduce an upper bounded wildcard to our add animals method:

```
public void addAnimals(Collection<? extends Animal> newAnimals)
```

Now if we try again, our code will compile. This is because we are now telling the compiler to accept a collection of any subtype of animal.

### 130) What is an Unbounded Wildcard?

An unbounded wildcard is a wildcard with no upper or lower bound, that can represent any type.

It's also important to know that the wildcard type is not synonymous to object. This is because a wildcard can be any type whereas an object type is specifically an object (and cannot be a subclass of an object). Let's demonstrate this with an example:

```
List<?> wildcardList = new ArrayList<String>();
```

```
List<Object> objectList = new ArrayList<String>(); // Compilation error
```

Again, the reason the second line does not compile is that a list of objects is required, not a list of strings. The first line compiles because a list of any unknown type is acceptable.

### 131) What Is a Lower Bounded Wildcard?

A lower bounded wildcard is when instead of providing an upper bound, we provide a lower bound by using the super keyword. In other words, a lower bounded wildcard means we are forcing the type to be a superclass of our bounded type. Let's try this with an example:

```
public static void addDogs(List<? super Animal> list) {
    list.add(new Dog("tom"))
}
```

By using super, we could call addDogs on a list of objects:

```
ArrayList<Object> objects = new ArrayList<>();
addDogs(objects);
```

This makes sense, as an object is a superclass of animal. If we did not use the lower bounded wildcard, the code would not compile, as a list of objects is not a list of animals.

If we think about it, we wouldn't be able to add a dog to a list of any subclass of animal, such as cats, or even dogs. Only a superclass of animal. For example, this would not compile:

```
ArrayList<Cat> objects = new ArrayList<>();
addDogs(objects);
```

### 132) Q13. When Would You Choose to Use a Lower Bounded Type vs. an Upper Bounded Type?

When dealing with collections, a common rule for selecting between upper or lower bounded wildcards is PECS. PECS stands for producer extends, consumer super.

This can be easily demonstrated through the use of some standard Java interfaces and classes.

Producer extends just means that if you are creating a producer of a generic type, then use the extends keyword. Let's try applying this principle to a collection, to see why it makes sense:

```
public static void makeLotsOfNoise(List<? extends Animal> animals) {
    animals.forEach(Animal::makeNoise);
}
```

Here, we want to call `makeNoise()` on each animal in our collection. This means our collection is a producer, as all we are doing with it is getting it to return animals for us to perform our operation on. If we got rid of `extends`, we wouldn't be able to pass in lists of cats, dogs or any other subclasses of animals. By applying the producer extends principle, we have the most flexibility possible.

Consumer super means the opposite to producer extends. All it means is that if we are dealing with something which consumes elements, then we should use the `super` keyword. We can demonstrate this by repeating our previous example:

```
public static void addCats(List<? super Animal> animals) {
    animals.add(new Cat());
}
```

We are only adding to our list of animals, so our list of animals is a consumer. This is why we use the `super` keyword. It means that we could pass in a list of any superclass of animal, but not a subclass. For example, if we tried passing in a list of dogs or cats then the code would not compile.

The final thing to consider is what to do if a collection is both a consumer and a producer. An example of this might be a collection where elements are both added and removed. In this case, an unbounded wildcard should be used.

### 133) Are There Any Situations Where Generic Type Information Is Available at Runtime?

There is one situation where a generic type is available at runtime. This is when a generic type is part of the class signature like so:

```
public class CatCage implements Cage<Cat>
```

By using reflection, we get this type parameter:

```
(Class<T>) ((ParameterizedType) getClass()  
.getGenericSuperclass()).getActualTypeArguments()[0];
```

This code is somewhat brittle. For example, it's dependant on the type parameter being defined on the immediate superclass. But, it demonstrates the JVM has does have this type information.

## STRING

### 134) Difference between String , string buffer and string builder?

Main difference between String and StringBuilder/StringBuffer is that String is immutable, i.e. a String you once have initialized cannot be changed anymore. Redefining a String just creates a new immutable String in memory. Instances of StringBuilder/StringBuffer can be modified by methods like append(String str) or insert(StringBuffer sb).

The only difference between StringBuilder/StringBuffer is that StringBuilder is not safe for use by multiple threads, StringBuffer instead is thread-safe. String builder is even faster than String.

EXERCISE :use string builder in multithreaded environment

<http://javarevisited.blogspot.in/2010/10/why-string-is-immutable-in-java.html>

### 135) Can we use equals in Stringbuilder and buffer?

No, these class don't override equals methods, so to compare them use sb.toString.

### 136) SubString() and memory leaks ?

In past versions of the JDK, the implementation of the substring method would build a new String object keeping a reference to the whole char array, to avoid copying it. You could thus inadvertently keep a reference to a very big character array with just a one character string. This method has now been changed and this "leak" doesn't exist anymore.

If you want to use an old JDK (that is older than OpenJDK 7, Update 6) and you want to have minimal strings after substring, use the constructor taking another string :

```
String s2 = new String(s1.substring(0,1));
```

<http://javarevisited.blogspot.in/2011/10/how-substring-in-java-works.html>

### 137) Why char[] is used to store password ?

1) Since Strings are immutable in Java if you store password as plain text it will be available in memory until Garbage collector clears it and since String are used in String pool for reusability there is pretty high chance that it will be remain in memory for long duration, which pose a security threat. Since any one who has access to memory dump can find the password in clear text and that's another reason you should always used an encrypted password than plain text.

Since Strings are immutable there is no way contents of Strings can be changed because any change will produce new String, while if you char[] you can still set all his element as blank or zero. So Storing password in character array clearly mitigates security risk of stealing password.

2) Java itself recommends using getPassword() method of JPasswordField which returns a char[] and deprecated getText() method which returns password in clear text stating security reason. Its good to follow advice from Java team and adhering to standard rather than going against it.

### 138) What in String intern method?

According to String#intern(), intern method is supposed to return the String from the String pool if the String is found in String pool,otherwise a new string object will be added in String pool and the reference of this String is returned. Java automatically interns String literals. This means that in many cases, the == operator appears to work for Strings in the same way that it does for ints or other primitive values.Since interning is automatic for String literals, the intern() method is to be used on Strings constructed with new String()

<https://dzone.com/articles/string-interning-what-why-and>

<https://www.geeksforgeeks.org/interning-of-string/>



String class is designed with the **Flyweight** design pattern in mind. Flyweight is all about re-usability without having to create too many objects in memory.

A pool of Strings is maintained by the String class. When the *intern()* method is invoked, **equals(..)** method is invoked to determine if the String already exist in the pool. If it does then the String from the pool is returned instead of creating a new object. If not already in the string pool, a new String object is added to the pool and a reference to this object is returned. For any two given strings s1 & s2, *s1.intern() == s2.intern()* only if *s1.equals(s2)* is true.

Two String objects are created by the code shown below. Hence *s1 == s2* returns false.

//Two new objects are created. Not interned and not recommended.

```
String s1 = new String("A");
```

```
String s2 = new String("A");
```

```
1 //Two new objects are created. Not interned and not recommended.
2 String s1 = new String("A");
3 String s2 = new String("A");
4
```

*s1.intern() == s2.intern()* returns **true**, but you have to remember to make sure that you actually do *intern()* all of the strings that you're going to compare. It's easy to forget to *intern()* all strings and then you can get confusingly incorrect results. Also, why unnecessarily create more objects? Instead use string literals as shown below to intern automatically:

```
String s1 = "A";
```

```
String s2 = "A";
```



```
1 String s1 = "A";
2 String s2 = "A";
3
```

s1 and s2 point to the same String object in the pool. Hence s1 == s2 returns true.

Since interning is automatic for String literals String s1 = "A", the *intern()* method is to be used on Strings constructed with new String("A").

When to use ??

when you need speed since you can compare strings by reference (== is faster than equals)

Disadvantages:

1. You might forget to intern all strings
2. Expensive as String pool need to be maintained
3. interned Strings live in PermGen space, which is usually quite small; you may run into an OutOfMemoryError with plenty of free heap space.(Java 7 or less)

## 139) Creation of String?

Let's look at a couple of examples of how a String might be created, and let's further assume that no other String objects exist in the pool:

```
String s = "abc"; // creates one String object and one  
// reference variable
```

In this simple case, "abc" will go in the pool and s will refer to it.

```
String s = new String("abc"); // creates two objects,  
// and one reference variable
```

In this case, because we used the `new` keyword, Java will create a new String object in normal (nonpool) memory, and s will refer to it. In addition, the literal "abc" will be placed in the pool.

## 140) Common method of String?

**charAt()** Returns the character located at the specified index

- **concat()** Appends one String to the end of another ( "+" also works)
- **equalsIgnoreCase()** Determines the equality of two Strings, ignoring case
- **length()** Returns the number of characters in a String
- **replace()** Replaces occurrences of a character with a new character
- **substring()** Returns a part of a String
- **toLowerCase()** Returns a String with uppercase characters converted
- **toString()** Returns the value of a String
- **toUpperCase()** Returns a String with lowercase characters converted
- **trim()** Removes whitespace from the ends of a String

## 141) Can you tell me the reason why String class is considered immutable?

Answer: For better performance and thread safety. It is because to avoid change in String object once it is created. As String is immutable, you can share it between different threads in a safe way. This is quite crucial in multithreaded programming.

1. **Performance: Immutable** objects are ideal for representing values of abstract data (i.e. value objects) types like numbers, enumerated types, etc. If you need a different value, create a different object. In Java, *Integer*, *Long*, *Float*, *Character*, *BigInteger* and *BigDecimal* are all immutable objects. Optimization strategies like caching of hashcode, caching of objects, object pooling, etc can be easily applied to improve performance. If Strings were made mutable, string pooling would not be possible as changing the string with one reference will lead to the wrong value for the other references.
2. **Thread safety** as immutable objects are inherently thread safe as they cannot be modified once created. They can only be used as read only objects. They can easily be shared among multiple threads for better scalability.

## 142) What is String in Java? String is a data type?

String is a Class in java and defined in java.lang package. It's not a primitive data type like int and long. String class represents character Strings. String is used in almost all the Java applications and there are some interesting facts we should know about String. String is immutable and final in Java and JVM uses String Pool to store all the String objects.

Some other interesting things about String is the way we can instantiate a String object using double quotes and overloading of "+" operator for concatenation.

A String instance in Java is an object with two fields: a char[] value field and an int hash field. The value field is an array of chars representing the string itself, and the hash field contains the hashCode of a string which is initialized with zero, calculated during the first hashCode() call and cached ever since. As a curious edge case, if a hashCode of a string has a zero value, it has to be recalculated each time the hashCode() is called.

Important thing is that a String instance is immutable: you can't get or modify the underlying char[] array. Another feature of strings is that the static constant strings are loaded and cached in a string pool. If you have multiple identical String objects in your source code, they are all represented by a single instance at runtime.

## 143) StringBuilder use case?

Q15. What is a StringBuilder and what are its use cases? What is the difference between appending a string to a StringBuilder and concatenating two strings with a + operator? How does StringBuilder differ from StringBuffer?

StringBuilder allows manipulating character sequences by appending, deleting and inserting characters and strings. This is a mutable data structure, as opposed to the String class which is immutable.

When concatenating two String instances, a new object is created, and strings are copied. This could bring a huge garbage collector overhead if we need to create or modify a string in a loop. StringBuilder allows handling string manipulations much more efficiently.

StringBuffer is different from StringBuilder in that it is thread-safe. If you need to manipulate a string in a single thread, use StringBuilder instead.

#### 144) How can we make String upper case or lower case?

We can use String class `toUpperCase` and `toLowerCase` methods to get the String in all upper case or lower case. These methods have a variant that accepts Locale argument and use that locale rules to convert String to upper or lower case.

#### 145) What is String subSequence method?

Java 1.4 introduced CharSequence interface and String implements this interface, this is the only reason for the implementation of subSequence method in String class. Internally it invokes the String substring method. Check this post for [String subSequence](#) example.

#### 146) How to compare two Strings in java program?

Java String implements `Comparable` interface and it has two variants of `compareTo()` methods.

`compareTo(String anotherString)` method compares the String object with the String argument passed lexicographically. If String object precedes the argument passed, it returns negative integer and if String object follows the argument String passed, it returns positive integer. It returns zero when both the String have same value, in this case `equals(String str)` method will also return true.

`compareToIgnoreCase(String str)`: This method is similar to the first one, except that it ignores the case. It uses String `CASE_INSENSITIVE_ORDER` Comparator for case insensitive comparison. If the value is zero then `equalsIgnoreCase(String str)` will also return true.

Check this post for [String compareTo](#) example.

#### 147) How to convert String to char and vice versa?

This is a tricky question because String is a sequence of characters, so we can't convert it to a single character. We can use `charAt` method to get the character at given index or we can use `toCharArray()` method to convert String to character array.

Check this post for sample program on converting [String to character array to String](#).

#### 148) How to convert String to byte array and vice versa?

We can use String `getBytes()` method to convert String to byte array and we can use String constructor `new String(byte[] arr)` to convert byte array to String. Be careful to mention encoding, else it will return different output for different platform.

Check this post for [String to byte array](#) example.

### 149) Can we use String in switch case?

This is a tricky question used to check your knowledge of current Java developments. Java 7 extended the capability of switch case to use Strings also, earlier java versions doesn't support this.

If you are implementing conditional flow for Strings, you can use if-else conditions and you can use switch case if you are using Java 7 or higher versions.

Check this post for [Java Switch Case String](#) example.

### 150) Difference between String, String builder and String buffer?

String is immutable and final in java, so whenever we do String manipulation, it creates a new String. String manipulations are resource consuming, so java provides two utility classes for String manipulations – StringBuffer and StringBuilder.

StringBuffer and StringBuilder are mutable classes. StringBuffer operations are thread-safe and synchronized where StringBuilder operations are not thread-safe. So when multiple threads are working on same String, we should use StringBuffer but in single threaded environment we should use StringBuilder.

StringBuilder performance is fast than StringBuffer because of no overhead of synchronization.

Check this post for extensive details about [String vs StringBuffer vs StringBuilder](#).

Read this post for benchmarking of [StringBuffer vs StringBuilder](#).

### 151) Why String is immutable or final in Java

There are several benefits of String because it's immutable and final.

- String Pool is possible because String is immutable in java.
- It increases security because any hacker can't change its value and it's used for storing sensitive information such as database username, password etc.
- Since String is immutable, it's safe to use in multi-threading and we don't need any synchronization.
- Strings are used in java classloader and immutability provides security that correct class is getting loaded by Classloader.

Check this post to get more details [why String is immutable in java](#).

### 152) How to Split String in java?

We can use `split(String regex)` to split the String into String array based on the provided regular expression.

Learn more at [java String split](#).

### 153) Why Char array is preferred over String for storing password?

String is immutable in java and stored in String pool. Once it's created it stays in the pool until unless garbage collected, so even though we are done with password it's available in memory for longer duration and there is no way to avoid it. It's a security risk because anyone having access to memory dump can find the password as clear text.

If we use char array to store password, we can set it to blank once we are done with it. So we can control for how long it's available in memory that avoids the security threat with String.

### 154) How do you check if two Strings are equal in Java?

There are two ways to check if two Strings are equal or not – using “==” operator or using `equals` method. When we use “==” operator, it checks for value of String as well as reference but in our programming, most of the time we are checking equality of String for value only. So we should use `equals` method to check if two Strings are equal or not.

There is another function `equalsIgnoreCase` that we can use to ignore case.

```
String s1 = "abc";
String s2 = "abc";
String s3= new String("abc");
System.out.println("s1 == s2 ? "+(s1==s2)); //true
System.out.println("s1 == s3 ? "+(s1==s3)); //false
System.out.println("s1 equals s3 ? "+(s1.equals(s3)));
//true
```

### 155) Can you list all strings from string pool?

You are not able to access the string pool from Java code, at least not in the HotSpot implementation of Java VM.

String pool in Java is implemented using [string interning](#). According to [JLS §3.10.5](#):

a string literal always refers to the same instance of class `String`. This is because string literals - or, more generally, strings that are the values of constant expressions (§15.28) - are "interned" so as to share unique instances, using the method `String.intern`.

And according to [JLS §15.28](#):

Compile-time constant expressions of type `String` are always "interned" so as to share unique instances, using the method `String.intern`.

`String.intern` is a native method, as we can see in [its declaration in OpenJDK](#):  
`public native String intern();`

[The native code for this method](#) calls [JVM InternString](#) function.

```
JVM_ENTRY(jstring, JVM_InternString(JNIEnv *env, jstring str))
    JVMWrapper("JVM_InternString");
    JvmtiVMObjectAllocEventCollector oam;
    if (str == NULL) return NULL;
    oop string = JNIHandles::resolve_non_null(str);
    oop result = StringTable::intern(string, CHECK_NULL);
    return (jstring) JNIHandles::make_local(env, result);
JVM_END
```

That is, string interning is implemented using native code, and there's no Java API to access the string pool directly. You may, however, be able to write a native method yourself for this purpose.

### 156) What is String pool?

As the name suggests, String Pool is a pool of Strings stored in [Java heap memory](#). We know that String is special class in java and we can create String object using new operator as well as providing values in double quotes. Check this post for more details about [String Pool](#).

### 157) Why String is popular HashMap key in Java?

Since String is immutable, its hashCode is cached at the time of creation and it doesn't need to be calculated again. This makes it a great candidate for key in a Map and it's processing is fast than other HashMap key objects. This is why String is mostly used Object as HashMap keys.

## INNER CLASS



Inner class means one class which is a member of another class. There are basically four types of inner classes in java.

- 1) Nested Inner class
- 2) Method Local inner classes
- 3) Anonymous inner classes
- 4) Static nested classes

**Nested Inner class** can access any private instance variable of outer class. Like any other instance variable, we can have access modifier private, protected, public and default modifier.

Like class, interface can also be nested and can have access specifiers.

Following example demonstrates a nested class.

```
class Outer {
    // Simple nested inner class
    class Inner {
        public void show() {
            System.out.println("In a nested class
method");
        }
    }
}
class Main {
    public static void main(String[] args) {
        Outer.Inner in = new Outer().new Inner();
        in.show();
    }
}
```

Output:

In a nested class method

As a side note, we can't have static method in a nested inner class because an inner class is implicitly associated with an object of its outer class so it cannot define any static method for itself. For example the following program doesn't compile.

edit

```
class Outer {  
    void outerMethod() {  
        System.out.println("inside outerMethod");  
    }  
    class Inner {  
        public static void main(String[] args){  
            System.out.println("inside inner class Method");  
        }  
    }  
}
```

Output:

Error illegal static declaration in inner class  
Outer.Inner public static void main(String[] args)  
modifier 'static' is only allowed in constant  
variable declaration

An interface can also be nested and nested interfaces have some interesting properties. We will be covering nested interfaces in the next post.

### **Method Local inner classes**

Inner class can be declared within a method of an outer class. In the following example, Inner is an inner class in outerMethod().

edit

```
class Outer {
    void outerMethod() {
        System.out.println("inside outerMethod");
        // Inner class is local to outerMethod()
        class Inner {
            void innerMethod() {
                System.out.println("inside innerMethod");
            }
        }
        Inner y = new Inner();
        y.innerMethod();
    }
}

class MethodDemo {
    public static void main(String[] args) {
        Outer x = new Outer();
        x.outerMethod();
    }
}
```

Output

Inside outerMethod

Inside innerMethod

Method Local inner classes can't use local variable of outer method until that local variable is not declared as final. For example, the following code generates compiler error (Note that x is not final in outerMethod() and innerMethod() tries to access it)

[edit](#)

```
class Outer {
    void outerMethod() {
        int x = 98;
        System.out.println("inside outerMethod");
        class Inner {
            void innerMethod() {
                System.out.println("x= "+x);
            }
        }
        Inner y = new Inner();
        y.innerMethod();
    }
}

class MethodLocalVariableDemo {
    public static void main(String[] args) {
        Outer x=new Outer();
        x.outerMethod();
    }
}
```

Output:

local variable x is accessed from within inner class;  
needs to be declared final

**Note :** Local inner class cannot access non-final local variable till JDK 1.7.  
Since JDK 1.8, it is possible to access the non-final local variable in method  
local inner class.

But the following code compiles and runs fine (Note that x is final this time)

edit

```
class Outer {
    void outerMethod() {
        final int x=98;
        System.out.println("inside outerMethod");
        class Inner {
            void innerMethod() {
                System.out.println("x = "+x);
            }
        }
        Inner y = new Inner();
        y.innerMethod();
    }
}

class MethodLocalVariableDemo {
    public static void main(String[] args){
        Outer x = new Outer();
        x.outerMethod();
    }
}
```

Output-:

Inside outerMethod

X = 98

The main reason we need to declare a local variable as a final is that local variable lives on stack till method is on the stack but there might be a case the object of inner class still lives on the heap.

Method local inner class can't be marked as private, protected, static and transient but can be marked as abstract and final, but not both at the same time.

### **Static nested classes**

Static nested classes are not technically an inner class. They are like a static member of outer class.

edit

```
class Outer {  
    private static void outerMethod() {  
        System.out.println("inside outerMethod");  
    }  
  
    // A static inner class  
    static class Inner {  
        public static void main(String[] args) {  
            System.out.println("inside inner class Method");  
            outerMethod();  
        }  
    }  
}
```

Output

```
inside inner class Method  
inside outerMethod
```

### **Anonymous inner classes**

Anonymous inner classes are declared without any name at all. They are created in two ways.

**a)** *As subclass of specified type*

edit

```
class Demo {
    void show() {
        System.out.println("i am in show method of super
class");
    }
}
class Flavor1Demo {

    // An anonymous class with Demo as base class
    static Demo d = new Demo() {
        void show() {
            super.show();
            System.out.println("i am in Flavor1Demo
class");
        }
    };

    public static void main(String[] args){
        d.show();
    }
}
```

Output

i am in show method of super class

i am in Flavor1Demo class

In the above code, we have two class Demo and Flavor1Demo. Here demo act as super class and anonymous class acts as a subclass, both classes have a method show(). In anonymous class show() method is overridden.

**a)** *As implemter of the specified interface*

edit

```
class Flavor2Demo {

    // An anonymous class that implements Hello interface
    static Hello h = new Hello() {
        public void show() {
            System.out.println("i am in anonymous
class");
        }
    };

    public static void main(String[] args) {
        h.show();
    }
}

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

Output:

**i am in anonymous class**

In above code we create an object of anonymous inner class but this anonymous inner class is an implementer of the interface Hello. Any anonymous inner class can implement only one interface at one time. It can either extend a class or implement interface at a time.

## 158) What is inner class in java?

We can define a class inside a class and they are called nested classes. Any non-static nested class is known as inner class. Inner classes are associated with the object of the class and they can access all the variables and methods of the outer class. Since inner classes are associated with instance, we can't have any static variables in them.

We can have local inner class or anonymous inner class inside a class. For more details read [java inner class](#).

## 159) What is anonymous inner class?

A local inner class without name is known as anonymous inner class. An anonymous class is defined and instantiated in a single statement. Anonymous inner class always extend a class or implement an interface.

Since an anonymous class has no name, it is not possible to define a constructor for an anonymous class. Anonymous inner classes are accessible only at the point where it is defined.



<http://java-questions.com/InnerClass-interview-questions.html>

there are two types of inner classes in java

- 1.Static inner classes
- 2.Non-static inner classes

Non-static inner classes are:

- 1.member inner class
- 2.local inner class
- 3.annonymous inner class

A static member class behaves much like an ordinary top-level class, except that it can access the static members of the class that contains it.

The static nested class can be accessed as the other static members of the enclosing class without having an instance of the outer class.

The static class can contain non-static and static members and methods.

```
InnerClass.StaticInner staticObj= new InnerClass. StaticInner ();
```

Member class - Classes declared outside a function (hence a "member") and not declared "static".

The member class can be declared as public, private, protected, final and abstract

Method local class – The inner class declared inside the method is called method local inner class.

**\*\*Method local inner class can only be declared as final or abstract.**

**\*\*Method local class can only access global variables or method local variables if declared as final**

Since java 1.8 version, method local inner class can access non final method local variable

Anonymous inner class - These are local classes which are automatically declared and instantiated in the middle of an expression.

\*Also, like local classes, anonymous classes cannot be public, private, protected, or static.

\*They can specify arguments to the constructor of the superclass, but cannot otherwise have a constructor.

\*They can implement only one interface or extend a class.

\*Anonymous class cannot define any static fields, methods, or classes, except for static final constants.

Outer class instance variable can be accessed in inner class as inner class has a link to outer class.

you can use OuterClass.this to return reference of outer class

If you want to access inner class object directly you have to use outer class object

```
Outer obj = new Outer();
```

```
Outer.Inner objInner = obj.new Inner();
```

/\*Local inner class can only be abstract or final.\*/Local inner class Cannot be accessed outside method/block scope as it is a part of method.

Argument must be final to use inside anonymous inner class

```
package misc;
```

```
interface AB{  
    int i=5;  
    void print();  
}
```

```
class BC{  
  
    public AB getAB(){  
        return new AB(){  
            int y = 5;  
  
            @Override  
            public void print() {  
                // TODO Auto-generated method stub  
                System.out.println("Hey You");  
            }  
        };  
    }  
}
```

```
public class MethodInnerClassTest {  
  
    public AB getAB(){  
  
        class ABimpl implements AB{  
  
            void show(){  
                System.out.println("hey");  
            }  
        }  
    }  
}
```

```
        @Override
        public void print() {
            // TODO Auto-generated method stub
            System.out.println("Hey");
        }

    }

    return new ABimpl();
}

public static void main(String[] args) {
    MethodInnerClassTest obj = new MethodInnerClassTest();
    AB objAB = obj.getAB();
    System.out.println(objAB.i);
    objAB.print();

    BC bc = new BC();
    bc.getAB().print();

}

}
```

If you don't need a connection between the inner class object and the outer class object, then you can make the inner class static.

Constructor for anonymous inner class:

```
public class Parcel9 {  
    public Destination dest(final String dest, final float price) {  
        return new Destination() {  
            private int cost;  
            // Instance initialization for each object:  
            {  
                cost = Math.round(price);  
                if(cost > 100)  
                    System.out.println("Over budget!");  
            }  
            private String label = dest;  
            public String readLabel() { return label; }  
        };  
    }  
    public static void main(String[] args) {  
        Parcel9 p = new Parcel9();  
        Destination d = p.dest("Tanzania", 101.395F);  
    }  
}
```

an inner class has automatic access to the members of the enclosing class.

A static inner class

means:

1. You don't need an outer-class object in order to create an object of a static inner class.
2. You can't access an outer-class object from an object of a static inner class.

In an ordinary (non-static) inner class, the link to the outer class object is achieved with a special this reference.

A static inner class does not have this special this reference, which makes it analogous to a static method.

An interface can contain static inner class .

Since java 1.8 version, method local inner class can access non final method local variable.

why inner classes ????

-----Each inner class can independently inherit from an implementation.

Thus, the inner class is not limited by whether the outer class is already inheriting from an implementation.

A closure is a callable object that retains information from the scope in which it was created.

From this definition, you can see that an inner class is an object-oriented closure,

because it doesn't just contain each piece of information from the outer-class object

("the scope in which it was created"), but it automatically holds a reference back to the whole outer-class object,

where it has permission to manipulate all the members, even private ones.

```
class outer{  
    class inner{  
    }  
}
```

```
class test extends outer.inner{
```

```
test(Outer o)
{
    o.super();
}

}
```

In this case you would need to give reference as shown above otherwise it wont compile.

## CLASS LOADER

### 160) What is Classloader in Java?

Java Classloader is the program that loads byte code program into memory when we want to access any class. We can create our own classloader by extending `ClassLoader` class and overriding `loadClass(String name)` method. Learn more at [java classloader](#).

### 161) What are different types of classloaders?

There are three types of built-in Class Loaders in Java:

**Bootstrap Class Loader** – It loads JDK internal classes, typically loads `rt.jar` and other core classes.

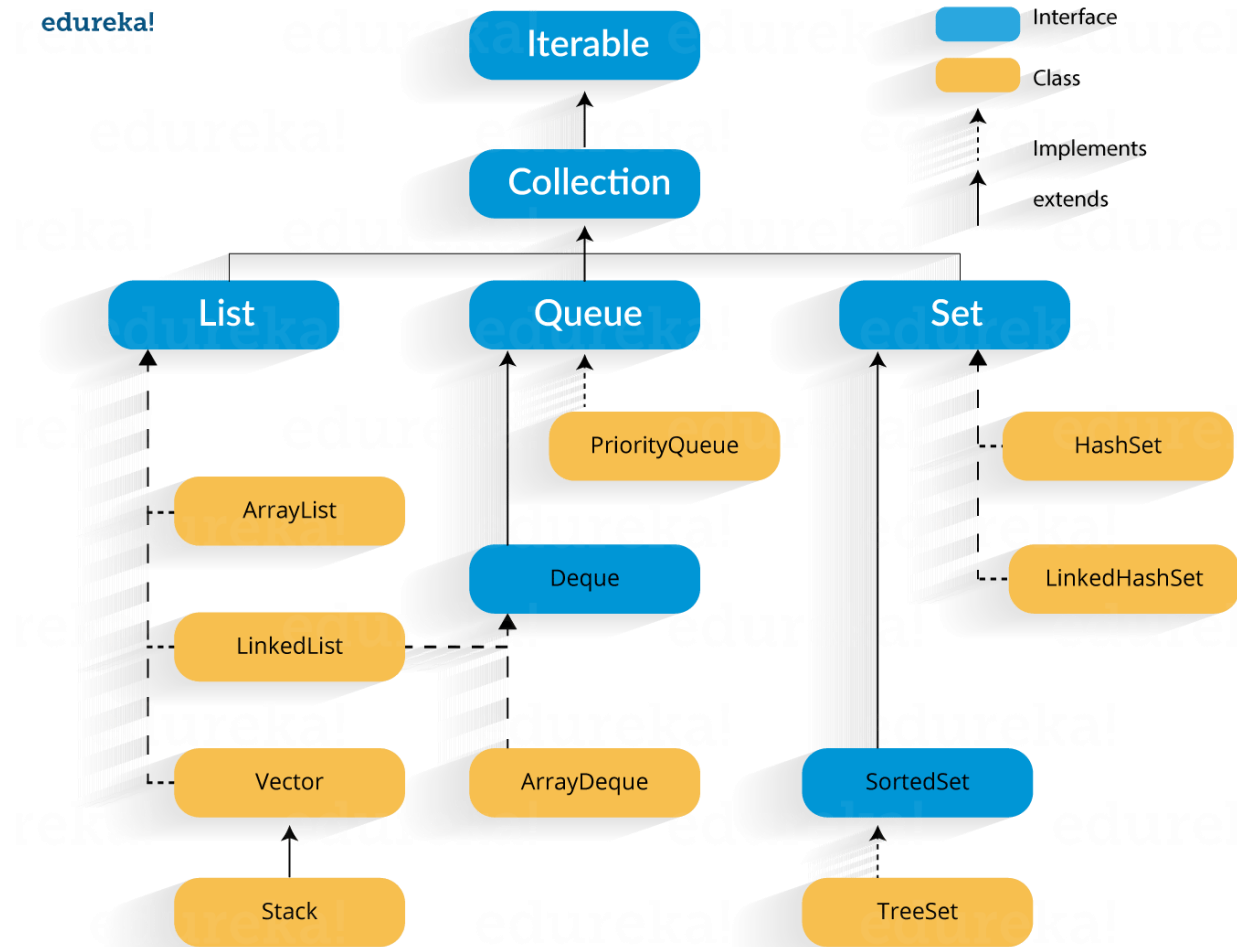
**Extensions Class Loader** – It loads classes from the JDK extensions directory, usually `$JAVA_HOME/lib/ext` directory.

**System Class Loader** – It loads classes from the current classpath that can be set while invoking a program using `-cp` or `-classpath` command line options.

## COLLECTION

## 162) Draw collection framework?

edureka!





The **Iterable** interface represents any collection that can be iterated using the *for-each* loop. The **Collection** interface inherits from *Iterable* and adds generic methods for checking if an element is in a collection, adding and removing elements from the collection, determining its size etc.

The **List**, **Set**, and **Queue** interfaces inherit from the *Collection* interface.

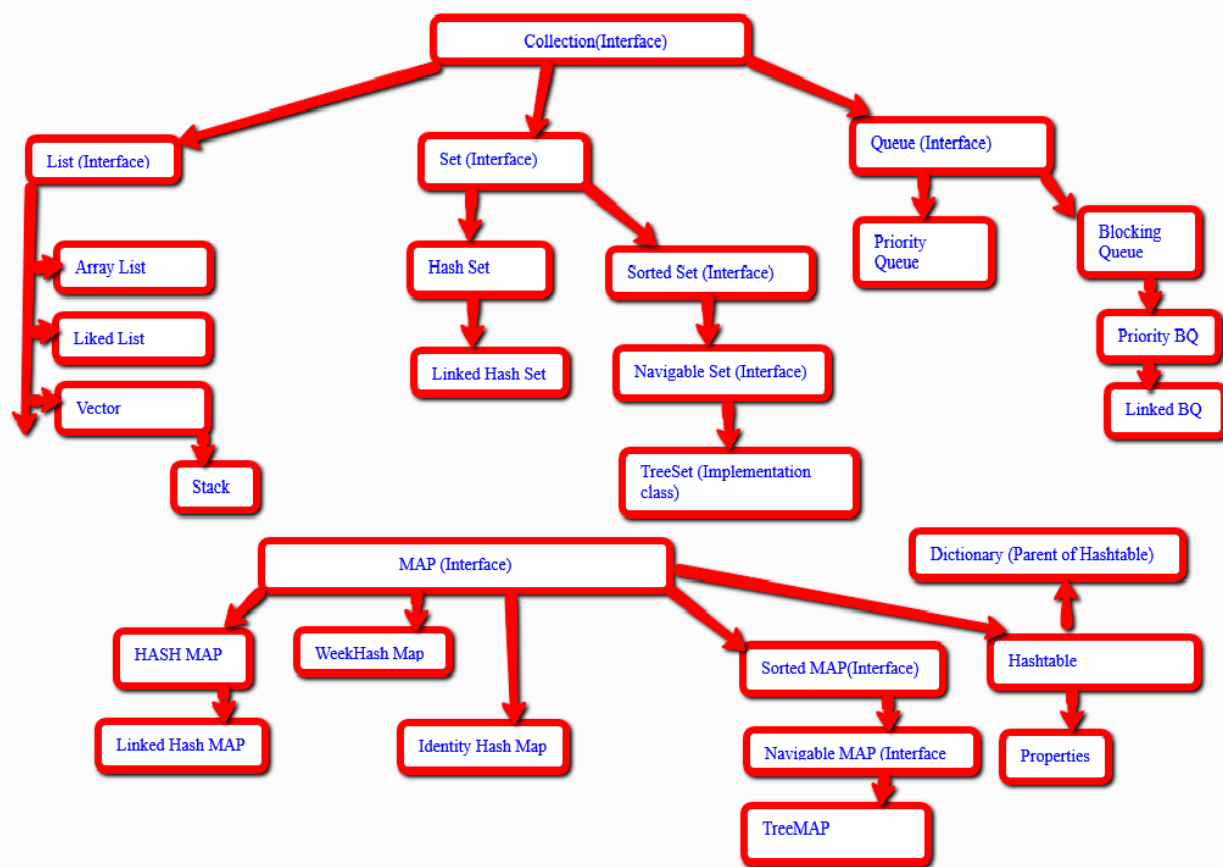
**List** is an ordered collection, and its elements can be accessed by their index in the list.

**Set** is an unordered collection with distinct elements, similar to the mathematical notion of a set.

**Queue** is a collection with additional methods for adding, removing and examining elements, useful for holding elements prior to processing.

**Map** interface is also a part of the collection framework, yet it does not extend *Collection*. This is by design, to stress the difference between collections and mappings which are hard to gather under a common abstraction. The *Map* interface represents a key-value data structure with unique keys and no more than one value for each key.

## Collection Framework



## 163) Draw map framework?

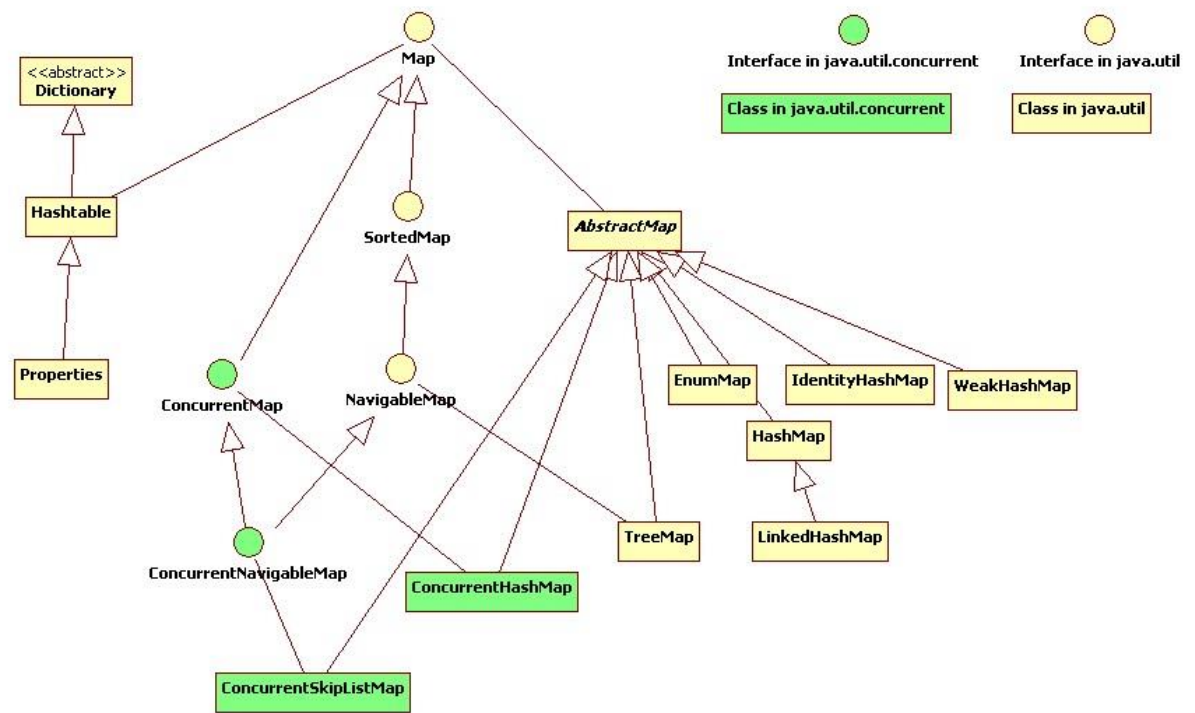
One of the most often used implementations of the *Map* interface is the **HashMap**. It is a typical hash map data structure that allows accessing elements in constant time, or  $O(1)$ , but **does not preserve order and is not thread-safe**.

To preserve insertion order of elements, you can use the **LinkedHashMap** class which extends the *HashMap* and additionally ties the elements into a linked list, with foreseeable overhead.

The **TreeMap** class stores its elements in a red-black tree structure, which allows accessing elements in logarithmic time, or  $O(\log(n))$ . It is slower than the *HashMap* for most cases, but it allows keeping the elements in order according to some *Comparator*.

The **ConcurrentHashMap** is a thread-safe implementation of a hash map. It provides full concurrency of retrievals (as the *get* operation does not entail locking) and high expected concurrency of updates.

The **Hashtable** class has been in Java since version 1.0. It is not deprecated but is mostly considered obsolete. It is a thread-safe hash map, but unlike *ConcurrentHashMap*, all its methods are simply *synchronized*, which means that all operations on this map block, even retrieval of independent values.



## 164) Describe Array

- There are three things that distinguish arrays from other containers:
  - 1. efficiency
  - 2. type
  - 3. ability to hold primitives
- This is the second place where an array is superior to the generic containers:
  - when you create an array, you create it to hold a specific type.
  - This means that you get compile-time type checking.
  - arrays of objects hold references, while arrays of primitives hold the primitive values directly.
  - Container classes can hold only references to objects.
  - An array, however, can be created to hold primitives directly, as well as references to objects
  - when you create an array object, its size is fixed and cannot be changed length tells you only how many elements can be placed in the array; that is, the size of the array object, not the number of elements it actually holds.
  - `System.arraycopy()` - copy array
  - `Arrays.sort()` - sort array
  - In Java, the `Arrays.sort()` methods use merge sort or a tuned quicksort depending on the datatypes
  - and for implementation efficiency switch to insertion sort when fewer than seven array elements are being sorted.
  - Array is an object in java so is allocated memory in heap
  - Inplace QuickSort
  - <http://javarevisited.blogspot.in/2014/08/quicksort-sorting-algorithm-in-java-in-place-example.html>

### 165) Can you store String in an array of Integer in Java? compile time error or runtime exception?

- yes and no
- if you do something like this you will get `ArrayStoreException`
- `Object[] names = new String[];`
- `names[0] = new Integer[0];`

### 166) Why arraylist search is constant?

This is because array store data in contiguous location, so when jvm has to get a particular index it get arr[0] location and add the required search index to get the data.

### 167) Explain arraylist?

the references of the objects are stored in contiguous location on the heap. Objects can be stored anywhere when arraylist reaches 75% maximum JVM allocates another array and copies the contents.

if JVM is not able to give memory or there is memory constraint then it will give out of Memory error

ArrayList.get() is O(1) not because of how the array is stored in memory, but because the cost of the element access is constant and not proportional to the number of elements in the ArrayList.

implements Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

ArrayList Resizing

```
int newCapacity = (oldCapacity * 3)/2 + 1;
```

```
elementData = Arrays.copyOf(elementData, newCapacity);
```

ArrayList.trimToSize() -- to minimize storage instance of arraylist

```
public void trimToSize() {  
    modCount++;  
    int oldCapacity = elementData.length;  
    if (size < oldCapacity) {  
        elementData = Arrays.copyOf(elementData, size);  
    }  
}
```

----Resizing

```
elementData = Arrays.copyOf(elementData,newlength);
```

### 168) why is elementData transient in arraylist ?

It can be serialized; the ArrayList class just takes care of things itself, rather than using the default mechanism.

Look at the writeObject() and readObject() methods in that class, which are part of the standard serialization mechanism.

If you look at the source, you see that rather than saving the whole array, possibly with some or many null elements, just the actual objects are saved.

This is always more efficient, and allows the ArrayList to be reconstituted with elementData at an optimal size

(i.e., without unnecessary excessive empty space in the array.)

This avoids serializing the private array object which 1) has its own header and overheads, and 2) is typically padded with nulls. The space saving can be significant.

### 169) how to Remove Duplicates from ArrayList in Java ?

Copy ArrayList to LinkedHashSet

```
Set<T> set = new LinkedHashSet<T>(ArrayList<T> list);
```

```
list.clear();
```

```
list.addAll(set);
```

### 170) How to Synchronize ArrayList in Java ?

```
List list = Collections.synchronizedList(new ArrayList());
```

```
...
```

```
synchronized(list) {
```

```
    Iterator i = list.iterator(); // Must be in synchronized block
```

```
    while (i.hasNext())
```

```
        foo(i.next());
```

```
}
```

### 171) How to convert ArrayList to String ?

```
String str = StringUtils.collectionToCommaDelimitedString(language);
```

OR

```
String str = StringUtils.collectionToDelimitedString(language, "|");
```

java.lang does not contain a class called StringUtils.

Several third-party libs do, such as Apache Commons Lang or the Spring framework

### 172) How to get subList ?

```
List list = arrayList.subList(1,3);
```

### 173) What is copyOnWrite ArayList ?

CopyOnWriteArrayList is very expensive because it involves costly Array copy with every write operation but its very efficient if you have a List where Iteration outnumber mutation e.g. you mostly need to iterate the ArrayList and don't modify it too often.

Iterator of CopyOnWriteArrayList is fail-safe and doesn't throw ConcurrentModificationException even if underlying CopyOnWriteArrayList is modified once Iteration begins because Iterator is operating on separate copy of ArrayList.

Iterator doesn't support remove method.

- As the name indicates, CopyOnWriteArrayList creates a Cloned copy of underlying ArrayList, for every update operation at certain point both will be synchronized automatically ,which is taken care by JVM. Therefore 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 if our frequent operation is read operation.
- It extends object and implements Serializable, Cloneable, Iterable<E>, Collection<E>, List<E> and RandomAccess
- The underlined data structure is grow-able array.
- It is thread-safe version of ArrayList.
- Insertion is preserved, duplicates are allowed and heterogeneous Objects are allowed.
- The main important point about CopyOnWriteArrayList is Iterator of CopyOnWriteArrayList can not perform remove operation otherwise we get Run-time exception saying UnsupportedOperationException.

### 174) How to remove elements from ArrayList ?

`remove(int index)`

`remove(Object o)`

`itr.remove()`

`al.remove(2)` – will remove 2 index ; where `al = 1,2,3`; to remove 2 element say `remove(new Integer(2))`

### 175) How to make ArrayList read only ?

`Collection readOnlyCollection = Collections.unmodifiableCollection(new ArrayList<String>());`

### 176) How to sort ArrayList in descending order in Java? (Answer)

By default, elements are sorted on increasing order as this is how their `compareTo()` or `compare()` method compares them.

If you want to sort into descending order, just reverse the comparison logic using `Collections.reverseComparator()` method.

`Comparator cmp = Collections.reverseOrder();`

`// sort the list`  
`Collections.sort(list, cmp);`

### 177) Difference between arraylist and linkedList?

LinkedList might allocate fewer entries, but those entries are astronomically more expensive than they'd be for ArrayList -- enough that even the worst-case ArrayList is cheaper as far as memory is concerned.

LinkedList consumes 24 bytes per element, while ArrayList consumes in the best case 4 bytes per element, and in the worst case 6 bytes per element. (Results may vary depending on 32-bit versus 64-bit JVMs, and compressed object pointer options, but in those comparisons LinkedList costs at least 36 bytes/element, and ArrayList is at best 8 and at worst 12.)

Once you have got past the initial capacity of the array list, the size of the backing will be between 1 and 2 references times the number of entries. This is due to strategy used to grow the backing array.



For a linked list, each node occupies AT LEAST 3 times the number of entries, because each node has a next and prev reference as well as the entry reference.

(And in fact, it is more than 3 times, because of the space used by the nodes' object headers and padding. Depending on the JVM and pointer size, it can be as much as 6 times.)

Even in the worst case, ArrayList is 4x smaller than a LinkedList with the same elements.

The only possible way to make LinkedList win is to deliberately fix the comparison by calling ensureCapacity with a deliberately inflated value, or to remove lots of values from the ArrayList after they've been added.

### 178) How arraylist works internally?

### 179) Can you quickly brief about Map, HashMap, HashTable, and TreeMap?

Answer: Map is an interface. HashMap is a class that implements a Map. It is unsynchronized and supports null values and keys

Hashtable is a synchronized version of HashMap.

TreeMap is similar to HashMap but uses Tree to implement Map.

### 180) Do you think not overriding hashCode() method has any performance implication?

Answer: A weak hashCode function will result into frequent collision in HashMap, which will at the end increase the time to add an object within Hash Map.

### 181) How to do custom sorting?

We need to implement Comparable interface to support sorting of custom objects in a collection. Comparable interface has compareTo(T obj) method which is used by sorting methods and by providing this method implementation, we can provide default way to sort custom objects collection.

However, if you want to sort based on different criteria, such as sorting an Employees collection based on salary or age, then we can create Comparator instances and pass it as sorting methodology

## 182) ArrayList definition as per java code?

- \* Resizable-array implementation of the `List` interface. Implements
- \* all optional list operations, and permits all elements, including
- \* `null`. In addition to implementing the `List` interface,
- \* this class provides methods to manipulate the size of the array that is
- \* used internally to store the list. (This class is roughly equivalent to
- \* `Vector`, except that it is unsynchronized.)
- \*
- \* **The `size`, `isEmpty`, `get`, `set`,**
- `iterator`, and `listIterator` operations run in**
- constant**
- time. The `add` operation runs in amortized constant**
- time,**
- that is, adding  $n$  elements requires  $O(n)$  time. All of the other**
- operations**
- run in linear time (roughly speaking). The constant factor is low**
- compared**
- to that for the `LinkedList` implementation.**
- \*
- \* Each `ArrayList` instance has a *capacity*. The
- capacity is
- \* the size of the array used to store the elements in the list. It is always
- \* at least as large as the list size. As elements are added to an `ArrayList`,
- \* its capacity grows automatically. The details of the growth policy are not
- \* specified beyond the fact that adding an element has constant amortized
- \* time cost.
- \*
- \* An application can increase the capacity of an `ArrayList` instance
- instance
- \* before adding a large number of elements using the
- `ensureCapacity`
- \* operation. This may reduce the amount of incremental reallocation.
- \*
- \* **Note that this implementation is not synchronized.**
- \* If multiple threads access an `ArrayList` instance concurrently,
- \* and at least one of the threads modifies the list structurally, it
- \* *must* be synchronized externally. (A structural modification is
- \* any operation that adds or deletes one or more elements, or explicitly
- \* resizes the backing array; merely setting the value of an element is not
- \* a structural modification.) This is typically accomplished by
- \* synchronizing on some object that naturally encapsulates the list.
- \*

\* If no such object exists, the list should be "wrapped" using the

\* [{@link Collections#synchronizedList Collections.synchronizedList}](#)

\* method. This is best done at creation time, to prevent accidental

\* unsynchronized access to the list: 

```
List list = Collections.synchronizedList(new ArrayList(...));
```

\*

\* 

<a name="fail-fast">

\* The iterators returned by this class's [{@link #iterator\(\) iterator}](#) and

\* [{@link #listIterator\(int\) listIterator}](#) methods are *fail-fast*:</a>

\* if the list is structurally modified at any time after the iterator is

\* created, in any way except through the iterator's own

\* [{@link ListIterator#remove\(\) remove}](#) or

\* [{@link ListIterator#add\(Object\) add}](#) methods, the iterator will throw a

\* [{@link ConcurrentModificationException}](#). Thus, in the face of

\* concurrent modification, the iterator fails quickly and cleanly, rather

\* than risking arbitrary, non-deterministic behavior at an undetermined

\* time in the future.

\*

\* 

Note that the fail-fast behavior of an iterator cannot be guaranteed

\* as it is, generally speaking, impossible to make any hard guarantees in the

\* presence of unsynchronized concurrent modification. Fail-fast iterators

\* throw `ConcurrentModificationException` on a best-effort basis.

\* Therefore, it would be wrong to write a program that depended on this

\* exception for its correctness: *the fail-fast behavior of iterators*

\* should be used only to detect bugs.</i>

### 183) What is the difference between Array list and vector?

Array List	Vector
Array List is not synchronized.	Vector is synchronized.
Array List is fast as it's non-synchronized.	Vector is slow as it is thread safe.
If an element is inserted into the Array List, it increases its Array size by 50%.	Vector defaults to doubling size of its array.
Array List does not define the increment size.	Vector defines the increment size.
Array List can only use Iterator for traversing an Array List.	Except Hashtable, Vector is the only other class which uses both Enumeration and Iterator.

## 184) LinkedList specification as per Java code?

Doubly-linked list implementation of the List and Deque interfaces. Implements all optional list operations, and permits all elements (including null). All of the operations perform as could be expected for a doubly-linked list. \*Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.

Note that this implementation is not synchronized. If multiple threads access a linked list concurrently, and at least one of the threads modifies the list structurally, it must be synchronized externally.

(A structural modification is any operation that adds or deletes one or more elements; merely setting the value of an element is not a structural modification.)

This is typically accomplished by synchronizing on some object that naturally encapsulates the list.

If no such object exists, the list should be "wrapped" using the Collections.synchronizedList method.

This is best done at creation time, to prevent accidental unsynchronized access to the list:

```
List list = Collections.synchronizedList(new LinkedList(...));
```

The iterators returned by this class's iterator and listIterator methods are fail-fast:

if the list is structurally modified at any time after the iterator is created, in any way except through the Iterator's own remove or add methods, the iterator will throw a ConcurrentModificationException.

Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future. Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis.

## 185) Difference between arraylist and linkedList

The first point here to remember is Array list is useful for Faster accessing/retrieval of elements and Linked list is useful in cases where we have to insert or delete elements.

The reason behind this is Array list implements RandomAccess interface which gives the capability of array list to achieve the random accessing of the elements.

LinkedList might allocate fewer entries, but those entries are astronomically more expensive than they'd be for ArrayList -- enough that even the worst-case ArrayList is cheaper as far as memory is concerned.

LinkedList consumes 24 bytes per element, while ArrayList consumes in the best case 4 bytes per element, and in the worst case 6 bytes per element. (Results may vary depending on 32-bit versus 64-bit JVMs, and compressed object pointer options, but in those comparisons LinkedList costs at least 36 bytes/element, and ArrayList is at best 8 and at worst 12.)

Once you have got past the initial capacity of the array list, the size of the backing will be between 1 and 2 references times the number of entries. This is due to strategy used to grow the backing array.

For a linked list, each node occupies AT LEAST 3 times the number of entries, because each node has a next and prev reference as well as the entry reference.

(And in fact, it is more than 3 times, because of the space used by the nodes' object headers and padding. Depending on the JVM and pointer size, it can be as much as 6 times.)

Even in the worst case, ArrayList is 4x smaller than a LinkedList with the same elements.

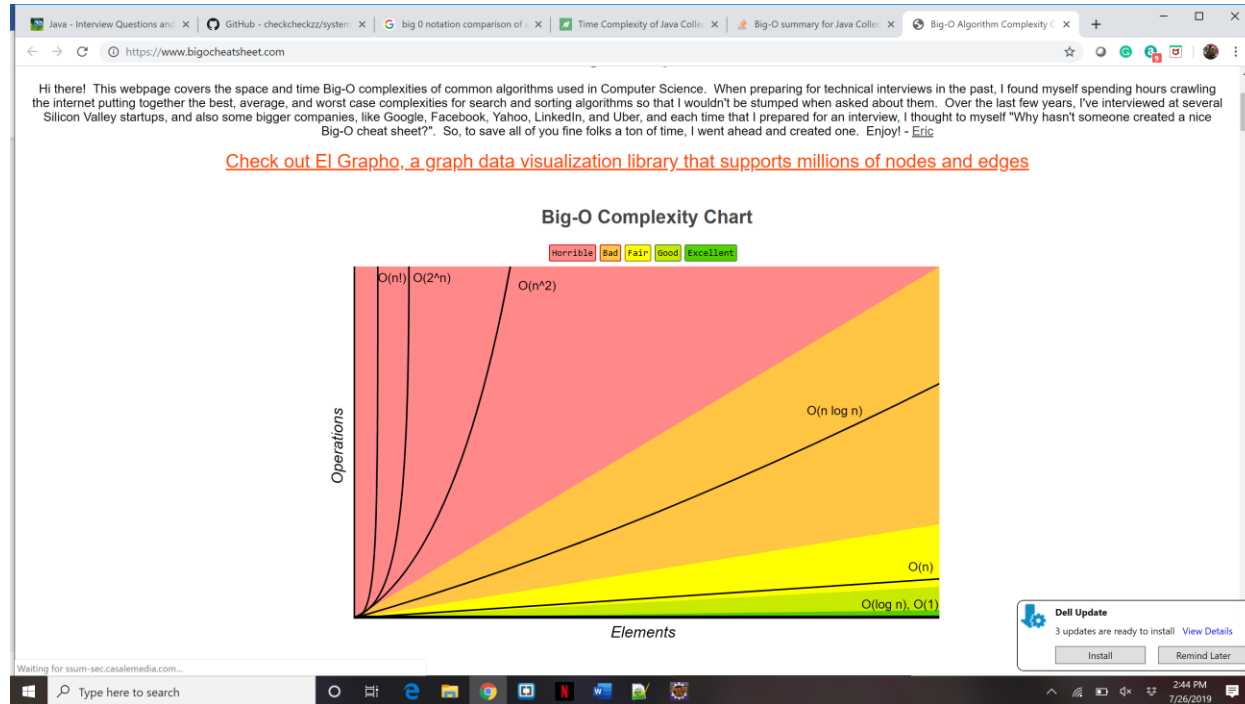
The only possible way to make LinkedList win is to deliberately fix the comparison by calling ensureCapacity with a deliberately inflated value, or to remove lots of values from the ArrayList after they've been added.

## 186) Compare Time complexity of all collection

<https://www.baeldung.com/java-collections-complexity>

<https://www.bigocheatsheet.com/>

# JAVA INTERVIEW QUESTIONS GENERIC, INNERCLAS,ENUMSET,EMUMAP,QUEUE, DEQUEUE



Ad closed by Google  
[Stop seeing this ad](#) [Why this ad?](#)

### Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity Worst
	Average				Worst				
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Stack	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Queue	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Singly-Linked List	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Doubly-Linked List	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Skip List	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n \log(n))$
Hash Table	N/A	$O(1)$	$O(1)$	$O(1)$	N/A	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Binary Search Tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Cartesian Tree	N/A	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	N/A	$O(n)$	$O(n)$	$O(n)$	$O(n)$
B-Tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
Red-Black Tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
Splay Tree	N/A	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	N/A	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
AVL Tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$
KD Tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

### Array Sorting Algorithms

Collection	Search	Add	Remove	comments
arrayList	O(1)	O(n)	O(n)	indexOf, contains()

JAVA INTERVIEW QUESTIONS GENERIC,  
INNERCLAS,ENUMSET,EMUMAP,QUEUE, DEQUEE

				runs $O(n)$ , get is $o(1)$
CopyOnWriteArrayList	$O(1)$	$O(n)$	$O(n)$ Contains()	
LinkedList	$O(n)$ Contains()	$O(1)$	$O(1)$	
HashMap	$O(1)$	$O(1)$		
TreeMap	$O(\log n)$	$O(\log n)$	$O(\log n)$	
Hashset	$O(1)$	$O(1)$	$O(1)$	
LinkedHashSet	$O(1)$	$O(1)$	$O(1)$	
Enumsete	$O(1)$	$O(1)$	$O(1)$	
Treeset	$O(\log n)$	$O(\log n)$	$O(\log n)$	

## 187) Difference between iterator and list iterator?

### 1. Traversal Direction :

ListIterator allows the programmers to iterate the list objects in both directions

Iterator can be used to iterate the list, map and set object in one direction i.e forward.

### 2. Set and Map implemented Objects Traversal :

ListIterator can be used to traverse List object only .

But Iterator can be used to traverse Map, List and Set implemented objects.

### 3. Add or Set operation at any index :

ListIterator can modify the list during iteration using add(E e) , remove() or set(E e).

Iterator can not add the element during traversal but they can remove the element from the underlying collection during the iteration

as they only consist of remove() method. There is no add(E e) and set(E e) method in Iterator.

### 4. Determine Iterator's current position :

ListIterator can obtain the iterator's current position in the list.

Iterator's current position during traversal can not be determined using Iterator.

### 5. Retrieve Index of the element :

ListIterator can obtain the index of the elements using previousIndex(E e) or nextIndex(E e) methods.

We can not obtain the index using Iterator as there is no such methods present.

## 188) Why iterator does not have add method ?

The Interface Iterator provides three methods that are hasNext(), next() and remove().

This means that we can remove any element from a collection while iterating over it. so why we cannot add element to the collection. Why they don't provide add() method.

Answer is, that they can't allow you to concurrently read and add element to a collection. The Iterator works on any collection whether it Set, List or Map.



So it does know very much the on which underlying it's going to work and we know all the collection implementation maintain ordering of elements based on some algorithm. For example TreeSet maintains the order of elements in by implementation Red-Black Tree data structure. Therefore if we tried to add an element to TreeSet using the iterator at given index or position of iterator. it might corrupt the state of the underlying data structure. So, the add() method could change structural state of a data structure.

While remove() method doesn't change or violate any state and rule of data structure or of the algorithm.

ListIterator provide the add() methods because it know the location where it needs to add the newly created element as List preserves the order of its element in order of their insertion.

=====

The sole purpose of an Iterator is to enumerate through a collection. All collections contain the add() method to serve your purpose. There would be no point in adding to an Iterator because the collection may or may not be ordered (in the case of a HashSet). Looking under the hood of ArrayList (line 111), and HashMap (line 149), we see that the implementation is just a few methods surrounding an array of objects.

Now we consider how arrays are treated in memory.

zero-based array indexes

This is an array of 5 elements. However, there are six indices. The letter "a" in this array is listed as element 0 because in order to read it,

left to right like a computer does, you have to start at index 0. Now, if we are iterating through this array (yes, collection, but it boils down to an array),

we will start at index 0 and continue to index 1. At this point in the Iterator, we want to call add("f");.

At this point, let's compare the implications of add() and remove(). remove() would leave a space in the array, which is easy to jump over,

because we can immediately recognize that it isn't a member. On the other hand, add() would put a new element in which wasn't there before.

This will affect the length of the array that we're iterating through. What happens when we get to that last element?

Can we even guarantee that it is there (that is, that the array hasn't exceeded the maximum size)?

=====

## 189) Comparable vs comparator?

Comparable meant for default natural sorting order. For customized use comparator Comparable present in java.lang. Comparator present in java.util Comparable only contains compareTo.....Comparator contains compare and equals All Wrapper classes under string class implements Comparable.

Comparator implemented by Collator and Rule- based Collator

Class implementing comparable must override compareTo()

method

```
import java.util.*;
```

```
class NameSorter implements Comparator<Employee>{
```

```
    @Override
```

```
    public int compare(Employee o1, Employee o2) {
```

```
        return
```

```
        o1.getEmpName().compareToIgnoreCase(o2.getEmpName());
```

```
    }
```

```
}
```

```
class deptSorter implements Comparator<Employee>
```

```
{
```

```
    @Override
```

```
    public int compare(Employee o1, Employee o2) {
```

```
        return o1.getDept().compareTo(o2.getDept());
```

```
    }
```

```
}
```

```
class Employee implements Comparable<Employee>{
```

```
//class Employee{
```

```
    private int empId;
```

```
    private String empName;
```

```
    private String dept;
```

```
    private String profession;
```

```
    public Employee() {
```

```
        super();
```

```
    }

    public Employee(int empId, String empName, String dept, String
profession) {
        super();
        this.empId = empId;
        this.empName = empName;
        this.dept = dept;
        this.profession = profession;
    }

    public int getEmpId() {
        return empId;
    }

    public String getEmpName() {
        return empName;
    }

    public String getDept() {
        return dept;
    }

    public String getProfession() {
        return profession;
    }

    public void setProfession(String profession) {
        this.profession = profession;
    }

    public String toString() {
        return "Employee [empId=" + empId + ", empName=" +
empName + ", dept="
                                + dept + ", profession=" + profession + "];"
    }

    public int compareTo(Employee o) {

        Integer it1=this.getEmpId();
        Integer it2=o.getEmpId();
```

```
        return it1.compareTo(it2);  
    }  
}
```

```

public class testComparable {

    public static void main(String[] args){

        List<Employee>list2 = new ArrayList<Employee>();

        list2.add(new Employee(111,"a","ba","c"));
        list2.add(new Employee(11,"d","bs","c"));
        list2.add(new Employee(1,"x","bd","c"));
        list2.add(new Employee(1111,"f","bf","c"));

        Iterator<Employee>it=list2.iterator();

        Collections.sort(list2);

        while(it.hasNext()){
            System.out.println(it.next());
        }

        /*Collections.sort(list2, new NameSorter());

        System.out.println("=====");

        it=list2.iterator();
        while(it.hasNext()){
            System.out.println(it.next());
        }

        Collections.sort(list2, new deptSorter());
        System.out.println("=====");
        System.out.println("=====");

        it=list2.iterator();
        while(it.hasNext()){
            System.out.println(it.next());
        }*/
    }
}

```

The *Comparable* interface is an interface for objects that can be compared according to some order. Its single method is *compareTo*, which operates on two values: the object itself and the argument object of the same type. For instance, *Integer*, *Long*, and other numeric types implement this interface. *String* also implements this interface, and its *compareTo* method compares strings in lexicographical order.

The *Comparable* interface allows to sort lists of corresponding objects with the *Collections.sort()* method and uphold the iteration order in collections that implement *SortedSet* and *SortedMap*. If your objects can be sorted using some logic, they should implement the *Comparable* interface.

The *Comparable* interface usually is implemented using natural ordering of the elements. For instance, all *Integer* numbers are ordered from lesser to greater values. But sometimes you may want to implement another kind of ordering, for instance, to sort the numbers in descending order.

The *Comparator* interface can help here.

The class of the objects you want to sort does not need to implement this interface. You simply create an implementing class and define the *compare* method which receives two objects and decides how to order them. You may then use the instance of this class to override the natural ordering of the *Collections.sort()* method or *SortedSet* and *SortedMap* instances.

As the *Comparator* interface is a functional interface, you may replace it with a lambda expression, as in the following example. It shows ordering a list using a natural ordering (*Integer*'s *Comparable* interface) and using a custom iterator (*Comparator<Integer>* interface).

```

1 List<Integer> list1 = Arrays.asList(5, 2, 3, 4, 1);
2 Collections.sort(list1);
3 assertEquals(new Integer(1), list1.get(0));
4
5 List<Integer> list1 = Arrays.asList(5, 2, 3, 4, 1);
6 Collections.sort(list1, (a, b) -> b - a);
7 assertEquals(new Integer(5), list1.get(0));

```

## 190) Ways to Iterate a Map?

There are generally **five** ways of iterating over a [Map](#) in Java. In this article, we will discuss all of them and also look at their advantages and disadvantages.

First of all, we **cannot** iterate a Map directly using [iterators](#), because Map are not [Collection](#). Also before going further, you must know a little-bit about [Map.Entry<K, V>](#) interface.

Since all maps in Java implement [Map](#) interface, following techniques will work for any map implementation ([HashMap](#), [TreeMap](#), [LinkedHashMap](#), [Hashtable](#), etc.)

### **Iterating over Map.entrySet() using For-Each loop :**

*Map.entrySet()* method returns a collection-view(*Set<Map.Entry<K, V>>*) of the mappings contained in this map. So we can iterate over key-value pair using *getKey()* and *getValue()* methods of [Map.Entry<K, V>](#). This method is most common and should be used if you need both map keys and values in the loop. Below is the java program to demonstrate it.



```
// Java program to demonstrate iteration over
// Map.entrySet() entries using for-each loop

import java.util.Map;
import java.util.HashMap;

class IterationDemo
{
    public static void main(String[] arg)
    {
        Map<String,String> gfg = new
HashMap<String,String>();

        // enter name/url pair
        gfg.put("GFG", "geeksforgeeks.org");
        gfg.put("Practice",
"practice.geeksforgeeks.org");
        gfg.put("Code", "code.geeksforgeeks.org");
        gfg.put("Quiz", "quiz.geeksforgeeks.org");

        // using for-each loop for iteration over
Map.entrySet()
        for (Map.Entry<String,String> entry :
gfg.entrySet())
            System.out.println("Key = " + entry.getKey() +
                                ", Value = " +
entry.getValue());
    }
}
```

Output:

```
Key = Quiz, Value = quiz.geeksforgeeks.org
Key = Practice, Value = practice.geeksforgeeks.org
Key = GFG, Value = geeksforgeeks.org
Key = Code, Value = code.geeksforgeeks.org
```

#### **Iterating over keys or values using keySet() and values() methods**

*Map.keySet()* method returns a Set view of the keys contained in this map and *Map.values()* method returns a collection-view of the values contained in this map. So If you need only keys or values from the map, you can iterate over keySet or values using for-each loops. Below is the java program to demonstrate it.

```
// Java program to demonstrate iteration over
// Map using keySet() and values() methods

import java.util.Map;
import java.util.HashMap;

class IterationDemo
{
    public static void main(String[] arg)
    {
        Map<String,String> gfg = new
        HashMap<String,String>();

        // enter name/url pair
        gfg.put("GFG", "geeksforgeeks.org");
        gfg.put("Practice",
        "practice.geeksforgeeks.org");
        gfg.put("Code", "code.geeksforgeeks.org");
        gfg.put("Quiz", "quiz.geeksforgeeks.org");

        // using keySet() for iteration over keys
        for (String name : gfg.keySet())
            System.out.println("key: " + name);

        // using values() for iteration over keys
        for (String url : gfg.values())
            System.out.println("value: " + url);
    }
}
```

Output:

key: Quiz

key: Practice

key: GFG

key: Code

value: quiz.geeksforgeeks.org

value: practice.geeksforgeeks.org

value: geeksforgeeks.org

value: code.geeksforgeeks.org

**Iterating using iterators over [Map.Entry<K, V>](#)**

This method is somewhat similar to first one. In first method we use for-each loop over Map.Entry<K, V>, but here we use [iterators](#). Using iterators over Map.Entry<K, V> has it's own advantage,i.e. we can remove entries from the map during iteration by calling *iterator.remove()* method.

```
// Java program to demonstrate iteration over
// Map using keySet() and values() methods

import java.util.Map;
import java.util.HashMap;
import java.util.Iterator;

class IterationDemo
{
    public static void main(String[] arg)
    {
        Map<String,String> gfg = new HashMap<String,String>();

        // enter name/url pair
        gfg.put("GFG", "geeksforgeeks.org");
        gfg.put("Practice", "practice.geeksforgeeks.org");
        gfg.put("Code", "code.geeksforgeeks.org");
        gfg.put("Quiz", "quiz.geeksforgeeks.org");

        // using iterators
        Iterator<Map.Entry<String, String>> itr =
gfg.entrySet().iterator();

        while(itr.hasNext())
        {
            Map.Entry<String, String> entry = itr.next();
            System.out.println("Key = " + entry.getKey() +
                                ", Value = " + entry.getValue());
        }
    }
}
```

Output :

Key = Quiz, Value = quiz.geeksforgeeks.org

Key = Practice, Value = practice.geeksforgeeks.org

Key = GFG, Value = geeksforgeeks.org

Key = Code, Value = code.geeksforgeeks.org

**Using forEach(action) method :**

In Java 8, you can iterate a map using *Map.forEach(action)* method and using [lambda expression](#). This technique is clean and fast.

```
// Java code illustrating iteration
// over map using forEach(action) method

import java.util.Map;
import java.util.HashMap;

class IterationDemo
{
    public static void main(String[] arg)
    {
        Map<String,String> gfg = new
HashMap<String,String>();

        // enter name/url pair
        gfg.put("GFG", "geeksforgeeks.org");
        gfg.put("Practice",
"practice.geeksforgeeks.org");
        gfg.put("Code", "code.geeksforgeeks.org");
        gfg.put("Quiz", "quiz.geeksforgeeks.org");

        // forEach(action) method to iterate map
        gfg.forEach((k,v) -> System.out.println("Key =
"
                + k + ", Value = " + v));
    }
}
```

Output :

Key = Quiz, Value = quiz.geeksforgeeks.org

Key = Practice, Value = practice.geeksforgeeks.org

Key = GFG, Value = geeksforgeeks.org

Key = Code, Value = code.geeksforgeeks.org

#### **Iterating over keys and searching for values (inefficient)**

Here first we loop over keys(using *Map.keySet()* method) and then search for value(using *Map.get(key)* method) for each key.This method is not used in practice as it is pretty slow and inefficient as getting values by a key might be time-consuming.

## JAVA INTERVIEW QUESTIONS GENERIC, INNERCLAS,ENUMSET,EMUMAP,QUEUE, DEQUEUE

```
// Java program to demonstrate iteration
// over keys and searching for values

import java.util.Map;
import java.util.HashMap;

class IterationDemo
{
    public static void main(String[] arg)
    {
        Map<String,String> gfg = new HashMap<String,String>();

        // enter name/url pair
        gfg.put("GFG", "geeksforgeeks.org");
        gfg.put("Practice", "practice.geeksforgeeks.org");
        gfg.put("Code", "code.geeksforgeeks.org");
        gfg.put("Quiz", "quiz.geeksforgeeks.org");

        // looping over keys
        for (String name : gfg.keySet())
        {
            // search for value
            String url = gfg.get(name);
            System.out.println("Key = " + name + ", Value = " +
url);
        }
    }
}
```

Output :

Key = Quiz, Value = quiz.geeksforgeeks.org

Key = Practice, Value = practice.geeksforgeeks.org

Key = GFG, Value = geeksforgeeks.org

Key = Code, Value = code.geeksforgeeks.org

### 191) Difference between fail-safe and fail-fast?

fail-fast : As the name implies, fail-fast iterator fail as soon as they believe that structure of the collection has been changed since iteration has started. Change in structure means insertion, deletion and updation of any element from Collection, while one thread is iterating over that collection.

This(fail-fast) behavior is implemented by keeping a modification count and if iteration thread realizes the change in modification count it throws `ConcurrentModificationException`. However, this check is done without synchronization, so there is a risk of seeing a stale value of the modification count and therefore that the iterator does not realize a modification has been made. This was a deliberate design tradeoff to reduce the performance impact of the concurrent modification detection code.

fail-safe : If Collection is modified structurally while one thread is Iterating over it then fail-safe iterator doesn't throw any Exception because they work on clone of Collection instead of original collection and the reason they are known as fail-safe iterator.

Iterator of `CopyOnWriteArrayList`, `CopyOnWriteArraySet` and `ConcurrentHashMap` are fail-safe and never throw `ConcurrentModificationException` in Java.

how exception occurs in single-threaded environment ?

```
While(itr.hasNext()){  
    list.remove();  
    itr.next();  
}
```

### 192) Difference between iterator and enumeration?

## Iterator

Iterator is the interface and found in the java.util package.  
It has three methods

- \*hasNext()
- \*next()
- \*remove()

***Read Also:*** [Java interview questions for experienced](#)

## Enumeration

Enumeration is also an interface and found in the java.util package .  
It is used for passing through a collection, usually of unknown size.

It has following methods

- \*hasMoreElements()
- \*nextElement()

**Note :** Enumeration does not have remove() method.

## Difference between Iterator and Enumeration:

**1. Remove() method :** The major difference between Iterator and Enumeration is that Iterator has the remove() method while Enumeration does not have remove() method.

Enumeration interface acts as a read only interface, one can not do any modifications to Collection while traversing the elements of the Collection. Iterator can do modifications (e.g using remove() method it removes the element from the Collection during traversal).

Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.

Enumeration is used for read only access while Iterator is useful to manipulate the list.

2. **Addition to JDK :** Enumeration is added to the jdk1.0 version while Iterator is added in jdk1.2 version.

**3. Fail-fast or Fail-safe :** Enumeration is fail-safe in nature. It does not throw ConcurrentModificationException if Collection is modified during the traversal.

Iterator is fail-fast in nature. It throws ConcurrentModificationException if a Collection is modified while iterating other than its own remove() method. I have already shared the [difference between fail-fast and fail-safe iterators in java](#).

**4. Legacy :** Enumeration is a legacy interface which is used for traversing Vector, Hashtable. Iterator is not a legacy interface. Iterator can be used for the traversal of HashMap, LinkedList, ArrayList, HashSet, TreeMap, TreeSet .

**5. Preference :** According to [Oracle docs](#),

The functionality of Enumeration is duplicated by the iterator interface. Iterator adds an optional remove operation, and has shorter method names. New implementations should consider using Iterator in preference to Enumeration.

3.

As *Iterator* is fail-fast in nature and doesn't allow modification of a collection by other threads while iterating, it is considered as safe and secure than *Enumeration*.

### Similarities between Iterator and Enumeration in Java

**1. Interface :** Both Iterator and Enumeration are interfaces.

**2. Package :** Both Iterator and Enumeration are present in java.util package.

### Example of Iterator and Enumeration

```
import java.util.*;
public class Performance {
    public static void main(String[] args){
        Vector v=new Vector();
        Object element;
        Enumeration enum;
        Iterator iter;
        long start;

        for(int i=0; i<1000000; i++){
            v.add("New Element");
```



```

    }

    enum=v.elements();
    iter=v.iterator();
    //*****CODE BLOCK FOR ITERATOR*****
    start=System.currentTimeMillis();
    while(iter.hasNext()){
        element=iter.next();
    }
    System.out.println("Iterator took " + (System.currentTimeMillis()-start));
    //*****END OF ITERATOR BLOCK*****

    System.gc(); //request to GC to free up some memory
    //*****CODE BLOCK FOR ENUMERATION*****
    start=System.currentTimeMillis();
    while(enum.hasMoreElements()){
        element=enum.nextElement();
    }
    System.out.println("Enumeration took " + (System.currentTimeMillis()-start));
    //*****END OF ENUMERATION BLOCK*****
}
}

```

### Recap : Difference between Iterator and Enumeration in Java

	Iterator	Enumeration
Throw ConcurrentModification Exception	Yes	No
Remove() method	Yes, you can remove the element while traversing it	No
Addition to JDK	1.2	1.0
Legacy	No	Yes

### 193) Why we need iterator when we have for loop?

For loops are expensive to the processor when the collection reaches large sizes, as many operations are done just to compute the first line:

**For loop total operations :**

int i = 0 is an assignment and creation (2 operations)

i get size, check value of i, and compare (3 operations)

i++ gets i then adds 1 to it [++i is only 2 operations] this one (3 operations)

**\*7/8 operations in total, each time the loop runs through**

**Enumeration total operations :**

where an enumeration or iterator uses a while(){}

while(v.hasNext()) has next true or false (1 operation)

while(v.hasMoreElements()) has more true or false (1 operation)

**\*Only one operation per repeat of this loop**

That's why enumeration (or iterator) has been added to the jdk.

## 194) What are the sorting algorithms used in java collection?

The API guarantees a stable sorting which Quicksort doesn't offer.

However, when sorting primitive values by their natural order you won't notice a difference as primitive values have no identity.

Therefore, Quicksort is used for primitive arrays as it is slightly more efficient.

Arrays.sort – use Quicksort for primitive array

For objects you may notice, when objects which are deemed equal according to their equals implementation or the provided Comparator change their order.

Therefore, Quicksort is not an option. So a variant of MergeSort is used, the current Java versions use TimSort.

This applies to both, Arrays.sort and Collections.sort, though with Java 8, the List itself may override the sort algorithms.

----- OR

The Quicksort is used by Arrays.sort for sorting primitive collections because stability isn't required (you won't know or care if two identical ints were swapped in the sort)

MergeSort or more specifically Timsort is used by Arrays.sort for sorting collections of objects. Stability is required. Quicksort does not provide for stability, Timsort does.

----- OR

<sup>1</sup> The efficiency advantage of [Quicksort](#) is needing less memory when done in-place. But it has a dramatic worst case performance and can't exploit runs of pre-sorted data in an array, which [TimSort](#) does.

Therefore, the sorting algorithms were reworked from version to version, while staying in the now-misleadingly named class `DualPivotQuicksort`. Also, the documentation didn't catch up, which shows, that it is a bad idea in general, to name an internally used algorithm in a specification, when not necessary.

The current situation (including Java 8 to Java 11) is as follows:

Generally, the sorting methods for primitive arrays will use [Quicksort](#) only under certain circumstances. For larger arrays, they will try to identify runs of pre-sorted data first, like [TimSort](#) does, and will merge them when the number of runs does not exceed a certain threshold. Otherwise they will fall back to [Quicksort](#), but with an implementation that will fall back to [Insertion sort](#) for small ranges, which does not only affect small arrays, but also quick sort's recursion.

`sort(char[],...)` and `sort(short[],...)` add another special case, to use [Counting sort](#) for arrays whose length exceeds a certain threshold

Likewise, `sort(byte[ ],...)` will use [Counting sort](#), but with a much smaller threshold, which creates the biggest contrast to the documentation, as `sort(byte[ ],...)` never uses Quicksort. It only uses [Insertion sort](#) for small arrays and [Counting sort](#) otherwise.

## 195) What is HashMap?

HashMap maintains an array of buckets. Each bucket is a linkedlist of key value pairs encapsulated as Entry objects. This array of buckets is called table. Each node of the linked list is an instance of a private class called Entry  
transient Entry[] table;

An entry is a private static class inside HashMap which implements Map. Entry contains node for next element, which kind of form linked List.Entry

```
private static class Entry<K,V> implements Map.Entry<K,V> {
    final K key;
    final int hash;
    V value;
    Entry<K,V> next;
}
```

Each entry object exists only for a particular key but values may change (if same key is reinserted later with a different value) -

hence key is final while value is not. Each Entry object has a field called next which points to the next Entry thus behaving like a singly linked list.

The hash field stores the hashed value of the key

constructor:

HashMap provides overloaded constructors with parameters for initial capacity and load factor but typically no args constructor is the one most frequently used

default values of these fields are :

initial capacity :  $1 \ll 4$  (ie 16)

load factor : 0.75

Whenever the element count of the hashmap reaches the load factor fraction of capacity, the map is resized and capacity is doubled

If capacity provided by client is a power of 2, then real capacity will be same as capacity

else real capacity = nearest power of 2 > provided capacity

maximum capacity is  $1 \ll 30$  (ie  $2^{30}$ ) if capacity provided is greater than that, then real capacity =  $2^{30}$

Note that capacity indicates the size of the table array (the array of buckets) and not the number of key-value pairs the `HashMap` can support

## 196) What is the purpose of initial capacity and load factor?

The *initialCapacity* argument of the *HashMap* constructor affects the size of the internal data structure of the *HashMap*, but reasoning about the actual size of a map is a bit tricky. The *HashMap*'s internal data structure is an array with the power-of-two size. So the *initialCapacity* argument value is increased to the next power-of-two (for instance, if you set it to 10, the actual size of the internal array will be 16).

The load factor of a *HashMap* is the ratio of the element count divided by the bucket count (i.e. internal array size). For instance, if a 16-bucket *HashMap* contains 12 elements, its load factor is  $12/16 = 0.75$ . A high load factor means a lot of collisions, which in turn means that the map should be resized to the next power of two. So the *loadFactor* argument is a maximum value of the load factor of a map. When the map achieves this load factor, it resizes its internal array to the next power-of-two value.

The *initialCapacity* is 16 by default, and the *loadFactor* is 0.75 by default, so you could put 12 elements in a *HashMap* that was instantiated with the default constructor, and it would not resize. The same goes for the *HashSet*, which is backed by a *HashMap* instance internally.

Consequently, it is not trivial to come up with *initialCapacity* that satisfies your needs. This is why the Guava library has *Maps.newHashMapWithExpectedSize()* and *Sets.newHashSetWithExpectedSize()* methods that allow you to build a *HashMap* or a *HashSet* that can hold the expected number of elements without resizing.

### 197) Is Hashmap fail-safe?

HashMap specifications need that iterators should throw ConcurrentModificationException if the map contents are changed while a client is iterating over an iterator

This done by keeping track of number of modifications.

HashMap has a member int variable named modCount which is incremented everytime the map is altered (any invocation of put(), remove(), putAll() or clear() methods)

A similar field (lets say iteratorModCount) is maintained in the iterator implemetation class.

When the iterator is created, the iteratorModCount value is initialized with the same value as HashMap modCount.

For every call to any of iterator methods (next(), hasNext() and remove() ) the iteratorModCount is checked against the HashMap modCount.

If these two values dont tally, that means HashMap has been modified and ConcurrentModificationException is thrown

When the remove() method of iterator is invoked, after internally calling remove() on the map, the iteratorModCount is reinitialized with the HashMap's modCount

Note: the HashMap's modCount and iterator's modCount fields are neither atomic nor volatile -

hence they are vulnerable to interleaving and are not guraranteed to work in a multithreaded context

### 198) Will hashmap shrink if data is removed?

HashMap does not shrink when data is removed. Even if all keys are removed from HashMap, the inner size of it's table does not change

### 199) What is HashMap collison?

A collision occurs when a hash function returns same bucket location for two different keys. Since all hash based Map class e.g. HashMap uses equals() and hashCode() contract to find the bucket. HashMap calls the hashCode() method to compute the hash value which is used to find the bucket location as shown in below code snippet from the HashMap class of JDK 1.7 (jkd1.7.0\_60) update.

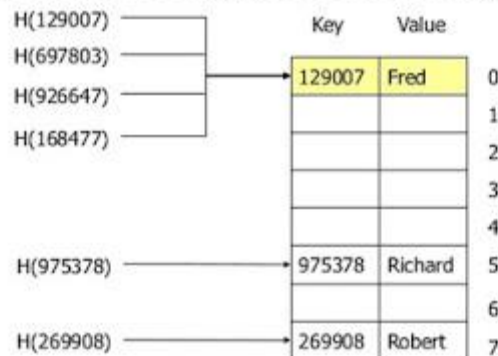
Ignoring the first two lines, which was the performance improvement done for String keys in JDK 7, you can see that computation of hash is totally based upon the hashCode method.

A collision will occur when two different keys have the same hashCode, which can happen because two unequal objects in Java can have the same hashCode.

### Collision handling strategies



- Closed addressing (open hashing).
- Open addressing (closed hashing).



Read more: <https://javarevisited.blogspot.com/2016/01/how-does-java-hashmap-or-linkedhahsmap-handles.html#ixzz5upPUnXPd>

## 200) How java handles hashmap collision?

1. Each bucket contains a linked list and whenever there is a collision the element get added at the end and can be retrieved correctly by use of equals method
2. Dynamic resizing : increase the number of buckets and rehash and redistribute all the elements

Java uses both methods

In order to address this issue in the case of frequent HashMap collisions, Java8 has started using a balanced tree instead of linked list for storing collided entries.

This also means that in the worst case you will get a performance boost from  $O(n)$  to  $O(\log n)$ .



## 201) How hashmap collision is handled in Java8?

Prior to Java 8, HashMap and all other hash table based Map implementation classes in Java handle collision by *chaining*, i.e. they use [linked list](#) to store map entries which ended in the same bucket due to a collision. If a key ends up in same bucket location where an entry is already stored then this entry is just added at the head of the linked list there. In the worst case this degrades the performance of the [get\(\) method of HashMap](#) to  $O(n)$  from  $O(1)$ . In order to address this issue in the case of frequent HashMap collisions, Java8 has started using a balanced tree instead of linked list for storing collided entries. This also means that in the worst case you will get a performance boost from  $O(n)$  to  $O(\log n)$ .

The threshold of switching to the balanced tree is defined as TREEIFY\_THRESHOLD constant in `java.util.HashMap` JDK 8 code. Currently, its value is 8, which means if there are more than 8 elements in the same bucket then HashMap will use a tree instead of linked list to hold them in the same bucket.

This change is in continuation of efforts to improve most used classes. If you remember earlier in JDK 7 they have also introduced a [change](#) so that empty ArrayList and HashMap will take less memory by postponing the allocation of the underlying array until an element is added.

This is a dynamic feature which means HashMap will initially use the linked list but when the number of entries crosses a certain threshold it will replace the linked list with a balanced binary tree. Also, this feature will not be available to all hash table based classes in Java e.g. **Hashtable will not have this feature** because of its legacy nature and given that this feature can change the traditional legacy [iteration order of Hashtable](#). Similarly, **WeakHashMap** will also not include this feature.

So far (until JDK 8) only ConcurrentHashMap, LinkedHashMap and HashMap will use the balanced tree in case of a frequent collision. This is a dynamic feature which means [HashMap will initially use the linked list](#) but when the number of entries crosses a certain threshold it will replace the linked list with a balanced binary tree.

Read more: <https://javarevisited.blogspot.com/2016/01/how-does-java-hashmap-or-linkedhashmap-handles.html#ixzz5upQgZqjN>

## 202) What is red black tree?

## 203) Difference between hashtable and hashmap?

1. Synchronization or Thread Safe :  
HashMap is non synchronized and not thread safe.  
HashTable is thread safe and synchronized.
2. Null keys and null values :  
Hashmap allows one null key and any number of null values,  
while \*\*\*\*Hashtable do not allow null keys and null values in the HashTable object.
3. Iterating the values:  
Hashmap object values are iterated by using iterator .  
HashTable is the only class other than vector which uses enumerator to iterate the values of HashTable object.
4. Fail-fast iterator :  
The iterator in Hashmap is fail-fast iterator while the enumerator for Hashtable is not.
5. Performance :  
Hashmap is much faster and uses less memory than Hashtable as former is unsynchronized .
6. Superclass and Legacy :  
Hashtable is a subclass of Dictionary class which is now obsolete in Jdk 1.7 ,so ,it is not used anymore.  
It is better off externally synchronizing a HashMap or using a ConcurrentMap implementation (e.g ConcurrentHashMap).  
HashMap is the subclass of the AbstractMap class.  
Although Hashtable and HashMap has different superclasses but they both are implementations of the "Map" abstract data type.

When to use HashMap ?

if your application do not require any multi-threading task, in other words hashmap is better for non-threading applications.  
HashTable should be used in multithreading applications.

## 204) Difference between hashset and treeset

Both **HashSet** and **TreeSet** classes implement the *Set* interface and represent sets of distinct elements. Additionally, *TreeSet* implements the *NavigableSet* interface. This interface defines methods that take advantage of the ordering of elements.

*HashSet* is internally based on a *HashMap*, and *TreeSet* is backed by a *TreeMap* instance, which defines their properties: *HashSet* does not keep elements in any particular order. Iteration over the elements in a *HashSet* produces them in a shuffled order. *TreeSet*, on the other hand, produces elements in order according to some predefined *Comparator*.

## 205) What is enumSet and enumMap?

**EnumSet** and **EnumMap** are special implementations of *Set* and *Map* interfaces correspondingly. You should always use these implementations when you're dealing with enums because they are very efficient.

An *EnumSet* is just a bit vector with "ones" in the positions corresponding to ordinal values of enums present in the set. To check if an enum value is in the set, the implementation simply has to check if the corresponding bit in the vector is a "one", which is a very easy operation. Similarly, an *EnumMap* is an array accessed with enum's ordinal value as an index. In the case of *EnumMap*, there is no need to calculate hash codes or resolve collisions.

## 206) Why Collection doesn't extend Cloneable and Serializable interfaces ?

The *Collection* interface specifies groups of objects known as elements. Each concrete implementation of a *Collection* can choose its own way of how to maintain and order its elements. Some collections allow duplicate keys, while some other collections don't. The semantics and the implications of either cloning or serialization come into play when dealing with actual implementations. Thus, the concrete implementations of collections should decide how they can be cloned or serialized.

## 207) What is the importance of hashCode() and equals() methods ?

A HashMap in Java uses the hashCode and equals methods to determine the index of the key-value pair. These methods are also used when we request the value of a specific key. If these methods are not implemented correctly, two different keys might produce the same hash value and thus, will be considered as equal by the collection. Furthermore, these methods are also used to detect duplicates. Thus, the implementation of both methods is crucial to the accuracy and correctness of the HashMap.

## 208) What are some of the best practices relating to the Java Collection framework ?

- Choosing the right type of the collection to use, based on the application's needs, is very crucial for its performance. For example if the size of the elements is fixed and known a priori, we shall use an Array, instead of an ArrayList.
- Some collection classes allow us to specify their initial capacity. Thus, if we have an estimation on the number of elements that will be stored, we can use it to avoid rehashing or resizing.
- Always use Generics for type-safety, readability, and robustness. Also, by using Generics you avoid the ClassCastException during runtime.
- Use immutable classes provided by the Java Development Kit (JDK) as a key in a Map, in order to avoid the implementation of the hashCode and equals methods for our custom class.
- Program in terms of interface not implementation.
- Return zero-length collections or arrays as opposed to returning a null in case the underlying collection is actually empty.

209) Internal working of linkedHashMap?

210) Internal working of ConcurrentHashMap?

211) What is navigable map?

212) Difference between hashmap and  
ConcurrentHashMap?

1.ConcurrentHashMap is thread-safe while HashMap is not thread-safe .

2.HashMap can be synchronized by using synchronizedMap(HashMap) method.By using this method we get a HashMap object which is equivalent to the Hashtable object .So every modification is performed on Map is locked on Map object.

```
Map<String,String> syncMap = Collections.synchronizedMap(map);
```

ConcurrentHashMap synchronizes or locks on the certain portion of the Map .  
To optimize the performance of ConcurrentHashMap ,

Map is divided into different partitions(buckets) depending upon the  
Concurrency level . So that we do not need to synchronize the whole Map  
Object.

3.\*\*\*ConcurrentHashMap does not allow NULL values . So the key can not  
be null in ConcurrentHashMap .

While In HashMap there can only be one null key .

---- if map.get(key) returns null, you can't detect whether the key explicitly  
maps to null vs the key isn't mapped.

In a non-concurrent map, you can check this via map.containsKey(), but in a  
concurrent one, the map might have changed between calls.

4. In multiple threaded environment HashMap is usually faster than ConcurrentHashMap .

As only single thread can access the certain portion of the Map and thus reducing the performance

while in HashMap any number of threads can access the code at the same time .

5. When we are performing the operation like adding & deleting objects at same time( I mean concurrently ),

then in the case of HashMap throws ConcurrentModification Exceptions.

But in ConcurrentHashMap, it'll not throws any Exceptions.

always use ConcurrentHashMap rather than synchronized hashmap, since even though the operations on the synchronized hashmap itself are thread-safe,

those on the iterator are not.

So if you are iterating on a synchronized hashmap and the map is changed, you will still get a concurrent modification exception.

## 213) Difference between hashmap and identityHashMap?

1. Equality used(Reference Equality instead of Object Equality) :  
IdentityHashMap uses reference equality to compare keys and values while  
HashMap uses object equality to compare keys and values .  
for example :

Suppose we have two keys k1 and k2

In HashMap :

two keys are considered equal if and only if (k1==null ? k2==null :  
k1.equals(k2))

// object equality i.e using equals() method to compare objects

In IdentityHashMap :

two keys are considered equal if and only if (k1 == k2)

//reference equality i.e using == operator to compare objects

2. Map's contract violation : IdentityHashMap implements the Map interface,  
it intentionally violates the Map general contract ,  
which mandates the use of equals method when comparing objects.

HashMap also implements Map interface but it does not violate the Map  
general contract as it uses equals method to compare objects .

3. Hashcode method : IdentityHashMap does not use hashCode() method  
instead it uses System.identityHashCode() to find bucket location.  
HashMap uses hashCode() method to find bucket location.

---Differences between hashcode and system.identityHashCode

Assuming that it hasn't been overridden, the Object.hashCode() method  
simply calls System.identityHashCode(this).

The exact behavior of System.identityHashCode(Object) depends on the JVM  
implementation.

That integer returned by `identityHashCode` may be related to the (a) machine address for the object, or it may not be.

The value returned by `identityHashCode()` is guaranteed not to change for the lifetime of the object.

This means that if the GC relocates an object (after an `identityHashCode()` call) then it cannot use the new object address as the identity hashcode

The `hashCode()` method is a non-final instance method, and should be overridden in any class where the `equals(Object)` is overridden.

By contrast, `identityHashCode(Object)` is a static method and therefore cannot be overridden.

The `identityHashCode(Object)` method gives you a identifier for an object which can (in theory) be used for other things than hashing and hash tables. (Unfortunately, it is not a unique identifier, but it is guaranteed to never change for the lifetime of the object.)

4. Immutable keys : `IdentityHashMap` does not require keys to be immutable as it does not relies on `equals()` and `hashCode()` method to safely store the object in `HashMap` keys must be immutable.

5. Performance : According to `IdentityHashMap` Oracle docs, `IdentityHashMap` will yield better performance than `HashMap`(which uses chaining rather than linear probing Hashtable) for many jre implementations and operation mixes

6. Implementation : Both `IdentityHashMap` and `HashMap` are Hashtable based implementation of Map interface.

`IdentityHashMap` is a simple linear probe hashtable while `HashMap` uses chaining instead of linear probe in hashtable.

7. Initial capacity of default constructor : Initial capacity of `HashMap` is 16 by default .

Initial capacity of `IdentityHashMap` is 21 by default.

8. Added to jdk : `IdentityHashMap` is added to jdk in java version 1.4 .



HashMap class is introduced to the java development kit in java version 1.2 .

----LinearProbing

Used in hashtables and identityHashMap

When the hash function causes a collision by mapping a new key to a cell of the hash table that is already occupied by another key, linear probing searches the table for the closest following free location and inserts the new key there.

Lookups are performed in the same way, by searching the table sequentially starting at the position given by the hash function, until finding a cell with a matching key or an empty cell

## 214) Difference between hashmap and weakhashmap

1.Entry object Garbage Collected : In HashMap , entry object(entry object stores key-value pairs) is not eligible for garbage collection .

In other words, entry object will remain in the memory even if the key object associated with key-value pair is null.

According to WeakHashMap Oracle docs ,

An entry in a WeakHashMap will automatically be removed when its key is no longer in ordinary use

(even having mapping for a given key will not prevent the key from being discarded by the garbage collector that is made finalizable , finalized and then reclaimed).

When a key is discarded then its entry is automatically removed from the map , in other words, garbage collected.

2.Key objects Reference : In HashMap key objects have strong(also called soft) reference.

Each key object in the WeakHashMap is stored indirectly as the referent of a Weak reference(also called hard ) reference.

Therefore , a key will automatically be removed only after the weak references to it , both inside and outside of the map , have been cleared by the garbage collector. Check here for the difference between strong and weak reference.

3Automatic Size decrease : Calling size() method on HashMap object will return the same number of key-value pairs.

size will decrease only if remove() method is called explicitly on the HashMap object.

Because the garbage collector may discard keys at anytime, a WeakHashMap may behave as though an unknown thread is silently removing entries.

So it is possible for the size method to return smaller values over time.So, in WeakHashMap size decrease happens automatically.

4.Clone method : HashMap implements Cloneable interface . HashMap class clone() method returns the shallow copy of the object , although keys and values themselves are not cloned.

WeakHashMap does not implement Cloneable interface , it only implements Map interface. Hence , \*\*\*\*there is no clone() method in the WeakHashMap class.

5. Serialize and Deserialize objects : HashMap implements Serializable interface .

So HashMap class object state can be serialized or deserialized (i.e state of the HashMap object can be saved and again resume from the saved state).

\*\*\*WeakHashMap does not implement Serializable interface .

As a result , WeakHashMap object will not have any of their state serialized or deserialized.

(i.e state of the WeakHashMap object cannot be saved and again resume from the saved state).

## 215) Describe hashset?

```
public class HashSet<E>
extends AbstractSet<E>
implements Set<E>, Cloneable, java.io.Serializable

{
    private transient HashMap<E,Object> map;

    // Dummy value to associate with an Object in the backing Map

    private static final Object PRESENT = new Object();

    public HashSet() {
        map = new HashMap<>();
    }

    // SOME CODE ,i.e Other methods in Hash Set

    public boolean add(E e) {
        return map.put(e, PRESENT)==null;
    }

    // SOME CODE ,i.e Other methods in Hash Set
}
```

The main point to notice in above code is that put (key,value) will return

1. null , if key is unique and added to the map
2. Old Value of the key , if key is duplicate

So , in HashSet add() method , we check the return value of map.put(key,value) method with null value

i.e.

```
public boolean add(E e) {
    return map.put(e, PRESENT)==null;
}
```

So , if `map.put(key,value)` returns null ,then

```
map.put(e, PRESENT)==null
```

will return true and element is added to the HashSet.

So , if `map.put(key,value)` returns old value of the key ,then

```
map.put(e, PRESENT)==null
```

will return false and element is not added to the HashSet .

## 216) What copy technique internally used by HashSet clone() method ?

There are two copy techniques in every object oriented programming language , deep copy and shallow copy.

To create a clone or copy of the Set object, HashSet internally uses shallow copy in clone() method , the elements themselves are not cloned .

In other words , a shallow copy is made by copying the reference of the object.

## 217) Why HashSet does not have get(Object o) method ?

Most of the people get puzzled by hearing this question . This question tests the deep understanding of the HashSet class .This question helps the interviewer to know whether candidate has the idea about contains() method in HashSet class or not .So let jump to the answer

`get(Object o)` is useful when we have one information linked to other information just like key value pair found in HashMap .So using `get()` method on one information we can get the second information or vice-versa.

Unlike HashMap , HashSet is all about having unique values or unique objects . There is no concept of keys in HashSet .

The only information we can derive from the HashSet object is whether the element is present in the HashSet Object or not . If the element is not present in the HashSet then add it otherwise return true leaving HashSet object unchanged. Here, contains() method helps to provide this information.

Due to the above reason there is no `get(Object o)` method in HashSet. Use iterator to get the values from the Set

JAVA INTERVIEW QUESTIONS GENERIC,  
INNERCLAS,ENUMSET,EMUMAP,QUEUE, DEQUEUE

```
public static void retrieveValuesFromListMethod1(Set set)
{
    Iterator itr = set.iterator();
    while(itr.hasNext())
    {
        System.out.println(itr.next());
    }
}
```

## 218) What is and when to use Collections.emptySet() . What is the advantage of having emptySet in Collections class ?

According to Oracle docs ,Collections.emptySet() returns the empty immutable Set ,not containing null .

## 219) Why we call emptySet() method,as we can also create empty Set using constructor ?

Advantages of using emptySet() method over creating object using constructor are :

1. Immutable : You should prefer to use immutable collection against the mutable collections wherever possible .

It becomes handy as multiple threads accessing the same instance of object will see the same values.

2. Concise : You do not need to manually type out the generic type of the collection - normally it is inferred from the context of the method call.

3. Efficient : As emptySet() method dont create new objects , so they just reuse the existing empty and immutable object .

Although ,practically,this trick is not that handy , and rarely improves the performance

-----HashMap is not thread Safe so we have ConcurrentHashMap(thread safe).

## 220) Why Java do not have ConcurrentHashSet class just like ConcurrentHashMap , as we know HashSet is also not thread safe and internally use HashMap.

You can answer that there is no need to have ConcurrentHashSet class in Java . The reason is you can produce a ConcurrentHashSet backed by ConcurrentHashMap by using newSetFromMap method.

According to Oracle docs ,method newSetFromMap is defined as :

"Returns a set backed by the specified map. The resulting set displays the same ordering, concurrency, and performance characteristics as the backing map. In essence, this factory method provides a Set implementation corresponding to any Map implementation.

There is no need to use this method on a Map implementation that already has a corresponding Set implementation (such as HashMap or TreeMap)."

```
Set<Object> weakHashSet = Collections.newSetFromMap( new  
WeakHashMap<Object, Boolean>());
```

Parameters:

map - the backing map

Returns:

the set backed by the map

## 221) Linkedhashset

LinkedHashSet is the Hashtable and linked list implementation of the Set interface with predictable iteration order. The linked list defines the iteration ordering, which is the order in which elements were inserted into the set.

Insertion order is not affected if an element is re-inserted into the set.

```
public HashSet (int initialCapacity , float loadFactor , boolean dummy)  
{  
    map = new LinkedHashMap<>(initialCapacity , loadFactor);  
}
```

## 222) Why we need LinkedHashSet when we already have the HashSet and TreeSet ?

HashSet and TreeSet classes were added in jdk 1.2 while LinkedHashSet was added to the jdk in java version 1.4. HashSet provides constant time performance for basic operations like (add, remove and contains) method but elements are in chaotic ordering i.e unordered.

In TreeSet elements are naturally sorted but there is increased cost associated with it. So, LinkedHashSet is added in jdk 1.4 to maintain ordering of the elements without incurring increased cost.

What is Initial capacity and load factor?



The capacity is the number of buckets(used to store key and value) in the Hash table ,

and the initial capacity is simply the capacity at the time Hash table is created.

The load factor is a measure of how full the Hash table is allowed to get before its capacity is automatically increased.

The main point to notice in above code is that put (key,value) will return

1. null , if key is unique and added to the map
2. Old Value of the key , if key is duplicate

So , in HashSet add() method , we check the return value of map.put(key,value) method with null value

i.e.

```
public boolean add(E e) {  
    return map.put(e, PRESENT)==null;  
}
```

So , if map.put(key,value) returns null ,then

```
map.put(e, PRESENT)==null
```

will return true and element is added to the HashSet.

So , if map.put(key,value) returns old value of the key ,then

```
map.put(e, PRESENT)==null
```

will return false and element is not added to the LinkedHashSet .

## 223) TreeMap

Map interface ---> Sorted Map ---> Navigable Map ---> TreeMap

TreeMap class is like HashMap which stores key- value pairs . The major difference is that TreeMap sorts the key in ascending order.

TreeMap 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.

This implementation provides guaranteed  $\log(n)$  time cost for the containsKey, get, put and remove operations.

TreeMap is a Red-Black tree based NavigableMap implementation. In other words , it sorts the TreeMap object keys using Red-Black tree algorithm

"Natural" ordering is the ordering implied by the implementation of the Comparable interface by the objects used as keys in the TreeMap.

Essentially, RBTree must be able to tell which key is smaller than the other key, and there are two ways to supply that logic to the RBTree implementation:

1. Implement Comparable interface in the class(es) used as keys to TreeMap, or
2. Supply an implementation of the Comparator that would do comparing outside the key class itself.

Natural ordering is the order provided by the Comparable interface.

If somebody puts the key that do not implement natural order then it will throw ClassCastException

clone() method returns the shallow copy of the TreeMap instance .

In shallow copy object B points to object A location in memory . In other words , both object A and B are sharing the same elements .

The keys and values themselves are not cloned .

## 224) What happens if the TreeMap is concurrently modified while iterating the elements ?

The iterator fails fast and quickly if structurally modified at any time after the iterator is created (in any way except through the iterator's own remove method ).

We already discussed the difference between Fail-fast and Fail safe iterators

## 225) Why java's treemap does not allow an initial size ?

HashMap reallocates its internals as the new one gets inserted while TreeMap does not reallocate nodes on adding new ones. Thus , the size of the TreeMap dynamically increases if needed , without shuffling the internals. So it is meaningless to set the initial size of the TreeMap .

## 226) TreeSet

```
public TreeSet(Comparator comparator)
```

Constructs a new, empty tree set, sorted according to the specified comparator.

Parameters:

comparator - the comparator that will be used to order this set. If null, the natural ordering of the elements will be used.

TreeSet is like HashSet which contains the unique elements only but in a sorted manner.

The major difference is that TreeSet provides a total ordering of the elements.

The elements are ordered using their natural ordering, or by a Comparator typically provided at sorted set creation time.

The set's iterator will traverse the set in ascending element order.

whenever you are adding element to the TreeSet object , it works just like HashSet ,

The only difference is that instead of HashMap here we have TreeMap object in the constructor

we pass the argument in the add(Elmene E) that is E as a key in the TreeSet .

Now we need to associate some value to the key , so what Java apis developer did is to pass the Dummy value that is ( new Object () )

which is referred by Object reference PRESENT .

The main point to notice in above code is that put (key,value) will return

1. null , if key is unique and added to the map
2. Old Value of the key , if key is duplicate

So , in TreeSet add() method , we check the return value of map.put(key,value) method with null value

i.e.

```
public boolean add(E e) {
    return map.put(e, PRESENT)==null;
}
```

So , if map.put(key,value) returns null ,then

map.put(e, PRESENT)==null will return true and element is added to the TreeSet.

So , if map.put(key,value) returns old value of the key ,then

map.put(e, PRESENT)==null will return false and element is not added to the TreeSet .

## 227) How to find the index of any element in the TreeSet ?

There are many ways to find out the index of element in the TreeSet. Below is the one liner :

```
set.headSet(element).size()
```

Note : headSet(element) method returns the sub TreeSet(portion of TreeSet) whose values are less than input element.

Then we are calling size() method on the sub TreeSet , which returns the index of the element as sub TreeSet is already sorted

According to TreeSet Oracle doc :

TreeSet implementation provides guaranteed  $\log(n)$  time cost for the basic operations (add, remove and contains ) method.

-----What is natural ordering in TreeSet ?

"Natural" ordering is the ordering implied by the implementation of Comparable interface by the objects in the TreeSet .

Essentially RBTree must be able to tell which object is smaller than other object , and there are two ways to supply that logic to the RB Tree implementation :

1. We need to implement the Comparable interface in the class(es) used as objects in TreeSet.
2. Supply an implementation of the Comparator would do comparing outside the class itself.

TreeSet is fail fast .

clone method does shallow copy

-----How to convert HashSet to TreeSet object ?

One-liner : `Set treeObject = new TreeSet( hashSetObject);`

-----Advantages over HashSet

1.Null value : \*\*\*\*HashSet can store null object while TreeSet does not allow null object.

If one try to store null object in TreeSet object , it will throw Null Pointer Exception.

2.Functionality : TreeSet is rich in functionality as compare to HashSet.

Functions like pollFirst(),pollLast(),first(),last(),ceiling(),lower() etc. makes TreeSet easier to use than HashSet.

3.Comparision : HashSet uses equals() method for comparison in java while TreeSet uses compareTo() method for maintaining ordering .

4. TreeSet has greater locality than HashSet.

5.TreeSet uses Red- Black tree algorithm underneath to sort out the elements.

When one need to perform read/write operations frequently , then TreeSet is a good choice.

## 228) CopyOnWriteArrayList

CopyOnWriteArrayList is a concurrent Collection class introduced in Java 5 Concurrency API along with its popular cousin ConcurrentHashMap in Java.

CopyOnWriteArrayList implements List interface like ArrayList, Vector and LinkedList but its a thread-safe collection and it achieves its thread-safety in a slightly different way than Vector or other thread-safe collection class. As name suggest CopyOnWriteArrayList creates copy of underlying ArrayList with every mutation operation e.g. add or set. Normally CopyOnWriteArrayList is very expensive because it involves costly Array copy with every write operation but its very efficient

if you have a List where Iteration outnumber mutation e.g. you mostly need to iterate the ArrayList and don't modify it too often. Iterator of CopyOnWriteArrayList is fail-safe and doesn't throw ConcurrentModificationException even if underlying CopyOnWriteArrayList is modified once Iteration begins because Iterator is operating on separate copy of ArrayList.

Consequently all the updates made on CopyOnWriteArrayList is not available to Iterator.

## 229) Difference between CopyOnWriteArrayList and ArrayList in Java.

In last section we have seen What is CopyOnWriteArrayList in Java and How it achieves thread-safety by creating a separate copy of List for each write operation. Now let's see Some difference between ArrayList and CopyOnWriteArrayList in Java , which is another implementation of List interface :

- 1) First and foremost difference between CopyOnWriteArrayList and ArrayList in Java is that CopyOnWriteArrayList is a thread-safe collection while ArrayList is not thread-safe and can not be used in multi-threaded environment.
- 2) Second difference between ArrayList and CopyOnWriteArrayList is that Iterator of ArrayList is fail-fast and throw ConcurrentModificationException

once detect any modification in List once iteration begins but Iterator of CopyOnWriteArrayList is fail-safe and doesn't throw ConcurrentModificationException.

3) Third difference between CopyOnWriteArrayList vs ArrayList is that Iterator of former doesn't support remove operation while Iterator of later supports remove() operation. If the CopyOnWriteArrayList changes by another thread, the array you're currently observing will not be effected.

To get the most updated version do a new read like list.iterator();

That being said, updating this collection alot will kill performance.

If you tried to sort a CopyOnWriteArrayList you'll see the list throws an UnsupportedOperationException (the sort invokes set on the collection N times).

You should only use this read when you are doing upwards of 90+% reads.

## 230) concurrentSkiplistmap

uses compareAndSet method to acheive concurrency

same as treeMap but synchronized

\*\*\*\*TreeMap and ConcurrentSkipListMap both does not allow to store null key but allow many null values in java

A scalable concurrent ConcurrentNavigableMap implementation.

The map 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.

This class implements a concurrent variant of SkipLists 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.



Iterators are weakly consistent, returning elements reflecting the state of the map at some point at or since the creation of the iterator.

They do not throw `ConcurrentModificationException`, and may proceed concurrently with other operations.

Ascending key ordered views and their iterators are faster than descending ones.

All `Map.Entry` pairs returned by methods in this class and its views represent snapshots of mappings at the time they were produced.

They do not support the `Entry.setValue` method.

(Note however that it is possible to change mappings in the associated map using `put`, `putIfAbsent`, or `replace`, depending on exactly which effect you need.)

Beware that, unlike in most collections, the `size` method is not a constant-time operation.

Because of the asynchronous nature of these maps, determining the current number of elements requires a traversal of the elements,

and so may report inaccurate results if this collection is modified during traversal.

Additionally, the bulk operations `putAll`, `equals`, `toArray`, `containsValue`, and `clear` are not guaranteed to be performed atomically.

For example, an iterator operating concurrently with a `putAll` operation might view only some of the added elements.

This class and its views and iterators implement all of the optional methods of the `Map` and `Iterator` interfaces.

Like most other concurrent collections, this class does not permit the use of null keys or values

because some null return values cannot be reliably distinguished from the absence of elements.

This class is a member of the Java Collections Framework.

## 231) Concurrentskiplistset

A scalable concurrent `NavigableSet` implementation based on a `ConcurrentSkipListMap`.

The elements of the set are kept sorted according to their natural ordering, or by a Comparator provided at set creation time, depending on which constructor is used.

This 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.

Iterators are weakly consistent, returning elements reflecting the state of the set at some point at or since the creation of the iterator.

They do not throw ConcurrentModificationException, and may proceed concurrently with other operations.

Ascending ordered views and their iterators are faster than descending ones.

Beware that, unlike in most collections, the size method is not a constant-time operation.

Because of the asynchronous nature of these sets, determining the current number of elements requires a traversal of the elements,

and so may report inaccurate results if this collection is modified during traversal.

Additionally, the bulk operations addAll, removeAll, retainAll, containsAll, equals, and toArray are not guaranteed to be performed atomically.

For example, an iterator operating concurrently with an addAll operation might view only some of the added elements.

This class and its iterators implement all of the optional methods of the Set and Iterator interfaces.

\*\*\*\*\*Like most other concurrent collection implementations, this class does not permit the use of null elements,

because null arguments and return values cannot be reliably distinguished from the absence of elements.

This class is a member of the Java Collections Framework.

## 232) Priority queue

This class implements Queue interface and provides a sorted element from head of the queue.

Though it provides sorting, its little different with other Sorted collections e.g. TreeSet or TreeMap, which also allows you to iterate over all elements,

in priority queue there is no guarantee on iteration. The only guaranteed PriorityQueue gives is that lowest or highest priority element will be on head of the queue.

So when you call remove() or poll() method, you will get this element and next on priority will acquire the head spot.

Like other collection classes which provides sorting, PriorityQueue also uses Comparable and Comparator interface for priority.

when you add or remove elements from PriorityQueue, other elements are compared to each other to put highest/lowest priority element at head of the queue.

priority queue data structure is internally implemented using binary heap data structure,

which allows constant time access to maximum and minimum element in heap by implementing max heap and min heap data structure.

In these data structure root of the binary tree always contain either maximum value (max heap) or minimum value (min heap),

since you have constant time access to root, you can get max or minimum value in constant time.

Once this element is removed from root, next maximum/minimum is promoted to root.

So, if you want to process elements in their relative priority order, you can use PriorityQueue in Java.

It provides constant time access to highest or lowest priority element.

Important Points:

Doesnt allow null values.

If multiple elements are tied for least value then anyone of them can occupy head of the queue, ties are broken arbitrarily.

PriorityQueue is unbounded, which means you can add as many elements as you want, but its backed by array and has an internal capacity to govern size of array.

Iterator returned by PriorityQueue doesn't guarantee any ordered traversal. If you need ordered traversal consider using `Arrays.sort(pq.toArray());`

PriorityQueue class is not synchronized, so you should not share between multiple thread if one of the threads modifies the queue.

Time complexity of enqueueing and dequeing elements is in order of  $O(\log(n))$

Time complexity is linear for `remove(object)` and `contains(object)`

PriorityQueue provides constant time performance for `peek()`, `element()` and `size()` method,

which means you can retrieve maximum or minimum element in constant time from priority queue in Java.

## 233) Blocking queue

A Queue that additionally supports operations that wait for the queue to become non-empty when retrieving an element,

and wait for space to become available in the queue when storing an element.

BlockingQueue methods come in four forms, with different ways of handling operations that cannot be satisfied immediately,

but may be satisfied at some point in the future: one throws an exception, the second returns a special value

(either null or false, depending on the operation), the third blocks the current thread indefinitely until the operation can succeed,

and the fourth blocks for only a given maximum time limit before giving up. These methods are summarized in the following table:

	Throws exception	Special value	Blocks	Times out
Insert	<code>add(e)</code>	<code>offer(e)</code>	<code>put(e)</code>	<code>offer(e, time, unit)</code>
Remove	<code>remove()</code>	<code>poll()</code>	<code>take()</code>	<code>poll(time, unit)</code>
Examine	<code>element()</code>	<code>peek()</code>	not applicable	not applicable

**\*\*A BlockingQueue does not accept null elements. Implementations throw NullPointerException on attempts to add, put or offer a null.**

A null is used as a sentinel value to indicate failure of poll operations.

**\*\*A BlockingQueue may be capacity bounded. At any given time it may have a remainingCapacity beyond which no additional elements can be put without blocking.**

A BlockingQueue without any intrinsic capacity constraints always reports a remaining capacity of Integer.MAX\_VALUE.

BlockingQueue implementations are designed to be used primarily for producer-consumer queues, but additionally support the Collection interface.

So, for example, it is possible to remove an arbitrary element from a queue using remove(x).

However, such operations are in general not performed very efficiently, and are intended for only occasional use, such as when a queued message is cancelled.

BlockingQueue implementations are thread-safe. All queuing methods achieve their effects atomically using internal locks or other forms of concurrency control.

However, the bulk Collection operations addAll, containsAll, retainAll and removeAll are not necessarily performed atomically

unless specified otherwise in an implementation.

So it is possible, for example, for addAll(c) to fail (throwing an exception) after adding only some of the elements in c.

A BlockingQueue does not intrinsically support any kind of "close" or "shutdown" operation to indicate that no more items will be added.

The needs and usage of such features tend to be implementation-dependent.

For example, a common tactic is for producers to insert special end-of-stream or poison objects, that are interpreted accordingly when taken by consumers.

All Known Implementing Classes:

ArrayBlockingQueue, DelayQueue, LinkedBlockingDeque,  
LinkedBlockingQueue, LinkedTransferQueue, PriorityBlockingQueue,  
SynchronousQueue

## 234) Queue

<http://www.sanfoundry.com/java-program-priority-queue/>

<http://www.journaldev.com/1034/java-blockingqueue-example-implementing-producer-consumer-problem>

<http://stackoverflow.com/questions/12554390/producer-consumer-multithreading>

<http://tutorials.jenkov.com/java-concurrency/blocking-queues.html>

<http://www.java-samples.com/showtutorial.php?tutorialid=306>

<http://www.journaldev.com/1037/java-thread-wait-notify-and-notifyall-example>

<http://howtodoinjava.com/core-java/multi-threading/how-to-work-with-wait-notify-and-notifyall-in-java/>

## 235) NavigableMap

NavigableMap is an extension of [SortedMap](#) which provides convenient navigation method like lowerKey, floorKey, ceilingKey and higherKey, and along with these popular navigation method it also provide ways to create a Sub Map from existing Map in Java e.g. headMap whose keys are less than specified key, tailMap whose keys are greater than specified key and a subMap which is strictly contains keys which falls between toKey and fromKey.

An example class that implements NavigableMap is [TreeMap](#).

**Methods** of NavigableMap:

**lowerKey(Object key)** : Returns the greatest key strictly less than the given key, or if there is no such key.

**floorKey(Object key)** : Returns the greatest key less than or equal to the given key, or if there is no such key.

**ceilingKey(Object key)** : Returns the least key greater than or equal to the given key, or if there is no such key.

**higherKey(Object key)** : Returns the least key strictly greater than the given key, or if there is no such key.

**descendingMap()** : Returns a reverse order view of the mappings contained in this map.

**headMap(object toKey, boolean inclusive)** : Returns a view of the portion of this map whose keys are less than (or equal to, if inclusive is true) toKey.

**subMap(object fromKey, boolean fromInclusive, object toKey, boolean toInclusive)** : Returns a view of the portion of this map whose keys range from fromKey to toKey.

**tailMap(object fromKey, boolean inclusive)** : Returns a view of the portion of this map whose keys are greater than (or equal to, if inclusive is true) fromKey.

```
// Java program to demonstrate NavigableMap
import java.util.NavigableMap;
import java.util.TreeMap;

public class Example
{
    public static void main(String[] args)
    {
        NavigableMap<String, Integer> nm =
            new TreeMap<String, Integer>();

        nm.put("C", 888);
        nm.put("Y", 999);
        nm.put("A", 444);
        nm.put("T", 555);
        nm.put("B", 666);
        nm.put("A", 555);

        System.out.printf("Descending Set : %s\n",
            nm.descendingKeySet());
    }
}
```

```
        System.out.printf("Floor Entry  : %s\n",
                           nm.floorEntry("L"));
        System.out.printf("First Entry  : %s\n",
                           nm.firstEntry());
        System.out.printf("Last Key   : %s\n",
                           nm.lastKey());
        System.out.printf("First Key  : %s\n",
                           nm.firstKey());
        System.out.printf("Original Map : %s\n", nm);
        System.out.printf("Reverse Map : %s\n",
                           nm.descendingMap());
    }
}
```

Output:

```
Descending Set : [Y, T, C, B, A]
Floor Entry   : C=888
First Entry   : A=555
Last Key      : Y
First Key     : A
Original Map   : {A=555, B=666, C=888, T=555, Y=999}
Reverse Map    : {Y=999, T=555, C=888, B=666, A=555}
```

## ANNOTATION

### 236) What is annotation?

Java Annotations provide information about the code and they have no direct effect on the code they annotate. Annotations are introduced in Java 5. Annotation is metadata about the program embedded in the program itself. It can be parsed by the annotation parsing tool or by compiler. We can also specify annotation availability to either compile time only or till runtime also. Java Built-in annotations are @Override, @Deprecated and @SuppressWarnings. Read more at [java annotations](#).

This is metadata for code.

<https://dzone.com/articles/how-annotations-work-java>

<https://beginnersbook.com/2014/09/java-annotations/>

<http://tutorials.jenkov.com/java/annotations.html>



*Annotations*, a form of metadata, provide data about a program that is not part of the program itself. Annotations have no direct effect on the operation of the code they annotate.

Annotations have a number of uses, 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.

**Runtime processing** — Some annotations are available to be examined at runtime.

- annotation starts with @
- annotation can have one and multiple values. If only one attribute then attribute name must be value.
- We can give array in annotation attribute
- @Repeated means same annotation on same place can be used multiple times as @CodeModifucation; This is introduced in java 8.

```
@CodeModification("harbhajan");
```

```
@CodeModification("CJ");
```

```
Class A{
```

```
}
```

- Annotation can be with element(@Author(name="hk")) or without element (@Author)

## 237) On which fields/methods/place can we use annotation?

Annotations can be applied to declarations: declarations of classes, fields, methods, and other program elements. When used on a declaration, each annotation often appears, by convention, on its own line.

As of the Java SE 8 release, annotations can also be applied to the *use* of types.

## 238) How to declare custom annotation?

- Annotations are created by using @interface, followed by annotation name as shown in the below example.
- An annotation can have elements as well. They look like methods. For example in the below code, we have four elements. We should not provide implementation for these elements.
- All annotations extends java.lang.annotation.Annotation interface. Annotations cannot include any extends clause.

```
import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Documented
@Target(ElementType.METHOD)
@Inherited
@Retention(RetentionPolicy.RUNTIME)
public @interface MyCustomAnnotation{
    int studentAge() default 18;
    String studentName();
    String stuAddress();
    String stuStream() default "CSE";
}
```

<https://beginnersbook.com/2014/09/java-annotations/>

## 239) What is meta-annotations?

Annotations that apply to other annotations are called *meta-annotations*. There are several meta-annotation types defined in `java.lang.annotation`.

**@Retention** `@Retention` annotation specifies how the marked annotation is stored:

`RetentionPolicy.SOURCE` – The marked annotation is retained only in the source level and is ignored by the compiler.

`RetentionPolicy.CLASS` – The marked annotation is retained by the compiler at compile time, but is ignored by the Java Virtual Machine (JVM).

`RetentionPolicy.RUNTIME` – The marked annotation is retained by the JVM so it can be used by the runtime environment.

**@Documented** `@Documented` annotation indicates that whenever the specified annotation is used those elements should be documented using the Javadoc tool. (By default, annotations are not included in Javadoc.) For more information, see the [Javadoc tools page](#).

**@Target** `@Target` annotation marks another annotation to restrict what kind of Java elements the annotation can be applied to. A target annotation specifies one of the following element types as its value:

- `ElementType.ANNOTATION_TYPE` can be applied to an annotation type.
- `ElementType.CONSTRUCTOR` can be applied to a constructor.
- `ElementType.FIELD` can be applied to a field or property.
- `ElementType.LOCAL_VARIABLE` can be applied to a local variable.
- `ElementType.METHOD` can be applied to a method-level annotation.
- `ElementType.PACKAGE` can be applied to a package declaration.
- `ElementType.PARAMETER` can be applied to the parameters of a method.
- `ElementType.TYPE` can be applied to any element of a class.

**@Inherited** `@Inherited` annotation indicates that the annotation type can be inherited from the super class. (This is not true by default.) When the user queries the annotation type and the class has no annotation for this type, the class' superclass is queried for the annotation type. This annotation applies only to class declarations.

The @Inherited annotation signals that a custom annotation used in a class should be inherited by all of its sub classes. For example:

java.lang.annotation.[Inherited](#)

```
@Inherited
public @interface MyCustomAnnotation {

}

@MyCustomAnnotation
public class MyParentClass {
    ...
}

public class MyChildClass extends MyParentClass {
    ...
}
```

Here the class `MyParentClass` is using annotation `@MyCustomAnnotation` which is marked with `@inherited` annotation. It means the sub class `MyChildClass` inherits the `@MyCustomAnnotation`.

**@Repeatable** [@Repeatable](#) annotation, introduced in Java SE 8, indicates that the marked annotation can be applied more than once to the same declaration or type use. For more information, see [Repeating Annotations](#).

## 240) Default java annotation?

**@Deprecated** `@Deprecated` annotation indicates that the marked element is *deprecated* and should no longer be used. The compiler generates a warning whenever a program uses a method, class, or field with the `@Deprecated` annotation. When an element is deprecated, it should also be documented using the Javadoc `@deprecated` tag, as shown in the following example. The use of the at sign (`@`) in both Javadoc comments and in annotations is not coincidental: they are related conceptually. Also, note that the Javadoc tag starts with a lowercase *d* and the annotation starts with an uppercase *D*.

```
// Javadoc comment follows
/**
 * @deprecated
 * explanation of why it was deprecated
 */
@Deprecated
static void deprecatedMethod() { }
}
```

**@Override** `@Override` annotation informs the compiler that the element is meant to override an element declared in a superclass. Overriding methods will be discussed in [Interfaces and Inheritance](#).

```
// mark method as a superclass method
// that has been overridden
@Override
int overriddenMethod() { }
```

While it is not required to use this annotation when overriding a method, it helps to prevent errors. If a method marked with `@Override` fails to correctly override a method in one of its superclasses, the compiler generates an error.

**@SuppressWarnings** `@SuppressWarnings` annotation tells the compiler to suppress specific warnings that it would otherwise generate. In the following example, a deprecated method is used, and the compiler usually generates a warning. In this case, however, the annotation causes the warning to be suppressed.

```
// use a deprecated method and tell
// compiler not to generate a warning
@SuppressWarnings("deprecation")
void useDeprecatedMethod() {
    // deprecation warning
    // - suppressed
    objectOne.deprecatedMethod();
}
```

Every compiler warning belongs to a category. The Java Language Specification lists two categories: `deprecation` and `unchecked`. The `unchecked` warning can occur when interfacing with legacy code written before the advent of [generics](#). To suppress multiple categories of warnings, use the following syntax:

```
@SuppressWarnings({"unchecked", "deprecation"})
```

**@SafeVarargs** `@SafeVarargs` annotation, when applied to a method or constructor, asserts that the code does not perform potentially unsafe operations on its `varargs` parameter. When this annotation type is used, unchecked warnings relating to `varargs` usage are suppressed.

**@FunctionalInterface** `@FunctionalInterface` annotation, introduced in Java SE 8, indicates that the type declaration is intended to be a functional interface, as defined by the Java Language Specification.

## 241) What is repeated annotation?

<https://docs.oracle.com/javase/tutorial/java/annotations/repeating.html>

These are annotations that can be applied more than once to the same element declaration.

For compatibility reasons, since this feature was introduced in Java 8, repeating annotations are stored in a *container annotation* that is automatically generated by the Java compiler. For the compiler to do this, there are two steps to declared them.

First, we need to declare a repeatable annotation:

```
1      @Repeatable(Schedules.class)
2      public @interface Schedule {
3          String time() default "morning";
4      }
```

Then, we define the containing annotation with a mandatory *value* element, and whose type must be an array of the repeatable annotation type:

```
1      public @interface Schedules {
2          Schedule[] value();
3      }
```

Now, we can use `@Schedule` multiple times:

```
1      @Schedule
2      @Schedule(time = "afternoon")
3      @Schedule(time = "night")
4      void scheduledMethod() {
5          // ...
6      }
```

## 242) What is Type annotation?

Before the Java SE 8 release, annotations could only be applied to declarations. As of the Java SE 8 release, annotations can also be applied to any *type use*. This means that annotations can be used anywhere you use a type. A few examples of where types are used are class instance creation expressions (`new`), casts, `implements` clauses, and `throws` clauses.

- ✓ Class instance creation expression:

```
new @Interned MyObject();
```

- ✓ Type cast:

```
myString = (@NonNull String) str;
```

- ✓ `implements` clause:

```
class UnmodifiableList<T> implements
    @ReadOnly List<@ReadOnly T> { ... }
```

- ✓ Thrown exception declaration:

```
void monitorTemperature() throws
    @Critical TemperatureException { ... }
```

This form of annotation is called a *type annotation*

## 243) What Object Types Can Be Returned from an Annotation Method Declaration?

The return type must be a primitive, *String*, *Class*, *Enum*, or an array of one of the previous types. Otherwise, the compiler will throw an error.

```
public @interface FailingAnnotation {
    Object complexity();
}
```

This will give compiler error.

## 244) How Can You Retrieve Annotations? How Does This Relate to Its Retention Policy?

You can use the Reflection API or an annotation processor to retrieve annotations.

The **@Retention** annotation and its **RetentionPolicy** parameter affect how you can retrieve them. There are three constants in **RetentionPolicy** enum:

**RetentionPolicy.SOURCE** – makes the annotation to be discarded by the compiler but annotation processors can read them

**RetentionPolicy.CLASS** – indicates that the annotation is added to the class file but not accessible through reflection

**RetentionPolicy.RUNTIME** –Annotations are recorded in the class file by the compiler and retained by the JVM at runtime so that they can be read reflectively

Here's an example code to create an annotation that can be read at runtime:

```
1      @Retention(RetentionPolicy.RUNTIME)
2      public @interface Description {
3          String value();
4      }
```

Now, annotations can be retrieved through reflection:

```
1      Description description
2          = AnnotatedClass.class.getAnnotation(Description.class);
3      System.out.println(description.value());
```

An annotation processor can work with **RetentionPolicy.SOURCE**, this is described in the article [Java Annotation Processing and Creating a Builder](#).

**RetentionPolicy.CLASS** is usable when you're writing a Java bytecode parser.



## 245) Will code compile?

```

1      @Target({ ElementType.FIELD, ElementType.TYPE, ElementType.FIELD
2      public @interface TestAnnotation {
3          int[] value() default {};
4      }

```

No. It's a compile-time error if the same enum constant appears more than once in an *@Target* annotation.

Removing the duplicate constant will make the code to compile successfully:

```

1      @Target({ ElementType.FIELD, ElementType.TYPE})

```

## 246) Can we extend annotation?

No. Annotations always extend *java.lang.annotation.Annotation*, as stated in the [Java Language Specification](#).

If we try to use the *extends* clause in an annotation declaration, we'll get a compilation error:

```

1      public @interface AnAnnotation extends OtherAnnotation {
2          // Compilation error
3      }

```

# REFLECTION API

## 247) Describe reflection API

- ✓ Reflection is an API which is used to examine or modify the behavior of methods, classes, interfaces at runtime.
- ✓ The required classes for reflection are provided under java.lang.reflect package.
- ✓ Reflection gives us information about the class to which an object belongs and also the methods of that class which can be executed by using the object.
- ✓ Through reflection we can invoke methods at runtime irrespective of the access specifier used with them.

## 248) Use of reflection api

. Even though we don't use Reflection API in normal programming, it's very important to have. We can't have any frameworks such as Spring, Hibernate or servers such as Tomcat, JBoss without Reflection API. They invoke the appropriate methods and instantiate classes through reflection API and use it a lot for other processing.

Read [Java Reflection Tutorial](#) to get in-depth knowledge of reflection api.

The Reflection API is mainly used in:

- ✓ Frameworks
- ✓ IDE (Integrated Development Environment) e.g. Eclipse, MyEclipse, NetBeans etc.
- ✓ Debugger
- ✓ Test Tools etc.
- ✓ **Extensibility Features**
- ✓ An application may make use of external, user-defined classes by creating instances of extensibility objects using their fully-qualified names.
- ✓ Class Browsers and Visual Development Environments
- ✓ A class browser needs to be able to enumerate the members of classes. Visual development environments can benefit from making use of type information available in reflection to aid the developer in writing correct code.
- ✓ Debuggers and Test Tools
- ✓ Debuggers need to be able to examine private members on classes. Test harnesses can make use of reflection to systematically call a discoverable set APIs defined on a class, to insure a high level of code coverage in a test suite.

## 249) Class in reflection?

Every type is either a reference or a primitive. Classes, enums, and arrays (which all inherit from `java.lang.Object`) as well as interfaces are all reference types. Examples of reference types include `java.lang.String`, all of the wrapper classes for primitive types such as `java.lang.Double`, the interface `java.io.Serializable`, and the enum `javax.swing.SortOrder`.

There is a fixed set of primitive

types: `boolean`, `byte`, `short`, `int`, `long`, `char`, `float`, and `double`.

**For every type of object, the Java virtual machine instantiates an immutable instance of `java.lang.Class`** which provides methods to examine the runtime properties of the object including its members and type information. `Class` also provides the ability to create new classes and objects. Most importantly, it is the entry point for all of the Reflection APIs. This lesson covers the most commonly used reflection operations involving classes:

[Retrieving Class Objects](#) describes the ways to get a `Class`

[Examining Class Modifiers and Types](#) shows how to access the class declaration information

[Discovering Class Members](#) illustrates how to list the constructors, fields, methods, and nested classes in a class

[Troubleshooting](#) describes common errors encountered when using `Class`

**Objects of these types are created by the JVM at run time to represent the corresponding member in the unknown class.**

## 250) How can we get Class?

There are several ways to get a `Class` depending on whether the code has access to an object, the name of class, a type, or an existing `Class`.

- If an instance of an object is available, then the simplest way to get its `Class` is to invoke `Object.getClass()`. Of course, this only works for reference types which all inherit from `Object`. Some examples follow.

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

- If the type is available but there is no instance then it is possible to obtain a `Class` by appending `".class"` to the name of the type. This is also the easiest way to obtain the `Class` for a primitive type.

```
boolean b;
```

```
Class c = b.getClass();    // compile-time error
```

```
Class c = boolean.class;  // correct
```

- If the fully-qualified name of a class is available, it is possible to get the corresponding `Class` using the static method `Class.forName()`. This cannot be used for primitive types. The syntax for names of array classes is described by `Class.getName()`. This syntax is applicable to references and primitive types.

```
Class c =
```

```
Class.forName("com.duke.MyLocaleServiceProvider");
```

- TYPE Field for Primitive Type Wrappers

The `.class` syntax is a more convenient and the preferred way to obtain the `Class` for a primitive type; however there is another way to acquire the `Class`. Each of the primitive types and `void` has a wrapper class in `java.lang` that is used for boxing of primitive types to reference types. Each wrapper class contains a field named `TYPE` which is equal to the `Class` for the primitive type being wrapped.

`Class c = Double.TYPE;`

There is a class `java.lang.Double` which is used to wrap the primitive type `double` whenever an `Object` is required. The value of `Double.TYPE` is identical to that of `double.class`.

`Class c = Void.TYPE;`

`Void.TYPE` is identical to `void.class`.

## 251) Reflection API methods

<https://www.geeksforgeeks.org/reflection-in-java/>

[https://www.interviewgrid.com/interview\\_questions/java/java\\_reflection](https://www.interviewgrid.com/interview_questions/java/java_reflection)

Once class is available we can get-

- Class modifier
- Class inheritance path
- All fields
- All constructor
- All method
- All annotation
- Invoke method
- And so on
- Package info
- Interface info

1) `public String getName()` returns the class name

2) `public static Class.forName(String className)` throws `ClassNotFoundException` loads the class and returns the reference of `Class` class.

3) `public Object newInstance()` throws `InstantiationException, IllegalAccessException` creates new instance.

4) `public boolean isInterface()` checks if it is interface.

5) `public boolean isArray()` checks if it is array.

6) `public boolean isPrimitive()` checks if it is primitive.

7) `public Class getSuperclass()` returns the superclass class reference

8) `public Field[] getDeclaredFields()` throws `SecurityException` returns the total number of fields of this class.

- 9) public Method[ ] getDeclaredMethods()throws SecurityException  
returns the total number of methods of this class.
- 10) public Constructor[ ] getDeclaredConstructors()throws SecurityException  
returns the total number of constructors of this class.
- 11) public Method getDeclaredMethod(String name,Class[ ]  
parameterTypes)throws NoSuchMethodException,SecurityException  
returns the method class instance.

## 252) Why reflection is slower?

Because it has to inspect the metadata in the bytecode instead of just using pre compiled addresses and constants.

## 253) difference between RTTI and reflection

difference between RTTI and reflection is that with RTTI, the compiler opens and examines the .class file at compile time. Put another way, you can call all the methods of an object in the "normal" way. With reflection, the .class file is unavailable at compile time; it is opened and examined by the runtime environment

## 254) Drawbacks of Reflection

Reflection is powerful, but should not be used indiscriminately. If it is possible to perform an operation without using reflection, then it is preferable to avoid using .The following concerns should be kept in mind when accessing code via reflection:

Breaks singleton, design pattern as abstraction, encapsulation.

### **Performance Overhead**

Because reflection involves types that are dynamically resolved, certain Java virtual machine optimizations can not be performed. Consequently, reflective operations have slower performance than their non-reflective counterparts, and should be avoided in sections of code which are called frequently in performance-sensitive applications.

### **Security Restrictions**

Reflection requires a runtime permission which may not be present when running under a security manager. This is an important consideration for code which has to run in a restricted security context, such as in an Applet.

## Exposure of Internals

Since reflection allows code to perform operations that would be illegal in non-reflective code, such as accessing private fields and methods, the use of reflection can result in unexpected side-effects, which may render code dysfunctional and may destroy portability. Reflective code breaks abstractions and therefore may change behavior with upgrades of the platform.

## ENUM

### 255) Describe Enum?

Enum was introduced in Java 1.5 as a new type whose fields consists of fixed set of constants. For example, in Java we can create Direction as enum with fixed fields as EAST, WEST, NORTH, SOUTH.

enum is the keyword to create an enum type and similar to class. Enum constants are implicitly static and final. Read more in detail at [java enum](#).

- Enumerations serve the purpose of representing a group of named constants in a programming language.
- Enum are compile time constant.
- Enum can have variable.method
- Enum.values() used to iterate the enum.
- Enum is constant. Best use of singleton. Enum declaration simple is as follows – but internally enum is class and every constant is object of its type.

```
enum Color
{
    RED, GREEN, BLUE;
}
```

Is translated to

```
/* internally above enum Color is converted to
class Color
{
    public static final Color RED = new Color();
    public static final Color BLUE = new Color();
    public static final Color GREEN = new Color();
}*
```

- enum type can be passed as an argument to **switch** statement.
- Enum can be declared outside the class Or inside the class but not inside methods.
- Every enum constant is always implicitly public static final. Since it is static, we can access it by using enum Name. Since it is final, we can't create child enums.
- All enums implicitly extend java.lang.Enum class. As a class can only extend one parent in Java, so an enum cannot extend anything else.
- toString() method is overridden in java.lang.Enum class, which returns enum constant name.
- enum can implement many interfaces.

#### **enum methods:**

- These methods are present inside java.lang.Enum.
- values() method can be used to return all values present inside enum.
- Order is important in enums. By using ordinal() method, each enum constant index can be found, just like array index.
- valueOf() method returns the enum constant of the specified string value, if exists.

#### **enum and constructor :**

- enum can contain constructor and it is executed separately for each enum constant at the time of enum class loading.
- We can't create enum objects explicitly and hence we can't invoke enum constructor directly.

#### **enum and methods :**

- enum can contain concrete methods only i.e. no any abstract method.

## 256) Use of enum

- You should always use enums when a variable (especially a method parameter) can only take one out of a small set of possible values. If you use enums instead of integers (or String codes), you increase compile-time checking and avoid errors from passing in invalid constants, and you document which values are legal to use.
- can be used to implement singleton pattern. The enum singleton does not provide lazy creation either
- Singletons created at class load time (whether by enum or static final initialization) are thread safe.
- Lazily created ones are not unless explicitly synchronized

When you create an enum, an associated class is produced for you by the compiler. This class is automatically inherited from `java.lang.Enum`. You can step through the list of enum constants by calling `values()` on the enum.

The `values()` method produces an array of the enum constants in the order in which they were declared, so you can use the resulting array in (for example) a `foreach` loop. The `ordinal()` method produces an `int` indicating the declaration order of each enum instance, starting from zero.

You can always safely compare enum instances using `==`, and `equals()` and `hashCode()` are automatically created for you.

The `Enum` class is `Comparable`, so there's a `compareTo()` method, and it is also `Serializable`.

If you call `getDeclaringClass()` on an enum instance, you'll find out the enclosing enum class.

The `name()` method produces the name exactly as it is declared, and this is what you get with `toString()`, as well.

`valueOf()` is a static member of `Enum`, and produces the enum instance that corresponds to the `String` name you pass to it, or throws an exception if there's no match.

## 257) Interesting points about Enum:

- enums can't be inherited
- constructor must be private or default
- Java forces you to define the instances as the first thing in the enum
- can have main method just like a normal class.
- can implement interface
- Categorization can be done using interfaces.



## 258) enumMap

EnumMap is specialized implementation of [Map interface](#) for [enumeration types](#). It extends AbstractMap and implements [Map](#) Interface in Java. Few important features of EnumMap are as follows:

- EnumMap class is a member of the [Java Collections Framework](#) & is not synchronized.
- EnumMap is ordered collection and they are maintained in the natural order of their keys( natural order of keys means the order on which enum constant are declared inside enum type )
- It's a high performance map implementation, much faster than [HashMap](#).
- All keys of each EnumMap instance must be keys of a single [enum](#) type.
- EnumMap doesn't allow null key and throw NullPointerException, at same time null values are permitted.

// Java program to illustrate working of EnumMap and

// its functions.

```
import java.util.EnumMap;
```

```
public class Example
```

```
{
```

```
    public enum GFG
```

```
    {
```

```
        CODE, CONTRIBUTE, QUIZ, MCQ;
```

```
    }
```

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

```
    {
```

```
        // Java EnumMap
```

```
        // Creating EnumMap in java with key
```

```
        // as enum type STATE
```

```
        EnumMap<GFG, String> gfgMap = new
```

```
            EnumMap<GFG, String>(GFG.class);
```

```
        // Java EnumMap Example 2:
```

```
// Putting values inside EnumMap in Java
// Inserting Enum keys different from
// their natural order
gfgMap.put(GFG.CODE, "Start Coding with gfg");
gfgMap.put(GFG.CONTRIBUTE, "Contribute for others");
gfgMap.put(GFG.QUIZ, "Practice Quizes");
gfgMap.put(GFG.MCQ, "Test Speed with Mcqs");

// Printing size of EnumMap in java
System.out.println("Size of EnumMap in java: " +

gfgMap.size());

// Printing Java EnumMap
// Print EnumMap in natural order
// of enum keys (order on which they are declared)
System.out.println("EnumMap: " + gfgMap);

// Retrieving value from EnumMap in java
System.out.println("Key : " + GFG.CODE +" Value: "

+
gfgMap.get(GFG.CODE));

// Checking if EnumMap contains a particular key
System.out.println("Does gfgMap has "+GFG.CONTRIBUTE+": "

+
gfgMap.containsKey(GFG.CONTRIBUTE));

// Checking if EnumMap contains a particular value
System.out.println("Does gfgMap has :"+ GFG.QUIZ + " : "

+
gfgMap.containsValue("Practice Quizes"));
```

```

        System.out.println("Does gfgMap has :\" + GFG.QUIZ + \" : \"
                                +
        gfgMap.containsValue(null));
    }
}

```

EnumSet : EnumSet.of() , EnumSet.noneOf() ---- how enumset works internally ??

EnumMap : can have instances from one enum...so can be implemented internally as hashmap ----- how enumMap works ?

Pending ?

## ENUMERATION

### 259) Explain enumeration?

The Enumeration interface defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects.

This legacy interface has been superceded by Iterator. Although not deprecated, Enumeration is considered obsolete for new code. However, it is used by several methods defined by the legacy classes such as Vector and Properties, is used by several other API classes, and is currently in widespread use in application code.

The methods declared by Enumeration are summarized in the following table

—

Sr.No.	Method & Description
1	<b>boolean hasMoreElements( )</b>  When implemented, it must return true while there are still more elements to extract, and false when all the elements have been enumerated.
2	<b>Object nextElement( )</b>

This returns the next object in the enumeration as a generic Object reference.

3) enum is data type and enumeration is I nterface. V is vector here.

```
for (Enumeration<E> e = v.elements(); e.hasMoreElements();)
```

```
System.out.println(e.nextElement());
```

4) It is used by many data structures especially with legacy classes. The DS of JDK 1.0 are known as legacy classes. They are Stack, Vector, Hashtable and Properties.

## CLONEABLE

### 260) Explain cloneable?

- Object Cloning is a process of generating the exact field-to-field copy of object with the different name
- Cloneable interface is present in java.lang package.
- This is marker interface.
- There is a method clone() in [Object](#) class. A class that implements the Cloneable interface indicates that it is legal for clone() method to make a field-for-field copy of instances of that class. Invoking Object's clone method on an instance of the class that does not implement the Cloneable interface results in an exception CloneNotSupportedException being thrown. By convention, classes that implement this interface should override Object.clone() method.
- During clone() method call we handle CloneNotSupportedException using try catch blocks.
- Clone does not invoke constructor.
- Object.clone support shallow copy and not deep copy.
- Finals fields can not be manipulated in clone , we need constructor to manipulate final fields.

```
public class Employee implements Cloneable {  
  
    private String name;  
  
    public Employee(String name) {  
        this.name = name;  
    }  
}
```

```
public String getName() {  
    return name;  
}  
  
public Object clone() throws CloneNotSupportedException{  
    return (Employee) super.clone();  
}  
}
```

## 261) Does clone object and original object point to the same location in memory

The answer is no. The clone object has its own space in the memory where it copies the content of the original object. That's why when we change the content of original object after cloning, the changes does not reflect in the clone object.

## 262) Difference between clone and constructor?

- Clone method does not invoke constructor
- clone copy the object, whereas constructor set all variable to default value or given values.

## 263) Inheritance in clone

- If you are writing clone in child, then all aren't should implement cloneable otherwise super.clone() chain will fail.

## 264) Shallow copy

Shallow copies duplicate as little as possible. A shallow copy of a collection is a copy of the collection structure, not the elements..With a shallow copy, two collections now share the individual elements. It can be done by calling clone method.

Shallow clone is a copying the reference pointer to the object, which mean the new object is pointing to the same memory reference of the old object. The memory usage is lower.

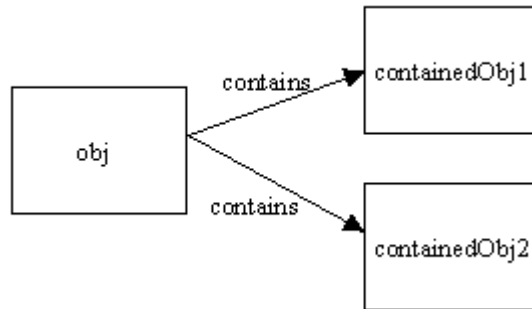


Figure 1: Original java object obj

The shallow copy is done for obj and new object obj1 is created but contained objects of obj are not copied.

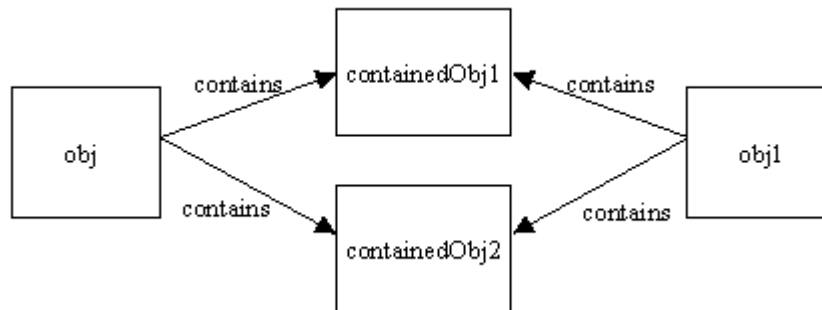


Figure 2: Shallow copy object obj1

It can be seen that no new objects are created for obj1 and it is referring to the same old contained objects. If either of the containedObj contain any other object no new reference is created.

## 265) Deep copy

Deep copies duplicate everything. A deep copy of a collection is two collections with all of the elements in the original collection duplicated need to override clone and provide functionality

If only primitive type fields or Immutable objects are there then there is no difference between shallow and deep copy in Java.

In deep copy is the copy of object itself. A new memory is allocated for the object and contents are copied.

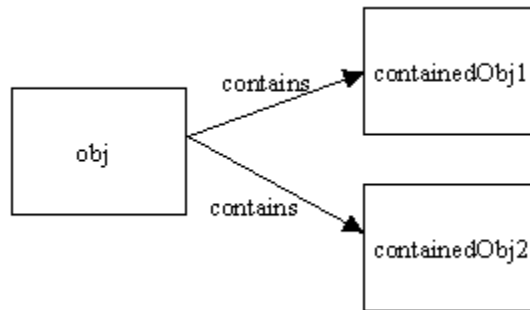


Figure 3 : Original Object obj

When a deep copy of the object is done new references are created.

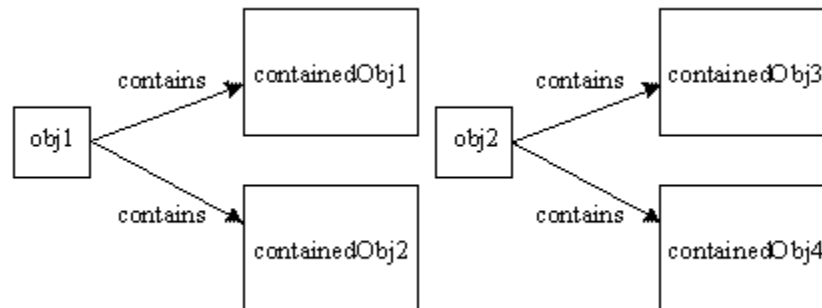


Figure 4: obj2 is deep copy of obj1

One solution is to simply implement your own custom method (e.g., `deepCopy()`) that returns a deep copy of an instance of one of your classes. This may be the best solution if you need a complex mixture of deep and shallow copies for different fields, but has a few significant drawbacks:

You must be able to modify the class (i.e., have the source code) or implement a subclass. If you have a third-party class for which you do not have the source and which is marked `final`, you are out of luck.

You must be able to access all of the fields of the classe's superclasses. If significant parts of the object's state are contained in private fields of a superclass, you will not be able to access them.

You must have a way to make copies of instances of all of the other kinds of objects that the object references. This is particularly problematic if the exact classes of referenced objects cannot be known until runtime.

Custom deep copy methods are tedious to implement, easy to get wrong, and difficult to maintain. The method must be revisited any time a change is made to the class or to any of its superclasses.

Other common solution to the deep copy problem is to use **Java Object Serialization** (JOS). The idea is simple: Write the object to an array using **ObjectOutputStream** and then use **ObjectInputStream** to reconstitute a copy of the object. The result will be a completely distinct object, with completely distinct referenced objects. JOS takes care of all of the details: superclass fields, following object graphs, and handling repeated references to the same object within the graph.

It will only work when the object being copied, as well as all of the other objects references directly or indirectly by the object, are serializable. (In other words, they must implement `java.io.Serializable`.) Fortunately it is often sufficient to simply declare that a given class implements `java.io.Serializable` and let Java's default serialization mechanisms do their thing. Java Object Serialization is slow, and using it to make a deep copy requires both serializing and deserializing.

There are ways to speed it up (e.g., by pre-computing serial version ids and defining custom `readObject()` and `writeObject()` methods), but this will usually be the primary bottleneck. The byte array stream implementations included in the `java.io` package are designed to be general enough to perform reasonable well for data of different sizes and to be safe to use in a multi-threaded environment. These characteristics, however, slow down `ByteArrayOutputStream` and (to a lesser extent) `ByteArrayInputStream`.

[http://www.jusfortechies.com/java/core-java/deepcopy\\_and\\_shallowcopy.php](http://www.jusfortechies.com/java/core-java/deepcopy_and_shallowcopy.php)

---

By convention, classes that implement this interface (`Cloneable`) should override

`Object.clone` which is protected) with a public method.

this interface does not contain the clone method.

Therefore, it is not possible to clone an object merely by virtue of the fact that it implements this interface. Even if the clone method is invoked reflectively, there is no guarantee that it will succeed.

<http://javarevisited.blogspot.in/2013/09/how-clone-method-works-in-java.html>

`CloneNotSupportedException`



Wrapper classes do not support clone

## 266) What is the disadvantage of deep cloning using serialization?

Disadvantages of using Serialization to achieve deep cloning –

Serialization is more expensive than using `object.clone()`.

Not all objects are serializable.

Serialization is not simple to implement for deep cloned object..

## 267) Cloning and inheritance?

- Cloning can be inserted at any level. A->b>c ; only c implements `cloneable`; a and b does not; it will work fine.
- `super.clone()` will call its `super.clone()` and the chain will continue until the call reaches the `clone()` method of the `Object` class, which will create a field by field mem copy of our object and return it back.

## 268) Copy constructor and alternative of clone?

### Copy Constructors

One option to provide copy functionality to your class instead of implementing Cloneable is to provide a copy constructor(s) instead. A copy constructor is like a regular constructor, which returns a new instance of the class. As an input, it has an object, which is supposed to be copied. Inside the body of the constructor, you implement your custom cloning logic.

```
public Person(Person personToCopy) {
    this.firstName = personToCopy.firstName;
    this.lastName = personToCopy.lastName;
    ...
}
```

This has several advantages. You don't need to implement any interface. You can accept any interface your class implements as an input and use it as a source of the clone instead (however, then this is rather a conversion constructor then, but can be useful under some circumstances). It does not force you to throw `CloneNotSupportedException` checked exception (and the caller is not forced to catch the exception).

### Static Factory Methods

This approach is similar to copy constructors and has similar advantages. The difference is that instead of a constructor, it utilizes a static method, which takes an object to be copied as an input and returns a copied instance.

```
public static Person deepCopyPerson(Person personToCopy) {
    Person copiedPerson = new Person();
    copiedPerson.firstName = personToCopy.firstName;
    copiedPerson.lastName = personToCopy.lastName;
    ...
    return copiedPerson;
}
```

I find it useful to include in the method's name whether it is a deep or a shallow copy, so it is immediately obvious from the code and it does not need to be documented separately. Depending on the needs you can choose whether you need a shallow or a deep copy or provide both. Unlike with copy constructors, you can decide whether to return an instance of the same class or rather any subclass.

### Serialization/Deserialization

Cloning and creating new instances through copy constructors and static factory methods are not the only ways to create a new instance of a class. A new instance is also created when deserializing a previously serialized object. Therefore, instead of cloning, you can serialize an object and then immediately deserialize it. That would result in a new instance created.

The good news is that there are already libraries supporting cloning using serialization/deserialization, such as [Apache Commons Serialization Utils](#). This makes it very easy to clone objects using `SerializationUtils.clone()`

```
public static T clone(T object)
```

The serialization approach has some advantages:

Simple alternative to cloning, especially when using library such as Apache Commons

Provides Deep Cloning

Suitable even for complex object graphs

Can be used on existing classes that currently provide just shallow copy

This also has some limitations and disadvantages:

All the classes in the object graph needs to implement Serializable

Transient fields are not cloned (Transient means not to be serialized)

Way more expensive than clone or copy constructors/factory methods

In some cases, this can be used, but you don't always have the luxury of having all the objects Serializable and with no transient fields. Especially when using third-party classes. Also, the performance hit is very significant. Depending on the amount of cloning required and performance requirements, it can be easily too much. More info about performance hit and the serialization approach, in general, can be found in [Java Tip 76: An alternative to the deep copy technique](#).

Reflection

Reflection is yet another way of making a copy of an object. Using reflection you can read all the fields of a class, then copy them and then assign them to a new instance even if the fields are not publicly accessible. There are various libraries and utilities providing this functionality.

If you are following JavaBeans convention and a shallow copy is enough for you (maybe all the fields are immutable/primitives?), you can use [Apache Commons BeanUtils.cloneBean\(\)](#), which provides this functionality. Alternatively, if you are using Spring, you can use similar util - [Spring BeanUtils.copyProperties\(\)](#). If you are looking for a deep copy, you cannot use BeanUtils. There are some other alternatives, such as [uk.com.robust-it.cloning.library](#) or [Kryo's deep cloning feature](#).

This approach can be used even if the classes don't implement the Serializable and therefore is well suited also for third-party libraries, where you cannot modify the code to support serialization or regular cloning. The performance is also much better than the regular serialization.

Conclusion

The process to make your class cloneable is the following:

Implement the Cloneable interface

Override the clone() method.

Call super.clone() if a shallow copy is sufficient.

Implement custom cloning logic, if a deep copy is required.

Alternatively, you can use various existing third-party libraries, which are usually based on serialization/deserialization or reflection approach.

Further Reading

[Josh Bloch on Design - Copy Constructor versus Cloning](#)

[Java Design Issues - A Conversation with Ken Arnold, Part VI - The clone Dilemma](#)

[Effective Java, 3rd Edition, Item 13: Override clone judiciously](#)

## 269) Why should we avoid cloning?

Josh Bloch also has [a much longer discussion](#): cloning should definitely be avoided. Here is an excellent summary paragraph on why he thinks cloning is a problem:

Object's clone method is very tricky. It's based on field copies, and it's "extra-linguistic." It creates an object without calling a constructor. There are no guarantees that it preserves the invariants established by the constructors. There have been lots of bugs over the years, both in and outside Sun, stemming from the fact that if you just call super.clone repeatedly up the chain until you have cloned an object, you have a shallow copy of the object. The clone generally shares state with the object being cloned. If that state is mutable, you don't have two independent objects. If you modify one, the other changes as well. And all of a sudden, you get random behavior.

**Bill Venners:** In your book you recommend using a copy constructor instead of implementing `Cloneable` and writing `clone`. Could you elaborate on that?

**Josh Bloch:** If you've read the item about cloning in my book, especially if you read between the lines, you will know that I think clone is deeply broken. There are a few design flaws, the biggest of which is that the `Cloneable` interface does not have a `clone` method. And that means it simply doesn't work: making something `Cloneable` doesn't say anything about what you can do with it. Instead, it says something about what it can do internally. It says that if by calling `super.clone` repeatedly it ends up calling `Object's clone` method, this method will return a field copy of the original.

But it doesn't say anything about what you can do with an object that implements the `Cloneable` interface, which means that you can't do a polymorphic `clone` operation. If I have an array of `Cloneable`, you would think that I could run down that array and clone every element to make a deep copy of the array, but I can't. You cannot cast something to `Cloneable` and call the `clone` method, because `Cloneable` doesn't have a public `clone` method and neither does `Object`. If you try to cast to `Cloneable` and call the `clone` method, the compiler will say you are trying to call the protected `clone` method on object.

The truth of the matter is that you don't provide any capability to your clients by implementing `Cloneable` and providing a public `clone` method other than the ability to copy. This is no better than what you get if you provide a copy operation with a different name and you don't implement `Cloneable`. That's basically what you're doing with a copy constructor. The copy constructor approach has several advantages, which I discuss in the book. One big advantage is that the copy can be made to have a different representation from the original. For example, you can copy a `LinkedList` into an `ArrayList`.

`Object's clone` method is very tricky. It's based on field copies, and it's "extra-linguistic." It creates an object without calling a constructor. There are no guarantees that it preserves the invariants established by the constructors. There have been lots of bugs over the years, both in and outside Sun, stemming from the fact that if you just call `super.clone` repeatedly up the chain until you have cloned an object, you have a shallow copy of the object. The clone generally shares state with the object being cloned. If that state is mutable, you don't have two independent objects. If you modify one, the other changes as well. And all of a sudden, you get random behavior. There are very few things for which I use `Cloneable` anymore. I often provide a public `clone` method on concrete classes because people expect it. I don't have abstract classes implement `Cloneable`, nor do I have interfaces extend it, because I won't place the burden of implementing `Cloneable` on all the classes that extend (or implement) the abstract class (or interface). It's a real burden, with few benefits.

Doug Lea goes even further. He told me that he doesn't use `clone` anymore except to copy arrays. You should use `clone` to copy arrays, because that's generally the fastest way to do it. But Doug's types simply don't implement `Cloneable` anymore. He's given up on it. And I think that's not unreasonable.

It's a shame that `Cloneable` is broken, but it happens. The original Java APIs were done very quickly under a tight deadline to meet a closing market window. The original Java team did an incredible job, but not all of the APIs are perfect. `Cloneable` is a weak spot, and I think people should be aware of its limitations.

## MARKER INTERFACE

### 270) Explain marker interface?

An interface without any methods is Marker Interface e.g. `Serializable`, `Cloneable`. Basically it is just to identify the special objects from normal objects.

Like in case of serialization, objects that need to be serialized must implement `Serializable` interface and down the line `writeObject()` method must be checking somewhere if it is an instance of `Serializable` or not.

### 271) Why this indication cannot be done using a flag inside a class?"

Yes this can be done by using a boolean flag or a String but doesn't marking a class like `Serializable` or `Cloneable` makes it more readable and it also allows to take advantage of Polymorphism in Java. Suppose you are writing a function which takes `Serializable` instance and it won't take any instance which does not implement marker interface.

Marker interfaces can be replaced with annotations in many places, however a marker interfaces can still be used for compile time checks.

You can have a method which must take an object of a class with a given marker interface(s) e.g.

```
public void myMethod(MyMarkerInterface MMI);
```

You cannot have this compile time check using an annotation alone.

<http://stackoverflow.com/questions/25850328/marker-interfaces-in-java>

Marker interface in Java is interfaces with no field or methods or in simple word empty interface in java is called marker interface. Example of market interface is Serializable, Cloneable and Remote interface. In Java is an empty interface and is used to signal to compiler or JVM that the objects of the class implementing this interface must be treated in a special way, like serializing, cloning, etc

- They used to indicate special processing and nothing to do with JVM.
- With annotation, annotation is better choice over marker.
- We can create our own marker, for that we need to create our own consumer , which via reflection/tool/anything checks the instance of type and pre process/perform respective operations.
- It is correct in the parts that (1) a marker interface must be empty, and (2) implementing it is meant to imply some special treatment of the implementing class. The part that is incorrect is that it implies that JVM or the compiler would treat the objects of that class differently: you are correct in observing that it is the code of Java class library that treats these objects as cloneable, serializable, etc. It has nothing to do with the compiler or the JVM.
- <https://stackoverflow.com/questions/25850328/marker-interfaces-in-java>

## WRAPPER

### 272) What are wrapper classes in java?

Java wrapper classes are the Object representation of eight primitive types in java. All the wrapper classes in java are immutable and final. Java 5 autoboxing and unboxing allows easy conversion between primitive types and their corresponding wrapper classes.

<http://javarevisited.blogspot.in/2012/07/auto-boxing-and-unboxing-in-java-be.html>

Pitfalls of auto-boxing in java ?

### 273) Methods of wrapper class?

Wrapper.valueOf()	To create wrapper object	Integer I = Integer.valueOf("10") Can take string and primitive
Wrapper.XXXValue	To get primitive	Integer i; Int x= i.intValue();
ParseXXX()	method to find primitive for the given String object.	double d = Double.parseDouble("10.5");
toString()	to convert Wrapper Object to String type	Integer I = new Integer(10); String s = I.toString();

JAVA INTERVIEW QUESTIONS GENERIC,  
INNERCLAS,ENUMSET,EMUMAP,QUEUE, DEQUEUE




**Utility methods of Wrapper classes:**The objective of Wrapper class is to define several utility methods which are required for the primitive types. There are 4 utility methods for primitive type which is defined by Wrapper class:

**valueOf() method :** We can use valueOf() method to create Wrapper object for given primitive or String. There are 3 types of valueOf() methods:

**Wrapper valueOf(String s) :** Every wrapper class except Character class contains a static valueOf() method to create Wrapper class object for given String.

**Syntax:**

**public static Wrapper valueOf(String s);**

```
// Java program to illustrate valueOf()

class GFG {
    public static void main(String[] args)
    {
        Integer I = Integer.valueOf("10");
        System.out.println(I);
        Double D = Double.valueOf("10.0");
        System.out.println(D);
        Boolean B = Boolean.valueOf("true");
        System.out.println(B);

        // Here we will get RuntimeException
        Integer I1 = Integer.valueOf("ten");
    }
}
```

Output:

10

10.0

true

Exception in thread "main" java.lang.NumberFormatException:  
For input string: "ten"

**Wrapper valueOf(String s, int radix) :** Every Integral Wrapper class Byte, Short, Integer, Long) contains the following valueOf() method to create a Wrapper object for the given String with specified radix. The range of the radix is 2 to 36.

**Syntax:**

**public static Wrapper valueOf(String s, int radix)**

```
// Java program to illustrate valueOf()

class GFG {
    public static void main(String[] args)
    {
        Integer I = Integer.valueOf("1111", 2);
        System.out.println(I);
        Integer I1 = Integer.valueOf("1111", 4);
        System.out.println(I1);
    }
}
```

Output:

15

85

**Wrapper valueOf(primitive p)** : Every Wrapper class including Character class contains the following method to create a Wrapper object for the given primitive type.

**Syntax:**

**public static Wrapper valueOf(primitive p);**

```
// Java program to illustrate valueOf()

class GFG {
    public static void main(String[] args)
    {
        Integer I = Integer.valueOf(10);
        Double D = Double.valueOf(10.5);
        Character C = Character.valueOf('a');
        System.out.println(I);
        System.out.println(D);
        System.out.println(C);
    }
}
```

Output:

10

10.5

a

**xxxValue() method:** We can use xxxValue() methods to get the primitive for the given Wrapper Object. Every number type Wrapper class( Byte, Short, Integer, Long, Float, Double) contains the following 6 methods to get primitive for the given Wrapper object:

public byte byteValue()

public short shortValue()

```
public int intValue()  
public long longValue()  
public float floatValue()  
public double doubleValue()
```

```
// Java program to illustrate byteValue()  
  
class GFG {  
    public static void main(String[] args)  
    {  
        Integer I = new Integer(130);  
        System.out.println(I.byteValue());  
        System.out.println(I.shortValue());  
        System.out.println(I.intValue());  
        System.out.println(I.longValue());  
        System.out.println(I.floatValue());  
        System.out.println(I.doubleValue());  
    }  
}
```

Output:

```
-126  
130  
130  
130  
130.0  
130.0
```

**parseXxx() method :** We can use parseXxx() methods to convert String to primitive. There are two types of parseXxx() methods:

**primitive parseXxx(String s) :** Every Wrapper class except character class contains the following parseXxx() method to find primitive for the given String object.

**Syntax:**

```
public static primitive parseXxx(String s);
```

```
// Java program to illustrate parseXxx()

class GFG {
    public static void main(String[] args)
    {
        int i = Integer.parseInt("10");
        double d = Double.parseDouble("10.5");
        boolean b = Boolean.parseBoolean("true");
        System.out.println(i);
        System.out.println(d);
        System.out.println(b);
    }
}
```

Output:

10

10.5

true

**parseXxx(String s, int radix) :** Every Integral type Wrapper class (Byte, Short, Integer, Long) contains the following parseXxx() method to convert specified radix String to primitive.

**Syntax:**

**public static primitive parseXxx(String s, int radix);**

```
// Java program to illustrate parseXxx()

class GFG {
    public static void main(String[] args)
    {
        int i = Integer.parseInt("1000", 2);
        long l = Long.parseLong("1111", 4);
        System.out.println(i);
        System.out.println(l);
    }
}
```

Output:

8

85

**toString() method:** We can use toString() method to convert Wrapper object or primitive to String. There are few forms of toString() method:

**public String toString() :** Every wrapper class contains the following toString() method to convert Wrapper Object to String type.

**Syntax:**

**public String toString();**

```
// Java program to illustrate toString()

class GFG {
    public static void main(String[] args)
    {
        Integer I = new Integer(10);
        String s = I.toString();
        System.out.println(s);
    }
}
```

Output:

10

**toString(primitive p)** : Every Wrapper class including Character class contains the following static toString() method to convert primitive to String.

**Syntax:**

**public static String toString(primitive p);**

```
// Java program to illustrate toString()

class GFG {
    public static void main(String[] args)
    {
        String s = Integer.toString(10);
        System.out.println(s);
        String s1 = Character.toString('a');
        System.out.println(s1);
    }
}
```

Output:

10

a

**toString(primitive p, int radix)** : Integer and Long classes contains the following toString() method to convert primitive to specified radix String.

**Syntax:**

**public static String toString(primitive p, int radix);**

```
// Java program to illustrate toString()

class GFG {
    public static void main(String[] args)
    {
        String s = Integer.toString(15, 2);
        System.out.println(s);
        String s1 = Long.toString(11110000, 4);
        System.out.println(s1);
    }
}
```

Output:

1111

222120121300

## SERILAZATION

### 274) Explain Serialization in detail?

Links-

<http://www.oracle.com/technetwork/articles/java/javaserial-1536170.html> ,  
<https://docs.oracle.com/javase/7/docs/platform/serialization/spec/version.html#6678>,  
<https://www.ibm.com/developerworks/library/j-5things1/index.html> ,  
<http://thecodersbreakfast.net/index.php?post/2011/05/12/Serialization-and-magic-methods> ,  
<http://www.geeksforgeeks.org/object-serialization-inheritance>  
<http://www.javapractices.com/topic/TopicAction.do?Id=45> ,  
<http://javarevisited.blogspot.com/2011/04/top-10-java-serialization-interview.html>,

- Serialization in Java allows us to convert an Object to stream that we can send over the network or save it as file or store in DB for later usage. Deserialization is the process of converting Object stream to actual Java Object to be used in our program.
- Serialization in java is implemented by ObjectOutputStream and ObjectInputStream and it is marker interface.
- It's okay for a class to implement Serializable even if its superclass doesn't.
- However, when you deserialize such an object, the non-serializable superclass must run its constructor. Remember, constructors don't run on deserialized classes that implement Serializable.
- Sample code:

```
try {
    FileOutputStream fos = new FileOutputStream("play.txt");
    ObjectOutputStream os = new ObjectOutputStream(fos);
    os.writeObject(c1);
    os.close();
}
```

## JAVA INTERVIEW QUESTIONS GENERIC, INNERCLAS,ENUMSET,EMUMAP,QUEUE, DEQUEUE

```
FileInputStream fis = new FileInputStream("play.txt");
ObjectInputStream is = new ObjectInputStream(fis);
CardPlayer c2 = (CardPlayer) is.readObject();
is.close();
} catch (Exception x ) { }
```

- If a class that implements serializable, contains object of another non serializable class then it will not serialize it and throw run time exception in while serializing.
- `static` and `transient` variables are not serialized when an object is serialized. Transient becomes to default value that is 0 or null. Static will remain as is.
- Sequence of writing and reading object variable should be same as data is written in bytes format until we are reading/writing complete Object.
- If you don't add `serialVersionUID` , any change in class throws exception **invalid class exception**. The reason is clear that `serialVersionUID` of the previous class and new class are different. Actually if the class doesn't define `serialVersionUID`, it's getting calculated automatically and assigned to the class. Java uses class variables, methods, class name, package etc to generate this unique long number. If class have `serialVersionUID` then -Some of the changes in class that will not affect the deserialization process are
  - a. Adding new variables to the class
  - b. Changing the variables from transient to non-transient, for serialization it's like having a new field.
  - c. Changing the variable from static to non-static, for serialization it's like having a new field.

```
private static final long serialVersionUID = 7526472295622776147L;
```

- We can use serialver Utility to generate unique `serialVersionUID`

```
SerializationExample/bin$serialver -classpath .
com.journaldev.serialization.Employee
```

- For maintain security , we can define `writeObject` and `readObject` in serializable class where we can encrypt the field data.
- Compatible changes: A compatible change is one that can be made to a new version of the class, which still keeps the stream compatible with older versions of the class. Examples of compatible changes are:
  - d. Addition of new fields or classes does not affect serialization, as any new data in the stream is simply ignored by older versions. When the instance of an older version of the class is deserialized, the newly added field will be set to its default value.

- e. You can field change access modifiers like private, public, protected or package as they are not reflected to the serial stream.
- f. You can change a transient or static field to a non-transient or non-static field, as it is similar to adding a field.
- g. You can change the access modifiers for constructors and methods of the class. For instance a previously private method can now be made public, an instance method can be changed to static, etc. The only exception is that you cannot change the default signatures for readObject() and writeObject() if you are implementing custom serialization. The serialization process looks at only instance data, and not the methods of a class.

- Incompatible changes:

- Once a class implements the Serializable interface, you cannot later make it implement the Externalizable interface, since this will result in the creation of an incompatible stream.
- Deleting fields can cause a problem. Now, when the object is serialized, an earlier version of the class would set the old field to its default value since nothing was available within the stream. Consequently, this default data may lead the newly created object to assume an invalid state.
- Changing a non-static into static or non-transient into transient is not permitted as it is equivalent to deleting fields.
- You also cannot change the field types within a class, as this would cause a failure when attempting to read in the original field into the new field.
- You cannot alter the position of the class in the class hierarchy. Since the fully-qualified class name is written as part of the bytestream, this change will result in the creation of an incompatible stream.
- You cannot change the name of the class or the package it belongs to, as that information is written to the stream during serialization.

2) Methods in serialization:

ObjectOutputStr eam		<b>Public final void</b> writeObject (ObjectOutputStream out) <b>throws</b> IOException	Write the specified object to the ObjectOutputs tream. The class of the * object, the signature of the class, and the values of the non-transient * and non-static fields of the class and all of its supertypes are
------------------------	--	--	---



# JAVA INTERVIEW QUESTIONS GENERIC, INNERCLAS,ENUMSET,EMUMAP,QUEUE, DEQUEE

		<p>* written. Default serialization for a class can be overridden using the</p> <p>* writeObject and the readObject methods. Objects referenced by this</p> <p>* object are written transitively so that a complete equivalent graph of</p> <p>* objects can be reconstructed by an ObjectInputStream.</p>
	<pre>public void defaultWriteObject() throws IOException</pre>	<p>Write the non-static and non- transient fields of the current class to this stream. This may only be called from the writeObject method of the class being serialized. It will throw the NotActiveExce ption if it is</p> <p>called otherwise</p>
	<pre>ANY-ACCESS- MODIFIER Object writeReplace() {  throws ObjectStreamExce ption;</pre>	<p>For Serializable and Externalizable classes, the writeReplace method allows a class of an object to nominate its own replacement in the stream before the object is written. By implementing the writeReplace method, a class can directly control the types and instances of its own instances being serialized. The writeReplace method is called when ObjectOutputStream is preparing to write the object to the stream. The ObjectOutputStream check s whether the class defines the writeReplace method. If</p>

		the method is defined, the <code>writeReplace</code> method is called to allow the object to designate its replacement in the stream. The object returned should be either of the same type as the object passed in or an object that when read and resolved will result in an object of a type that is compatible with all references to the object. If it is not, a <code>ClassCastException</code> will occur when the type mismatch is discovered.

Sequence if all methods are defined – `writeReplace`->`writeObject`->`readObject`->`readResolve`->`validateObject`

## 275) Serializable And inheritance:

**Case 1: If superclass is serializable then subclass is automatically serializable:** If superclass is `Serializable`, then by default every subclass is serializable. Hence, even though subclass doesn't implement `Serializable` interface (and if it's superclass implements `Serializable`), then we can serialize subclass object.

**If a superclass is not serializable then subclass can still be serialized:** Even though superclass doesn't implement `Serializable` interface, we can serialize subclass object if subclass itself implements `Serializable` interface. So, we can say that to serialize subclass object, superclass need not to be serializable. But what happens with the instances of superclass during serialization in this case. The following procedure explain this.

**What happens when a class is serializable but its superclass is not?**

**Serialization:** At the time of serialization, if any instance variable is inheriting from non-serializable superclass, then JVM ignores original value of that instance variable and save default value to the file.

**De- Serialization:** At the time of de-serialization, if any non-serializable superclass is present, then JVM will execute instance control flow in the superclass. To execute instance control flow in a class, JVM will always invoke default(no-arg) constructor of that class. So, every non-serializable superclass must necessarily contain default constructor, otherwise we will get runtime-exception.

## 276) If the superclass is serializable but we don't want the subclass to be serialized:

There is no direct way to prevent subclass from serialization in java. One possible way by which a programmer can achieve this is by implementing the `writeObject()` and `readObject()` methods in the subclass and needs to throw `NotSerializableException` from these methods. These methods are executed during serialization and de-serialization respectively. By overriding these methods, we are just implementing our own custom serialization.

## 277) defaultWriteObject and defaultReadObject ??

- This is also known as custom serialization.
- `defaultWriteObject()` is identical to default Serialization without `writeObject()`.
- When we override `writeObject` and `readObject`, and we need little tweak and let Java do rest of the serialization, call `defaultReadObject` and `defaultWriteObject` first from the `writeObject` and `ReadObject` methods.
- When there is only a little tweak in the standard Java serialization process that you want to do manually, for everything else, you need the default process to kick in. In this case, the call to `defaultWriteObject()` is the first thing in the program.
- When you simply do not want Java to come in and help during serialization. The entire serialization would now be done manually. I think this case comes closest to the "Externalizable" functionality.
- If all instance fields are transient, it is technically permissible to dispense with invoking `defaultWriteObject` and `defaultReadObject` , but it is not recommended. Even if all instance fields are transient, invoking `defaultWriteObject` affects the serialized form, resulting in greatly enhanced flexibility. The resulting serialized form makes it possible to add nontransient instance fields in a later release while preserving backward and forward compatibility. If an instance is serialized in a later version and deserialized in an earlier version, the added fields will be ignored. Had the earlier version's `readObject` method failed to invoke `defaultReadObject` , the deserialization would fail with a `StreamCorruptedException` .
- In default mechanism static field and transient variable are not serialized or deserialized. As an example if we want to serialize transient variable we need to use `readObject` and `writeObject`.

Default serialization example:

```
// Java code for serialization and deserialization
```

// of a Java object

```
import java.io.*;
```

```
class Demo implements java.io.Serializable
```

```
{
```

```
    public int a;
```

```
    public String b;
```

```
    // Default constructor
```

```
    public Demo(int a, String b)
```

```
    {
```

```
        this.a = a;
```

```
        this.b = b;
```

```
    }
```

```
}
```

```
class Test
```

```
{
```

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

```
    {
```

```
        Demo object = new Demo(1, "geeksforgeeks");
```

```
        String filename = "file.ser";
```

```
        // Serialization
```

```
        try
```

```
        {
```

```
            //Saving of object in a file
```

```
            FileOutputStream file = new  
FileOutputStream(filename);
```

```
        ObjectOutputStream out = new
ObjectOutputStream(file);

        // Method for serialization of object
        out.writeObject(object);

        out.close();
        file.close();

        System.out.println("Object has been serialized");

    }

    catch(IOException ex)
    {
        System.out.println("IOException is caught");
    }

    Demo object1 = null;

    // Deserialization
    try
    {
        // Reading the object from a file
        FileInputStream file = new FileInputStream(filename);
        ObjectInputStream in = new ObjectInputStream(file);

        // Method for deserialization of object
        object1 = (Demo)in.readObject();
```

```

        in.close();
        file.close();

        System.out.println("Object has been deserialized ");
        System.out.println("a = " + object1.a);
        System.out.println("b = " + object1.b);
    }

    catch(IOException ex)
    {
        System.out.println("IOException is caught");
    }

    catch(ClassNotFoundException ex)
    {
        System.out.println("ClassNotFoundException is
caught");
    }

}
}

```

```

1. package com.concretepage;
2.
3. import java.io.IOException;
4. import java.io.ObjectInputStream;
5. import java.io.ObjectOutputStream;
6. import java.io.Serializable;
7.
8. public class ConcretePage implements Serializable {
9.     public static final long serialVersionUID = 1L;
10.    private String user;
11.    private transient String author;
12.
13.    public ConcretePage(String user,String author){
14.        this.user=user;
15.        this.author=author;
16.    }
17.    private void writeObject(ObjectOutputStream out) throws IOException {

```

```
18.         out.defaultWriteObject();
19.         out.writeObject(this.author);
20.     }
21.
22.     private void readObject(ObjectInputStream in) throws
IOException,ClassNotFoundException {
23.         in.defaultReadObject();
24.         this.author = (String)in.readObject();
25.     }
26.
27.     public String getUser() {
28.         return user;
29.     }
30.     public void setUser(String user) {
31.         this.user = user;
32.     }
33.     public String getAuthor() {
34.         return author;
35.     }
36.     public void setAuthor(String author) {
37.         this.author = author;
38.     }
39. }
```

278) What if a serializable class contains an reference to no serializable class?

- Either make them transient
- Override writeObject and readObject in serializable class and write the code for non serializable attribute.
- <https://www.javaworld.com/article/2097430/java-se/serializing-java-objects-with-non-serializable-attributes.html>

279) Serialization methods:

During serialization

`writeObject`

`private void writeObject (ObjectOutputStream out)` throws `IOException`

This method allows to take complete control over what will be sent over the wire.

In most cases, you will just call `out.defaultWriteObject()` to benefit from the default serialization process, then add some more data of your choice (for instance, data from the parent class) by calling `out.writeDouble`, `out.writeUTF`, etc. (inherited by `ObjectOutputStream` from the `DataOutput` interface).

`writeReplace`

`private Object writeReplace()` throws `ObjectStreamException`

This method allows the developer to provide a replacement object that will be serialized instead of the original one.

During de-serialization

`readObject`

`private void readObject(java.io.ObjectInputStream in)` throws `IOException`, `ClassNotFoundException`

This method is the same as `writeObject` above, but for reading objects from the serialized stream.

You can call `in.defaultReadObject()` to automatically read most of the data, then manually read back the extra data you may have added. Be careful to read data in the same exact order they were written in the stream !

This method is also where you can declare stream validators.

`validateObject`

`public void validateObject()` throws `InvalidObjectException`

If the serialized object implements `ObjectInputValidation`, you may register it as a stream validator.

Useful to verify the stream has not been tampered with, or that the data makes sense before handing it back to your application.

`readResolve`

`private Object readResolve()` throws `ObjectStreamException`

This method mirrors `writeReplace` : it may be used to replace the de-serialized object by another one of your choice.

The serialization / de-serialization pipeline

Now that you know what magic methods exist and their typical use, let's see in what order they are called during a serialization / de-serialization roundtrip.

```
public class Pojo implements Serializable, ObjectInputValidation {
```



## JAVA INTERVIEW QUESTIONS GENERIC, INNERCLAS,ENUMSET,EMUMAP,QUEUE, DEQUEE

```
private String msg;

public Pojo(String msg) {
    this.msg = msg;
}

public String getMsg() {
    return msg;
}

private void writeObject(java.io.ObjectOutputStream out) throws IOException {
    System.out.println("writeObject");
    out.defaultWriteObject();
}

private Object writeReplace() throws ObjectOutputStreamException {
    System.out.println("writeReplace");
    return this;
}

private void readObject(java.io.ObjectInputStream in) throws IOException,
ClassNotFoundException {
    System.out.println("readObject");
    in.registerValidation(this, 0);
    in.defaultReadObject();
}

@Override
public void validateObject() throws InvalidObjectException {
    System.out.println("validateObject");
}

private Object readResolve() throws ObjectOutputStreamException {
    System.out.println("readResolve");
    return this;
}
}

public class Test {

    public static void main(String[] args) throws Exception {
        Pojo pojo = new Pojo("Hello world");
        byte[] bytes = serialize(pojo); // Serialization
        Pojo p = (Pojo) deserialize(bytes); // De-serialization
        System.out.println(p.getMsg());
    }

    private static byte[] serialize(Object o) throws IOException {
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(baos);
        oos.writeObject(o);
        oos.flush();
        oos.close();
        return baos.toByteArray();
    }

    private static Object deserialize(byte[] bytes) throws ClassNotFoundException, IOException {
        ByteArrayInputStream bais = new ByteArrayInputStream(bytes);
        ObjectInputStream ois = new ObjectInputStream(bais);
        Object o = ois.readObject();
        ois.close();
        return o;
    }
}
```

When this test program is executed, it prints the following :

```
writeReplace  
writeObject  
readObject  
readResolve  
validateObject  
Hello world  
Interesting facts :
```

The writeReplace method is executed first. The rest of the serialization process will be applied to the replacement object, if any.  
As expected, the validation method is executed on the replacement object, not on the one that was originally de-serialized - this one will be silently discarded.

## 280) Difference between serialization and externalization--

- **Marker interface:** Serializable is marker interface without any methods. Externalizable interface contains two methods: writeExternal() and readExternal().
- **Serialization process:** Default Serialization process will be kicked-in for classes implementing Serializable interface. Programmer defined Serialization process will be kicked-in for classes implementing Externalizable interface.
- **Maintenance:** [Incompatible changes](#) may break serialisation.
- **Backward Compatibility and Control:** If you have to support multiple versions, you can have full control with Externalizable interface. You can support different versions of your object. If you implement Externalizable, it's your responsibility to serialize super class
- **public No-arg constructor:** Serializable uses reflection to construct object and does not require no arg constructor. But Externalizable demands public no-arg constructor.
- When an Externalizable object is reconstructed, an instance is created first using the public no-argument constructor, then the readExternal method is called. So, it is mandatory to provide a no-argument constructor.  
When an object implements Serializable interface, is serialized or deserialized, no constructor of object is called and hence any initialization which is implemented in constructor can't be done.

2) Second difference between Serializable vs Externalizable is responsibility of Serialization. when a class implements Serializable interface, default Serialization process gets kicked off and that takes responsibility of serializing super class state.

When any class in Java implements java.io.Externalizable then its your responsibility to implement Serialization process i.e. preserving all important information.

3) This difference between Serializable and Externalizable is performance.

You can not do much to improve performance of default serialization process except reducing number of fields to be serialized by using transient and static keyword but with Externalizable interface you have full control over Serialization process.

4) Another important difference between Serializable and Externalizable interface is maintenance.

When your Java class implements Serializable interface its tied with default representation which is fragile and easily breakable if structure of class changes e.g. adding or removing field. By using java.io.Externalizable interface you can create your own custom binary format for your object.

// Java program to demonstrate working of Externalization interface

```
import java.io.*;

class Car implements Externalizable {
    static int age;
    String name;
    int year;

    public Car()
    {
        System.out.println("Default Constructor called");
    }

    Car(String n, int y)
    {
```

```
        this.name = n;
        this.year = y;
        age = 10;
    }

    public void writeExternal(ObjectOutput out)
                                   throws IOException
    {
        out.writeObject(name);
        out.writeInt(age);
        out.writeInt(year);
    }

    public void readExternal(ObjectInput in)
    throws IOException, ClassNotFoundException
    {
        name = (String)in.readObject();
        year = in.readInt();
        age = in.readInt();
    }

    public String toString()
    {
        return ("Name: " + name + "\n" +
                "Year: " + year + "\n" +
                "Age: " + age);
    }
}

public class ExternExample {
    public static void main(String[] args)
    {
        Car car = new Car("Shubham", 1995);
    }
}
```

```
Car newcar = null;

// Serialize the car
try {
    FileOutputStream fo = new FileOutputStream("gfg.txt");
    ObjectOutputStream so = new ObjectOutputStream(fo);
    so.writeObject(car);
    so.flush();
}
catch (Exception e) {
    System.out.println(e);
}

// Deserializa the car
try {
    FileInputStream fi = new FileInputStream("gfg.txt");
    ObjectInputStream si = new ObjectInputStream(fi);
    newcar = (Car)si.readObject();
}
catch (Exception e) {
    System.out.println(e);
}

System.out.println("The original car is:\n" + car);
System.out.println("The new car is:\n" + newcar);
}
}
```

## 281) Explain RTTI in java?

Ways to implement RTTI in java:

1. Interface refrence
2. Class.forName()
3. .class literal

It's interesting to note that creating a reference to a Class object using ".class"

doesn't automatically initialize the Class object.

There are actually three steps in preparing a class for use:

- 1.Loading, which is performed by the class loader.

This finds the bytecodes (usually, but not necessarily, on your disk in your classpath)

and creates a Class object from those bytecodes.

- 2.Linking. The link phase verifies the bytecodes in the class, allocates storage for static fields,

and if necessary, resolves all references to other classes made by this class.

- 3.Initialization. If there's a superclass, initialize that.

Execute static initializers and static initialization blocks.

Initialization is delayed until the first reference to a static method (the constructor is implicitly static)

or to a non-constant static field

If a static final value is a "compile-time constant," that value can be read without causing the class to be initialized.

Making a field static and final, however, does not guarantee this behavior: accessing it forces class initialization because it cannot be a compile-time constant.

If a static field is not final, accessing it always requires linking (to allocate storage for the field)

and initialization (to initialize that storage) before it can be read.

-----Generic class references

The ordinary class reference does not produce a warning. However, you can see that the ordinary class reference can be reassigned to any other Class object,

whereas the generic class reference can only be assigned to its declared type. By using the generic syntax, you allow the compiler to enforce extra type checking.

What if you'd like to loosen the constraint a little? Initially, it seems like you ought to be able to do something like:

```
Class<Number> genericNumberClass = int.class;
```

This would seem to make sense because Integer is inherited from Number.

But this doesn't work, because the Integer Class object is not a subclass of the Number Class object

To loosen the constraints when using generic Class references, I employ the wildcard, which is part of Java generics.

The wildcard symbol is '?', and it indicates "anything."

```
Class x = x.class;
```

and

```
class<?> x = x.class;
```

are same thing

```
class<? extends Number> num = int.class/double.class/Number.class
```

The reason for adding the generic syntax to Class references is only to provide compile-time type checking,

so that if you do something wrong you find out about it a little sooner.

-----casting

Up-casting is casting to a supertype, while downcasting is casting to a subtype.

Supercasting is always allowed, but subcasting involves a type check and can throw a `ClassCastException`

There's a third form of RTTI in Java. This is the keyword `instanceof`, which tells you if an object is an instance of a particular type.

It returns a boolean so you use it in the form of a question, like this:

dynamic instance of --- `Class.isInstance()`

```
if(x instanceof Dog)
```

```
((Dog)x).bark();
```



-----Reflection

## JAVA 7

### 282) Features of java 7:

- 1) String in switch
- 2) Try with resource

## JAVA 8

### 283) Features of java 8

Answer:

- Interface changes with default and static methods
- Functional interfaces and Lambda Expressions
- Java Stream API for collection classes
- Java Date Time API

## THREAD

112) What are the differences between processes and threads?

	Process	Thread
Definition	An executing instance of a program is called a process.	A thread is a subset of a process.
Communication	Processes must use inter-process communication to	Threads can directly communicate with other threads of its process.

	communicate with sibling processes.	
<b>Control</b>	Processes can only exercise control over child processes.	Threads can exercise considerable control over threads of the same process.
<b>Changes</b>	Any change in the parent process does not affect child processes.	Any change in the main thread may affect the behavior of the other threads of the process.
<b>Memory</b>	Run in separate memory spaces.	Run in shared memory spaces.
<b>Controlled by</b>	Process is controlled by the operating system.	Threads are controlled by the programmer in a process.
<b>Dependence</b>	Processes are independent.	Threads are dependent.

#### 284) What is synchronization?

Synchronization refers to multi-threading. A synchronized block of code can be executed by only one thread at a time. As Java supports execution of multiple threads, two or more threads may access the same fields or objects. Synchronization is a process which keeps all concurrent threads in execution to be in sync. Synchronization avoids memory consistency errors caused due to inconsistent view of shared memory. When a method is declared as synchronized the thread holds the monitor for that method's object. If another thread is executing the synchronized method the thread is blocked until that thread releases the monitor.

#### 285) Can throw some light on Yielding and Sleeping?

Answer: When any task invokes its yield() method, it will return to the ready state. Whenever a task invokes sleep() method, it will return to the wait state.

#### 286) 12. Explain different ways of creating a thread. Which one would you prefer and why ?

There are three ways that can be used in order for a Thread to be created:

A class may extend the Thread class.

A class may implement the Runnable interface.

An application can use the Executor framework, in order to create a thread pool.

The Runnable interface is preferred, as it does not require an object to inherit the Thread class. In case your application design requires multiple inheritance, only interfaces can help you. Also, the thread pool is very efficient and can be implemented and used very easily.

### 287) 13. Explain the available thread states in a high-level.

During its execution, a thread can reside in one of the following states:

Runnable: A thread becomes ready to run, but does not necessarily start running immediately.

Running: The processor is actively executing the thread code.

Waiting: A thread is in a blocked state waiting for some external processing to finish.

Sleeping: The thread is forced to sleep.

Blocked on I/O: Waiting for an I/O operation to complete.

Blocked on Synchronization: Waiting to acquire a lock.

Dead: The thread has finished its execution.

### 288) 14. What is the difference between a synchronized method and a synchronized block ?

In Java programming, each object has a lock. A thread can acquire the lock for an object by using the synchronized keyword. The synchronized keyword can be applied in a method level (coarse grained lock) or block level of code (fine grained lock).

15. How does thread synchronization occurs inside a monitor ? What levels of synchronization can you apply ?

The JVM uses locks in conjunction with monitors. A monitor is basically a guardian that watches over a sequence of synchronized code and ensuring that only one thread at a time executes a synchronized piece of code. Each monitor is associated with an object reference. The thread is not allowed to execute the code until it obtains the lock.

289) 16. What's a deadlock ?

A condition that occurs when two processes are waiting for each other to complete, before proceeding. The result is that both processes wait endlessly.

290) 17. How do you ensure that N threads can access N resources without deadlock ?

A very simple way to avoid deadlock while using N threads is to impose an ordering on the locks and force each thread to follow that ordering. Thus, if all threads lock and unlock the mutexes in the same order, no deadlocks can arise.

## JDBC

291) 72. What is JDBC ?

JDBC is an abstraction layer that allows users to choose between databases. JDBC enables developers to write database applications in Java, without having to concern themselves with the underlying details of a particular database.

292) 73. Explain the role of Driver in JDBC.

The JDBC Driver provides vendor-specific implementations of the abstract classes provided by the JDBC API. Each driver must provide implementations for the following classes of the java.sql package: Connection, Statement, PreparedStatement, CallableStatement, ResultSet and Driver.

293) 74. What is the purpose Class.forName method ?

This method is used to load the driver that will establish a connection to the database.

294) 75. What is the advantage of PreparedStatement over Statement ?

PreparedStatement are precompiled and thus, their performance is much better. Also, PreparedStatement objects can be reused with different input values to their queries.

295) 76. What is the use of CallableStatement ?

Name the method, which is used to prepare a CallableStatement. A CallableStatement is used to execute stored procedures. Stored procedures

are stored and offered by a database. Stored procedures may take input values from the user and may return a result. The usage of stored procedures is highly encouraged, because it offers security and modularity. The method that prepares a CallableStatement is the following:

```
CallableStatement.prepareCall();
```

## 296) 77. What does Connection pooling mean ?

The interaction with a database can be costly, regarding the opening and closing of database connections. Especially, when the number of database clients increases, this cost is very high and a large number of resources is consumed. A pool of database connections is obtained at start up by the application server and is maintained in a pool. A request for a connection is served by a connection residing in the pool. In the end of the connection, the request is returned to the pool and can be used to satisfy future requests.

## 297) Different type of statements and their use?

## 298) What is the return type for class.forName()

<https://snowdream.github.io/115-Java-Interview-Questions-and-Answers/115-Java-Interview-Questions-and-Answers/en/jdbc.html>

```
class test = Class.forName("className");
```

```
className obj = (className)test.newInstance();
```

newInstance() is a way to implement virtual constructor.

It always invokes the default constructor of the class.

But if we have to invoke some other constructor then we have to do something like this :

```
Constructor c = Class.forName("Foo").getConstructor(String.class,  
Integer.TYPE);
```

```
Foo foo = (Foo) c.newInstance("example", 34);
```

<http://javarevisited.blogspot.in/2012/12/how-classloader-works-in-java.html>

<http://www.allinterview.com/showanswers/11016/exactly-happens-we-execute-class-forname-driver-class-name-explain-indetail.html>

## IMPORTANT POINTS

- 1) A *class* cannot *extend* more than one class. That means one parent per class. A class *can* have multiple ancestors, however, since class B could extend class A, and class C could extend class B, and so on. So any given class might have multiple classes up its inheritance tree, but that's not the same as saying a class directly extends two classes.
- 2) Inner class can have constructor?
- 3) Narrowing a value, byte b = 128, since byte range is 127, java starts truncating the leftmost bit and flip the value add 0, causing it to print value in negative range.
- 4) How free memory is represented actually in RAM, because in RAM every thing is 1 or 0, so how free memory is represented ??? -- >> <https://www.quora.com/How-does-a-flash-drive-pen-drive-store-data>
- 5) Instance variable references are always given a default value of null, until explicitly initialized to something else. But local references are not given a default value; in other words, *they aren't null*

```
import java.util.Date;
public class TimeTravel {
    public static void main(String [] args) {
        Date date;
        if (date == null)
            System.out.println("date is null");
        }
    }
}
ERROR - date is not initialized.
```

- 6) Default values – int 0, Boolean false, obj null, char '\u0000', float -0.0
- 7) *any time we make any changes at all to a String, the VM will update the reference variable to refer to a different object.* The different object might be a new object, or it might not, but it will definitely be a different object. The reason we can't say for sure whether a new object is created is because of the String constant pool.

Anonymous array - f.takesAnArray(new int[] {7,7,8,2,5});

it is legal to test whether the null reference is an instance of a class.

This will always result in false, of course. For example:

```
class InstanceTest {
    public static void main(String [] args) {
        String a = null;
        boolean b = null instanceof String;
        boolean c = a instanceof String;
        System.out.println(b + " " + c);
    }
}
prints this: false false
```

## JAVA INTERVIEW QUESTIONS GENERIC, INNERCLAS,ENUMSET,EMUMAP,QUEUE, DEQUEUE

*If either operand is a String, the + operator becomes a String concatenation operator. If both operands are numbers, the + operator is the addition operator*

- 8) printStackTrace() method prints the most recently entered method first and continues down, printing the name of each method as it works its way down the call stack (this is called unwinding the stack) 9from the top.
- 9) When an Error or a subclass of Error is thrown, it's unchecked. You are not required to catch Error objects or Error subtypes
- 10) Java Echo args --- The first word in the command, "java," indicates that the Java virtual machine from Sun's Java 2 SDK should be run by the operating system.
- 11) When jvm load class , it creates only ONE class object of that loaded type that can be accessed by getClass()
- 12) JVM – abstract specification , concrete implementation , running instance.
- 13) If you start three Java applications at the same time, on the same computer, using the same concrete implementation, you'll get three Java virtual machine instances. Each Java application runs inside its own Java virtual machine.
- 14) Method Area - All threads share the same method area , it need not to be contiguous , The size of method area is not fixed it can expand and contract. The method area can also be garbage collected. It is optional and depends upon jvm and need.
- 15) Java application runs inside its "own" exclusive Java virtual machine instance, there is a separate heap for every individual running application. There is no way two different Java applications could trample on each other's heap data. If you a main program 3 times using dos/ide that runs forever , 3 jvm instance will be created each having own heap space/memory area.
- 16) No garbage collection technique is dictated by the Java virtual machine specification. Designers can use whatever techniques seem most appropriate .
- 17) 'heap area: heap need not be contiguous, and may be expanded and contracted as the running program progresses.
- 18) Local variable can be only final
- 19) Local variable has to be initialize before use.
- 20) We can have class /instance variable and local variable with the same name , that's called shadowing.
- 21) For final primitives, this means that once the variable is assigned a value, the value can't be altered. For example, if you assign 10 to the int variable x, then x is going to stay 10, forever. So that's straightforward for primitives, but what does it mean to have a final object reference variable? A reference variable marked final can't ever be reassigned to refer to a different object. The data within the object can be modified, but the reference variable cannot be changed.
- 22) Final /private/static method cant be overridden.
- 23) Boolean size is JVM dependent.
- 24) Char is 16 bit unsigned, more than shorts.
- 25) Byte 8 bit , short 16 bit , int 32 bit , long 64 bit. All number are signed , first bit is for sign , 1 negative, 0 positive and range is  $-2^{bits}$  to  $2^{bits} - 1$
- 26) Float 32 and double 64 , we can not determine range of floating variable.
  - Constructor, class(except nested class), local variable, interface, inner class method and variable, method local inner class cant be marked as static.
  - Enum can be declared inside the class and outside the class as class itself in both ways.
  - Enum cant be declared inside methods. And Enum end with semicolon but that is optional.
  - Polymorphic method invocations apply only to instance methods. You can
    - always refer to an object with a more general reference variable type (a superclass or interface), but at runtime, the ONLY things that are dynamically selected based on the actual object (rather than the reference type) are instance methods. Not static methods. Not variables. Only overridden instance methods are dynamically invoked based on the real object's type.
    - you cannot invoke an instance (in other words, nonstatic) method (or access an instance variable) until after the super constructor has run.
- Unlike downcasting, upcasting (casting *up* the inheritance tree to a more general type) works implicitly (i.e., you don't have to type in the cast) because when you

## JAVA INTERVIEW QUESTIONS GENERIC, INNERCLAS,ENUMSET,EMUMAP,QUEUE, DEQUEUE

- upcast you're implicitly restricting the number o methods you can invoke, as
- opposed to *downcasting*

- 27) Every constructor has first call to super or this.
- 28) Private base constructor is not visible to child hence Base constructor should not be private, if Child is trying to create new Object, it will give error.
- 29) Upcasting is casting to a supertype, while downcasting is casting to a subtype. Upcasting is always allowed, but downcasting involves a type check and can throw a ClassCastException. This will give classCast exception

**Child c = (Child)Parent**

```
Animal animal = new Animal();  
Dog notADog = (Dog) animal;
```

*Java permits an object of a subclass type to be treated as an object of any superclass type. This is called upcasting. Upcasting is done automatically, while downcasting must be manually done by the programmer.*

- 30) Polymorphism is only for instance methods. Not instance variables.
- 31) Watch out for calling non static methods and variables from static methods.
- 32) In general, overloaded var-args methods are chosen last. Remember that arrays are objects. Finally, an int can be boxed to an Integer and then "widened" to an Object.

**33) Boxing and Widening is OK but opposite is not.**

**34) Check for static context , constructor , downcasting , package access , local variable initialization , local variable modifier, enum location, check for exceptions type thrown by method , overriding rules, exception sequence in catch block,constructor type , constructor chaining**

- 35) Int [][] can not be casted to Object?
- 36) The final modifier assures that a reference variable cannot be referred to a different object, but final doesn't keep the object's state from changing.
- 37) Static init blocks are executed at class loading time, instance init blocks run right after the call to super() in a constructor. When multiple init blocks of a single type occur in a class, they run in order, from the top down.
- 38) Duplicate variable local – int w; for (int w=0 ;w<4;w++) – this gives compile time duplicate variable error.

**39) Long and float / long and double/ int and long/ are comparable.**

- 40) Enum can be compared either equals or ==
- 41) instanceof is used to check if an object is an instance of a class, an instance of a subclass, or an instance of a class that implements a particular interface. We should use instanceof as less as possible as general coding rule.
- 42) Looks like instanceof check on actual object instane type not on reference type.
- 43) It is acceptable to use assertions to test the arguments of private methods
- 44) An overriding method cannot throw a broader exception than the method it's overriding.
- 45) An overriding method cannot throw checked exceptions that are broader than those thrown by the overridden method. However an overriding method *can* throw RuntimeExceptions not thrown by the overridden method
- 46) the equals() method for the integer wrappers will only return true if the two primitive types and the two values are equal.

```
Integer i = 7;  
long l = 7L;  
int x = 7;  
  
if(i.equals(x))System.out.println("case 1"); // true  
if(i.equals(l))System.out.println("case 1.1");  
if(i==l)System.out.println("case 2");// true
```

- 47) You need to call flush() only when you're writing data not on reader.
- 48) The readPassword() of console method returns a char[]
- 49) The invocation of parse() of NumberFormat class must be placed within a try/catch block.
- 50) List<List<Integer>> table = new ArrayList<ArrayList<Integer>>(); is incorrect The type argument <List<Integer>> must be the same for both sides of the assignment  
List<List<Integer>> table = new ArrayList<List<Integer>>();



- 51) In sorting capita letter comes first and then small letter B>b
- 52) Collection.Sort takes list<T> not list<Object>**
- 53) Don't use iterator object in for loop.**
- 54) Integer cannot be cast to Double.
- 55) Correct sequence is import static.
- 56) Sequence of reading object variable in serialization should be same as while writing object's variable.
- 57) By placing a zero in front of the number is an integer in octal form. 010 is in octal form so its value is 8.
- 58) Downcating for primitive gives compilation error, method take short as input parameter cannot be called with short param. Method(short s) -- > method(9) is WRONG.
- 59) You cannot eat exception in the subclass method while overriding.
- 60) creating a new instance of the class File, you're not yet making an actual file, you're just creating a filename. So file.exists() return false. createNewFile() method created an actual file.so file.exists() return true.

-----Class.forName() vs  
ClassLoader.loadClass()\*\*\*\*

Class.forName() will always use the ClassLoader of the caller,  
whereas ClassLoader.loadClass() can specify a different ClassLoader.

Class.forName initializes the loaded class as well,  
whereas the ClassLoader.loadClass() approach doesn't do that right away (it's not initialized until it's used for the first time).

-----Constructor

If you provide your own constructor with parameter but if you try to create object using no-arg constructor.  
Compiler will throw error.

you can call one constructor from another using this, you cannot call two.

In addition, the constructor call must be the first thing you do, or you'll get a compiler error message

super must be the first line in the constructor or else we get a compile time error

Local variables are not given default values and on using it before initializing will give error

variables are initialized before any methods can be called—even the constructor

the package statement must be the first non-comment code in the file

## -----Polymorphism

All method binding in Java uses late binding unless a method has been declared final. This allows the compiler to generate slightly more efficient code for final method calls. However, in most cases it won't make any overall performance difference in your program, so it's best to only use final as a design decision, and not as an attempt to improve performance.

When you override finalize( ) in an inherited class, it's important to remember to call the base-class version of finalize( ), since otherwise the base-class finalization will not happen.

The act of checking types at run-time is called run-time type identification (RTTI).

## -----INLINE FUNCTIONS

There is no inline function in java.

Yes, you can use a public static method anywhere in the code when placed in a public class.

The java compiler may do inline expansion on a static or final method, but that is not guaranteed.

## ----- JVM Warmup

"Warm-up" in Java is generally about two things:

(1): Lazy class loading: This can be work around by force it to load.

The easy way to do that is to send a fake message. You should be sure that the fake message will trigger all access to classes.

For exmaple, if you send an empty message but your program will check if the message is empty and avoid doing certain things, then this will not work.

Another way to do it is to force class initialization by accessing that class when you program starts.

(2): The realtime optimization: At run time, Java VM will optimize some part of the code. This is the major reason why there is a warm-up time at all.

To ease this, you can send bunch of fake (but look real) messages so that the optimization can finish before your user use it.

Another that you can help to ease this is to support inline such as using private and final as much as you can. the reason is that, the VM does not need to look up the inheritance table to see what method to actually be called.

#### -----Lazy Initialization

Lazy initialization has two objectives:  
delay an expensive operation until it's absolutely necessary  
store the result of that expensive operation, such that you won't need to repeat it again

#### ----- hash code and equals

Equals is always called after the hashCode method in a java hashed collection while adding and removing elements. The reason being, if there is an element already at the specified bucket, then JVM checks whether it is the same element which it is trying to put. In case if the equals returns false then the element is added to the same bucket but at the end of list at the bucket. So now you just don't have a single element at the same bucket but a list of elements.

Now while retrieving the element, first hashCode will be called to reach the desired bucket and then the list will be scanned using the equals to fetch the desired element.

The ideal implementation of hashCode will make sure the size of list at each bucket is 1.

And hence the retrieval of elements is done using  $O(1)$  complexity.

But if there are multiple elements stored in the list at a bucket, then the retrieval of element will be done by  $O(n)$  complexity, where  $n$  is the size of the list.

Btw in case of HashSet there is no list created at the bucket, rather the object is simply replaced if hashCode and equals are same.

The list creation behavior is in hashmap.

if you don't override a class's equals() method, you won't be able to use those objects as a key in a hashtable and you probably won't get accurate Sets, such that there are no conceptual duplicates.

-----how to handle collisions

1.Each bucket contains a linked list and whenever there is a collision the element get added at the end and can be retrieved correctly by use of equals method

2.Dynamic resizing : increase the number of buckets and rehash and redistribute all the elements

Java uses both methods

In order to address this issue in the case of frequent HashMap collisions, Java8 has started using a balanced tree instead of linked list for storing collided entries.

This also means that in the worst case you will get a performance boost from  $O(n)$  to  $O(\log n)$ .

-----main method

string array can be changed

sequence not issue

varargs can also be taken

final, synchronized and strictfp can also be used

You can use `System#exit(int)` to quit your program with a specific exit code which can be interpreted by the operating system

-----difference between == and equals\*

== is an operator while equals is a method

difference between == and equals method is that former is used to compare both primitive and objects while later is only used for objects comparison.

== compares reference while equals compares value  
eg form two strings with new operator and then compare

Integer overrides equals method and compares by value.

\*\*\*\*Int values from -127 to 127 are in a range which most JVM will like to cache so the VM actually uses the same object instance

#### -----Exception Heirarchy

Throwable-

Exception - RuntimeException(Unchecked - arithmetic exception, null pointer etc.), I/O Exception(Checked - EOF , File not found) , Interrupted etc.

Error(Unchecked) - VM Error(StackOverflow, OutOfMemory etc.), Linkage Error(Verify etc)

Only throwable and Exception are partially checked. Rest are either fully checked or un checked.

Checked Exceptions are checked at compile time by compiler.

Both checked and un-checked exceptions occur at run time.

#### -----Difference between final, finally and finalize

final - modifier applicable for classes(can't be inherited), methods(can't be overridden) and variables(constant)

finally - block associated with try/catch - cleanup activities like closing db connection

finalize - IS a method present in Object Class. Is called before GC claim the object to perform last-minute clean-up activities.

Protected access specifier

can be overridden but its our responsibility to call It

finalize gets called only once by GC thread if object revives itself from finalize method than finalize will not be called again

Any Exception is thrown by finalize method is ignored by GC thread and it will not be propagated further, in fact, I doubt if you find any trace of it.

There is one way to increase the probability of running of finalize method by calling `System.runFinalization()` and `Runtime.getRuntime().runFinalization()`.

These methods put more effort that JVM call `finalize()` method of all object which are eligible for garbage collection and whose finalize has not yet called.

It's not guaranteed, but JVM tries its best.

Read more:

<http://javarevisited.blogspot.com/2012/03/finalize-method-in-java-tutorial.html#ixzz3xavj34Xz>

-----ArrayList

ArrayList uses array of Object class to store all its elements internally.

Event though the real elements, which are objects to be stored can be allocated space anywhere on heap



but the refernces that will refer to all of them must have contiguous memory.

As long as OS is able to serve JVM's contiguous (big) memory request fast your application runs fast.

Since you are adding and removing elements from arrayList in bulk , there is a lot of array copying going on inside placing frequent memory allocation demands.

In fact, the performance of your application depends on many factors beyond your control. Most of them are memory allocation/delocation.

If you use linked list then memory allocation will be fast because it doesn't have to be contiguous and in bulk

```
int newCapacity = oldCapacity + (oldCapacity>> 1); 10+5  
= 15
```

-----Difference between shallow and deep copying

Shallow copies duplicate as little as possible. A shallow copy of a collection is a copy of the collection structure, not the elements.

With a shallow copy, two collections now share the individual elements. Class must implement cloneable interface.

Deep copies duplicate everything. A deep copy of a collection is two collections with all of the elements in the original collection duplicated

In a shallow copy, a new instance of the type is created and the values are copied into the new instance.

The reference pointers are also copied just like the values. Therefore, the references are pointing to the original objects.

Any changes to the members that are stored by reference appear in both the original and the copy, since no copy was made of the referenced object.

In a deep copy, the fields that are stored by value are copied as before, but the pointers to objects stored by reference are not copied.

Instead, a deep copy is made of the referenced object, and a pointer to the new object is stored.

Any changes that are made to those referenced objects will not affect other copies of the object.

-----Static keyword in java

static members belong to the class instead of a specific instance.

It means that only one instance of a static field exists[1] even if you create a million instances of the class or you don't create any.

It will be shared by all instances.

Since static methods also do not belong to a specific instance, they can't refer to instance members.

static members can only refer to static members. Instance members can, of course access static members.

Side note: Of course, static members can access instance members through an object reference.

Example:

```
public class Example {
    private static boolean staticField;
    private boolean instanceField;
    public static void main(String[] args) {
        // a static method can access static fields
        staticField = true;

        // a static method can access instance fields through
        an object reference
        Example instance = new Example();
        instance.instanceField = true;
    }
}
```

[1]: Depending on the runtime characteristics, it can be one per ClassLoader or AppDomain or thread, but that is beside the point.

If you're using a web application server like Tomcat or Websphere, you can get the same class loaded more than once in a couple of ways:

If you have copies of the same jar in two different WARs, each class will be loaded independently by different classloaders.

If you redeploy or restart a WAR, the same class from the same jar will again be loaded independently.

In both these cases, the static variables will not be shared.

When you fire up a JVM and load a class for the first time (this is done by the classloader when the class is first referenced in any way)

any static blocks or fields are 'loaded' into the JVM and become accessible.

If there are multiple static blocks then they will be loaded in the sequence they appear

When Class is loaded in Java

Class loading is done by ClassLoaders in Java which can be implemented to eagerly load a class as soon as another class references it or lazy load the class

until a need of class initialization occurs. If Class is loaded before its actually being used it can sit inside before being initialized.

I believe this may vary from JVM to JVM. While its guaranteed by JLS that a class will be loaded when there is a need of static initialization.

When a Class is initialized in Java

After class loading, initialization of class takes place which means initializing all static members of class. A Class is initialized in Java when :

- 1) an Instance of class is created using either new() keyword or using reflection using class.forName(), which may throw ClassNotFoundException in Java.
- 2) an static method of Class is invoked.
- 3) an static field of Class is assigned.
- 4) an static field of class is used which is not a constant variable.

5) if Class is a top level class and an assert statement lexically nested within class is executed.

Read more:

<http://javarevisited.blogspot.com/2012/07/when-class-loading-initialization-java-example.html#ixzz3zZTKCN5e>

-----Perm Space

\*\*\*The permanent generation is special because it holds meta-data describing user classes (classes that are not part of the Java language) and advance features like String pool created by intern().

Examples of such meta-data are objects describing classes and methods and they are stored in the Permanent Generation.

Applications with large code-base can quickly fill up this segment of the heap which will cause java.lang.OutOfMemoryError: PermGen no matter how high your -Xmx and how much memory you have on the machine.

Java 8 has decommissioned Perm Space and now Meta space is being used

<https://dzone.com/articles/java-8-permgen-metaspace>

-----Java I/O API

<http://tutorials.jenkov.com/java-io/index.html>

-----why do we ever use Integer class when we have  
int ???

\*\*\*\*answer would be null handling -- coz if I pass u an  
array of employees, which is 10 sized but has only 7 ppl in it  
which means three positions are null  
so if we use int then null will be silently treated as zero  
where as Integer will handle null properly

-----Variables

Local Variables can have only one keyword 'Final';

There can only be one var-arg parameter in the method and  
that can only be the last one;

Interface methods must not be static ;

Because interface methods are abstract, they cannot be  
marked final, strictfp, or native ;

An abstract class can exist with no abstract methods, an interface can have only abstract methods ;

Local variables are always on the stack, not the heap

unlike instance variables—local variables don't get default values.

array itself will always be an object on the heap, even if the array is declared to hold primitive elements

**\*\*It is never legal to include the size of the array in your declaration.**

Yes, we know you can do that in some other languages, which is why you might see a question or two that include code similar to the following:

```
int[5] scores;
```

The preceding code won't compile. Remember, the JVM doesn't allocate space until you actually instantiate the array object. That's when size matters.

A reference variable marked final can't ever be reassigned to refer to a different object

transient and volatile can be applied only to instance variable

-----ENUM

You can NEVER invoke an enum constructor directly. The enum constructor is invoked automatically, with the arguments you define after the constant value.

TRY IT

```
enum CoffeeSize {
    BIG(8),
    HUGE(10),
    OVERWHELMING(16) { // start a code block that defines
        // the "body" for this constant
        public String getLidCode() { // override the method
            // defined in CoffeeSize
            return "A";
        }
    }; // the semicolon is REQUIRED when more code follows
    CoffeeSize(int ounces) {
        this.ounces = ounces;
    }
    private int ounces;
    public int getOunces() {
        return ounces;
    }
    public String getLidCode() { // this method is overridden
        // by the OVERWHELMING constant
        return "B"; // the default value we want to return for
        // CoffeeSize constants
    }
}
```

-----Cloneable interface



A class implements the Cloneable interface to indicate to the Object.clone() method that it is legal for that method to make a field-for-field copy of instances of that class.

Invoking Object's clone method on an instance that does not implement the Cloneable interface results in the exception CloneNotSupportedException being thrown.

By convention, classes that implement this interface should override Object.clone (which is protected) with a public method.

See Object.clone() for details on overriding this method.

Note that this interface does not contain the clone method. Therefore, it is not possible to clone an object merely by virtue of the fact that it implements this interface. Even if the clone method is invoked reflectively, there is no guarantee that it will succeed.

## -----Marker Interfaces

To check if object is an instance of an interface one can use instanceof which is a relatively low-cost operation nowadays. Using annotations requires Java reflection calls and is far more costly.

Marker interfaces are better than annotations when they're used to define a type.

For example, Serializable can be used (and should be used) as the type of an argument that must be serializable.

An annotation doesn't allow doing that:

```
public void writeToFile(Serializable object);
```

If the marker interface doesn't define a type, but only meta-data, then an annotation is better

#### -----Escape Analysis

Java 6e14 added support for something called 'escape analysis'. When you enable it with -XX:+DoEscapeAnalysis switch,

then if JVM determines that an object created in a method, is used only in that method and there is no way for reference to the object to 'escape' that method -

i.e. we can be sure that the object is not referenced after method completes - JVM can allocate it on stack (treating all its fields as if they were local variables)

#### -----Future interface

A Future represents the result of an asynchronous computation. Methods are provided to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation.

The result can only be retrieved using method get when the computation has completed, blocking if necessary until it is ready.

Cancellation is performed by the cancel method. Additional methods are provided to determine if the task completed normally or was cancelled.

Once a computation has completed, the computation cannot be cancelled.

If you would like to use a Future for the sake of cancellability but not provide a usable result, you can declare types of the form Future<?> and return null as a result of the underlying task.

## JAVA ARCHITECTURE

### 299) Explain Java architecture?

Java's architecture arises out of four distinct but interrelated technologies:

- the Java programming language
- the Java class file format
- the Java Application Programming Interface
- the Java virtual machine

When you write and run a Java program, you are tapping the power of these four technologies.

You express the program in source files written in the Java programming language, compile the source to Java class files,

and run the class files on a Java virtual machine. When you write your program,

you access system resources (such as I/O, for example) by calling methods in the classes that implement the Java Application Programming

Interface, or Java API. As your program runs, it fulfills your program's Java API calls by invoking methods in class files that implement the Java API

Java virtual machine supports all three prongs of Java's network-oriented architecture: platform independence, security, and network-mobility.

Java virtual machines must be able to execute Java bytecodes, they may use any technique to execute them.

Also, the specification is flexible enough to allow a Java virtual machine to be implemented either completely in software or to varying degrees in hardware.

The flexible nature of the Java virtual machine's specification enables it to be implemented on a wide variety of computers and devices.

Java virtual machine's main job is to load class files and execute the bytecodes they contain. Java virtual machine contains a class loader,

which loads class files from both the program and the Java API.

Only those class files from the Java API that are actually needed by a running program are loaded into the virtual machine.

The bytecodes are executed in an execution engine.

The execution engine is one part of the virtual machine that can vary in different implementations.

On a Java virtual machine implemented in software, the simplest kind of execution engine just interprets the bytecodes one at a time.

Another kind of execution engine, one that is faster but requires more memory, is a just-in-time compiler.

In this scheme, the bytecodes of a method are compiled to native machine code the first time the method is invoked.

The native machine code for the method is then cached, so it can be re-used the next time that same method is invoked.

A third type of execution engine is an adaptive optimizer. In this approach, the virtual machine starts by interpreting bytecodes,

but monitors the activity of the running program and identifies the most heavily used areas of code.

As the program runs, the virtual machine compiles to native and optimizes just these heavily used areas.

The rest of the code, which is not heavily used, remain as bytecodes which the virtual machine continues to interpret.

This adaptive optimization approach enables a Java virtual machine to spend typically 80 to 90% of its time executing highly optimized native code,

while requiring it to compile and optimize only the 10 to 20% of the code that really matters to performance.

Lastly, on a Java virtual machine built on top of a chip that executes Java bytecodes natively, the execution engine is actually embedded in the chip.

Sometimes the Java virtual machine is called the Java interpreter; however, given the various ways in which bytecodes can be executed, this term can be misleading.

While "Java interpreter" is a reasonable name for a Java virtual machine that interprets bytecodes, virtual machines also use other techniques

(such as just-in-time compiling) to execute bytecodes.

Therefore, although all Java interpreters are Java virtual machines, not all Java virtual machines are Java interpreters.

When running on a Java virtual machine that is implemented in software on top of a host operating system,

a Java program interacts with the host by invoking native methods. In Java, there are two kinds of methods: Java and native.

A Java method is written in the Java language, compiled to bytecodes, and stored in class files.

A native method is written in some other language, such as C, C++, or assembly, and compiled to the native machine code of a particular processor.

Native methods are stored in a dynamically linked library whose exact form is platform specific.

While Java methods are platform independent, native methods are not.

When a running Java program calls a native method, the virtual machine loads the dynamic library that contains the native method and invokes it.

Native methods are the connection between a Java program and an underlying host operating system.

Java gives you a choice. If you want to access resources of a particular host that are unavailable through the Java API,

you can write a platform-specific Java program that calls native methods.

If you want to keep your program platform independent, however, you must access the system resources of the underlying operating system only through the Java API.

-----Class loaders and network security

A Java application can use two types of class loaders: a "bootstrap" class loader and user-defined class loaders.

The bootstrap class loader (there is only one of them) is a part of the Java virtual machine implementation.

For example, if a Java virtual machine is implemented as a C program on top of an existing operating system,

then the bootstrap class loader will be part of that C program.

For each class it loads, the Java virtual machine keeps track of which class loader--whether bootstrap or user-defined--loaded the class.

When a loaded class first refers to another class, the virtual machine requests the referenced class from the same class loader

that originally loaded the referencing class.

For example, if the virtual machine loads class Volcano through a particular class loader,

it will attempt to load any classes Volcano refers to through the same class loader.

If Volcano refers to a class named Lava, perhaps by invoking a method in class Lava, the virtual machine will request Lava from the class loader that loaded Volcano.

The Lava class returned by the class loader is dynamically linked with class Volcano.

Because the Java virtual machine takes this approach to loading classes, classes can by default only see other classes that were loaded by the same class loader.

In this way, Java's architecture enables you to create multiple name-spaces inside a single Java application.

Each class loader in your running Java program has its own name-space, which is populated by the names of all the classes it has loaded

The Java application started by the web browser usually creates a different user-defined class loader for each location on the network

from which it retrieves class files. As a result, class files from different sources are loaded by different user-defined class loaders.

This places them into different name-spaces inside the host Java application.

Because the class files for applets from different sources are placed in separate name-spaces,

the code of a malicious applet is restricted from interfering directly with class files downloaded from any other source.

By allowing you to instantiate user-defined class loaders that know how to download class files across a network,

Java's class loader architecture supports network-mobility.

It supports security by allowing you to load class files from different sources through different user-defined class loaders.

This puts the class files from different sources into different name-spaces, which allows you to restrict or prevent access between code loaded from different sources

-----JAVA vs C++

When you compile and link a C++ program, the executable binary file you get is specific to a particular target hardware platform and operating system

because it contains machine language specific to the target processor.

A Java compiler, by contrast, translates the instructions of the Java source files into bytecodes, the "machine language" of the Java virtual machine.

In Java, there is no way to directly access memory by arbitrarily casting pointers to a different type or by using pointer arithmetic, as there is in C++.

Java requires that you strictly obey rules of type when working with objects.

If you have a reference (similar to a pointer in C++) to an object of type Mountain, you can only manipulate it as a Mountain.

You can't cast the reference to type Lava and manipulate the memory as if it were a Lava.

Another way Java prevents you from inadvertently corrupting memory is through automatic garbage collection.

A third way Java protects the integrity of memory at run-time is array bounds checking.

One final example of how Java ensures program robustness is by checking object references, each time they are used,

to make sure they are not null. In C++, using a null pointer usually results in a program crash.

In Java, using a null reference results in an exception being thrown.

The Java API is set of runtime libraries that give you a standard way to access the system resources of a host computer.

The class files of the Java API are inherently specific to the host platform.

The API's functionality must be implemented expressly for a particular platform before that platform can host Java programs.

To access the native resources of the host, the Java API calls native methods.

In addition to facilitating platform independence, the Java API contributes to Java's security model.

The methods of the Java API, before they perform any action that could potentially be harmful (such as writing to the local disk), check for permission.

In Java releases prior to 1.2, the methods of the Java API checked permission by querying the security manager.

The security manager is a special object that defines a custom security policy for the application.

A security manager could, for example, forbid access to the local disk.

If the application requested a local disk write by invoking a method from the pre-1.2 Java API, that method would first check with the security manager.

Upon learning from the security manager that disk access is forbidden, the Java API would refuse to perform the write.



In Java 1.2, the job of the security manager was taken over by the access controller,

a class that performs stack inspection to determine whether the operation should be allowed.

(For backwards compatibility, the security manager still exists in Java 1.2.)

By enforcing the security policy established by the security manager and access controller,

the Java API helps to establish a safe environment in which you can run potentially unsafe code.

The Java programming language reflects Java's platform independence in one principal way: the ranges and behavior of its primitive types are defined by the language.

In languages such as C or C++, the range of the primitive type int is determined by its size, and its size is determined by the target platform.

The size of an int in C or C++ is generally chosen by the compiler to match the word size of the platform for which the program is compiled.

This means that a C++ program might have different behavior when compiled for different platforms merely

because the ranges of the primitive types are not consistent across the platforms.

For example, no matter what underlying platform might be hosting the program, an int in Java behaves as a signed 32-bit two's complement number.

A float adheres to the 32-bit IEEE 754 floating point standard. This consistency is also reflected in the internals of the Java virtual machine,

which has primitive data types that match those of the language, and in the class file, where the same primitive data types appear.

By guaranteeing that primitive types behave the same on all platforms,

the Java language itself promotes the platform independence of Java programs.

Security:

<http://www.artima.com/insidejvm/ed2/security.html>

Network Mobility

<http://www.artima.com/insidejvm/ed2/netmob.html>

## JVM INTERNALS

### 300) Loader vs initializer vs linking

There are actually three steps in preparing a class for use:

1.Loading, which is performed by the class loader.

This finds the bytecodes (usually, but not necessarily, on your disk in your classpath) and creates a Class object from those bytecodes.

2.Linkage. The link phase verifies the bytecodes in the class, allocates storage for static fields, and if necessary,

resolves all references to other classes made by this class.

3.Initialization. If there's a superclass, initialize that. Execute static initializers and static initialization blocks.

Initialization is delayed until the first reference to a static method (the constructor is implicitly static) or to a non-constant static field:

creating a reference to a Class object using ".class" doesn't automatically initialize the Class object

If a static final value is a "compile-time constant," such as `Initable.staticFinal`, that value can be read without causing the `Initable` class to be initialized.

Making a field static and final, however, does not guarantee this behavior: it might be using some other classes method

If a static field is not final, accessing it always requires linking and initialization before it can be read

`instanceof` should be preferred whenever you know the kind of class you want to check against in advance. In those very rare cases where you do not, use `isInstance()` instead.

### 301) Explain JVM internals?

<https://www.artima.com/insidejvm/ed2/jvm.html>

A runtime instance of the Java virtual machine has a clear mission in life: to run one Java application. When a Java application starts, a runtime instance is born.

When the application completes, the instance dies. If you start three Java applications at the same time, on the same computer, using the same concrete implementation,

you'll get three Java virtual machine instances. Each Java application runs inside its own Java virtual machine.

Inside the Java virtual machine, threads come in two flavors: daemon and non- daemon.

A daemon thread is ordinarily a thread used by the virtual machine itself, such as a thread that performs garbage collection.

The application, however, can mark any threads it creates as daemon threads.

The initial thread of an application--the one that begins at main()--is a non-daemon thread.

A Java application continues to execute (the virtual machine instance continues to live) as long as any non-daemon threads are still running.

When all non-daemon threads of a Java application terminate, the virtual machine instance will exit.

If permitted by the security manager, the application can also cause its own demise by invoking the exit() method class Runtime or System.

Each Java virtual machine has a class loader subsystem: a mechanism for loading types (classes and interfaces) given fully qualified names.

Each Java virtual machine also has an execution engine: a mechanism responsible for executing the instructions contained in the methods of loaded classes.

When a Java virtual machine runs a program, it needs memory to store many things, including bytecodes and other information it extracts from loaded class files,

objects the program instantiates, parameters to methods, return values, local variables, and intermediate results of computations.

The Java virtual machine organizes the memory it needs to execute a program into several runtime data areas.

Although the same runtime data areas exist in some form in every Java virtual machine implementation, their specification is quite abstract.

Many decisions about the structural details of the runtime data areas are left to the designers of individual implementations.

Different implementations of the virtual machine can have very different memory constraints.

Some implementations may have a lot of memory in which to work, others may have very little.

Some implementations may be able to take advantage of virtual memory, others may not.

The abstract nature of the specification of the runtime data areas helps make it easier to implement the Java virtual machine on a wide variety of computers

and devices.

Some runtime data areas are shared among all of an application's threads and others are unique to individual threads.

Each instance of the Java virtual machine has one method area and one heap.

These areas are shared by all threads running inside the virtual machine.

When the virtual machine loads a class file, it parses information about a type from the binary data contained in the class file.

It places this type information into the method area. As the program runs, the virtual machine places all objects the program instantiates onto the heap.

As each new thread comes into existence, it gets its own pc register (program counter) and Java stack.

If the thread is executing a Java method (not a native method), the value of the pc register indicates the next instruction to execute.

A thread's Java stack stores the state of Java (not native) method invocations for the thread.

The state of a Java method invocation includes its local variables, the parameters with which it was invoked, its return value (if any), and intermediate calculations.

The state of native method invocations is stored in an implementation-dependent way in native method stacks,

as well as possibly in registers or other implementation-dependent memory areas.

The Java stack is composed of stack frames (or frames). A stack frame contains the state of one Java method invocation.

When a thread invokes a method, the Java virtual machine pushes a new frame onto that thread's Java stack.

When the method completes, the virtual machine pops and discards the frame for that method.

The Java virtual machine has no registers to hold intermediate data values. The instruction set uses the Java stack for storage of intermediate data values.

This approach was taken by Java's designers to keep the Java virtual machine's instruction set compact

and to facilitate implementation on architectures with few or irregular general purpose registers.

In addition, the stack-based architecture of the Java virtual machine's instruction set facilitates the code optimization work done by just-in-time

and dynamic compilers that operate at run-time in some virtual machine implementations

#### ----- Data Types

The data types can be divided into a set of primitive types and a reference type.

Variables of the primitive types hold primitive values, and variables of the reference type hold reference values.

Reference values refer to objects, but are not objects themselves. Primitive values, by contrast, do not refer to anything. They are the actual data themselves.

All the primitive types of the Java programming language are primitive types of the Java virtual machine.

Although boolean qualifies as a primitive type of the Java virtual machine, the instruction set has very limited support for it.

When a compiler translates Java source code into bytecodes, it uses ints or bytes to represent booleans.

In the Java virtual machine, false is represented by integer zero and true by any non-zero integer.

Operations involving boolean values use ints.

Arrays of boolean are accessed as arrays of byte, though they may be represented on the heap as arrays of byte or as bit fields.

The primitive types of the Java programming language other than boolean form the numeric types of the Java virtual machine.

The numeric types are divided between the integral types: byte, short, int, long, and char, and the floating- point types: float and double.

As with the Java programming language, the primitive types of the Java virtual machine have the same range everywhere.

A long in the Java virtual machine always acts like a 64-bit signed twos complement number, independent of the underlying host platform.

The Java virtual machine works with one other primitive type that is unavailable to the Java programmer: the returnAddress type.

This primitive type is used to implement finally clauses of Java programs.  
How ??

The reference type of the Java virtual machine is cleverly named reference. Values of type reference come in three flavors:

the class type, the interface type, and the array type. All three types have values that are references to dynamically created objects.

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

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

The interface type's values are references to class instances that implement an interface.

One other reference value is the null value, which indicates the reference variable doesn't refer to any object.

byte 8-bit signed two's complement integer ( $-2^7$  to  $2^7 - 1$ , inclusive)

short 16-bit signed two's complement integer ( $-2^{15}$  to  $2^{15} - 1$ , inclusive)

int 32-bit signed two's complement integer ( $-2^{31}$  to  $2^{31} - 1$ , inclusive)

long 64-bit signed two's complement integer ( $-2^{63}$  to  $2^{63} - 1$ , inclusive)

char 16-bit unsigned Unicode character (0 to  $2^{16} - 1$ , inclusive)

float 32-bit IEEE 754 single-precision float

double 64-bit IEEE 754 double-precision float

returnAddress address of an opcode within the same method

reference reference to an object on the heap, or null

-----Word Size\*\*\*

The basic unit of size for data values in the Java virtual machine is the word-- a fixed size chosen by the designer of each Java virtual machine implementation.

The word size must be large enough to hold a value of type byte, short, int, char, float, returnAddress, or reference.

Two words must be large enough to hold a value of type long or double.

An implementation designer must therefore choose a word size that is at least 32 bits,

but otherwise can pick whatever word size will yield the most efficient implementation.

The word size is often chosen to be the size of a native pointer on the host platform.

The specification of many of the Java virtual machine's runtime data areas are based upon this abstract concept of a word.

For example, two sections of a Java stack frame--the local variables and operand stack-- are defined in terms of words.

These areas can contain values of any of the virtual machine's data types. When placed into the local variables or operand stack,

a value occupies either one or two words.

As they run, Java programs cannot determine the word size of their host virtual machine implementation.

The word size does not affect the behavior of a program. It is only an internal attribute of a virtual machine implementation.



#### -----Class Loader Subsystem

Java virtual machine contains two kinds of class loaders: a bootstrap class loader and user-defined class loaders.

The bootstrap class loader is a part of the virtual machine implementation, and user-defined class loaders are part of the running Java application.

Classes loaded by different class loaders are placed into separate name spaces inside the Java virtual machine.

The class loader subsystem involves many other parts of the Java virtual machine and several classes from the `java.lang` library.

For example, user-defined class loaders are regular Java objects whose class descends from `java.lang.ClassLoader`.

The methods of class `ClassLoader` allow Java applications to access the virtual machine's class loading machinery.

Also, for every type a Java virtual machine loads, it creates an instance of class `java.lang.Class` to represent that type.

Like all objects, user-defined class loaders and instances of class `Class` reside on the heap. Data for loaded types resides in the method area.

#### -----Loading, Linking and Initialization

The class loader subsystem is responsible for more than just locating and importing the binary data for classes.

It must also verify the correctness of imported classes, allocate and initialize memory for class variables, and assist in the resolution of symbolic references.

These activities are performed in a strict order:

1. Loading: finding and importing the binary data for a type
2. Linking: performing verification, preparation, and (optionally) resolution a.  
Verification: ensuring the correctness of the imported type

- b. Preparation: allocating memory for class variables and initializing the memory to default values
  - c. Resolution: transforming symbolic references from the type into direct references.
3. Initialization: invoking Java code that initializes class variables to their proper starting values.

#### -----User-Defined Class Loaders

Although user-defined class loaders themselves are part of the Java application, four of the methods in class `ClassLoader` are gateways into the Java virtual machine:

// Four of the methods declared in class `java.lang.ClassLoader`:

```
protected final Class defineClass(String name, byte data[],
    int offset, int length);
protected final Class defineClass(String name, byte data[],
    int offset, int length, ProtectionDomain protectionDomain);
protected final Class findSystemClass(String name);
protected final void resolveClass(Class c);
```

Any Java virtual machine implementation must take care to connect these methods of class `ClassLoader` to the internal class loader subsystem.

The two overloaded `defineClass()` methods accept a byte array, `data[]`, as input.

Starting at position `offset` in the array and continuing for `length` bytes, class `ClassLoader` expects binary data conforming to the Java class file format--binary data that represents a new type for the running application -- with

the fully qualified name specified in `name`. The type is assigned to either a default protection domain, if the first version of `defineClass()` is used,

or to the protection domain object referenced by the `protectionDomain` parameter.

Every Java virtual machine implementation must make sure the `defineClass()` method of class `ClassLoader` can cause a new type to be imported into the method area.

The `findSystemClass()` method accepts a `String` representing a fully qualified name of a type.

When a user-defined class loader invokes this method in version 1.0 and 1.1, it is requesting that the virtual machine attempt to load the named type via its bootstrap class loader.

If the bootstrap class loader has already loaded or successfully loads the type, it returns a reference to the `Class` object representing the type.

If it can't locate the binary data for the type, it throws `ClassNotFoundException`.

In version 1.2, the `findSystemClass()` method attempts to load the requested type from the system class loader.

Every Java virtual machine implementation must make sure the `findSystemClass()` method can invoke the bootstrap (if version 1.0 or 1.1) or system (if version 1.2 or later) class loader in this way.

The `resolveClass()` method accepts a reference to a `Class` instance.

This method causes the type represented by the `Class` instance to be linked (if it hasn't already been linked).

The `defineClass()` method, described previous, only takes care of loading.

(See the previous section, "Loading, Linking, and Initialization" for definitions of these terms.)

When `defineClass()` returns a `Class` instance, the binary file for the type has definitely been located and imported into the method area, but not necessarily linked and initialized.

Java virtual machine implementations make sure the `resolveClass()` method of class `ClassLoader` can cause the class loader subsystem to perform linking.

When to compare using getClass() and when getClass().getName()?

If you want to know whether two objects are of the same type you should use the equals method to compare the two classes -- the first option.

I can't imagine why you'd want to do this, but if you want to know whether two objects with different concrete types have types with the same fully qualified name,

then you could use the second

-----The Method Area

All threads share the same method area, so access to the method area's data structures must be designed to be thread-safe.

If two threads are attempting to find a class named Lava, for example, and Lava has not yet been loaded, only one thread should be allowed to load it while the other one waits.

The size of the method area need not be fixed. As the Java application runs, the virtual machine can expand and contract the method area to fit the application's needs.

Also, the memory of the method area need not be contiguous. It could be allocated on a heap--even on the virtual machine's own heap.

Implementations may allow users or programmers to specify an initial size for the method area, as well as a maximum or minimum size.

The method area can also be garbage collected. Because Java programs can be dynamically extended via user-defined class loaders,

classes can become "unreferenced" by the application.

If a class becomes unreferenced, a Java virtual machine can unload the class (garbage collect it) to keep the memory occupied by the method area at a minimum.

-----Type Information

For each type it loads, a Java virtual machine must store the following kinds of information in the method area:

- The fully qualified name of the type
- The fully qualified name of the type's direct superclass (unless the type is an interface or class `java.lang.Object`, neither of which have a superclass)
- Whether or not the type is a class or an interface
- The type's modifiers ( some subset of ` public, abstract, final)
- An ordered list of the fully qualified names of any direct superinterfaces

Inside the Java class file and Java virtual machine, type names are always stored as fully qualified names.

In Java source code, a fully qualified name is the name of a type's package, plus a dot, plus the type's simple name.

For example, the fully qualified name of class `Object` in package `java.lang` is `java.lang.Object`.

In class files, the dots are replaced by slashes, as in `java/lang/Object`. In the method area,

fully qualified names can be represented in whatever form and data structures a designer chooses.

In addition to the basic type information listed previously, the virtual machine must also store for each loaded type:

- The constant pool for the type
- Field information
- Method information
- All class (static) variables declared in the type, except constants
- A reference to class `ClassLoader`
- A reference to class `Class`

This data is described in the following sections.

The Constant Pool

For each type it loads, a Java virtual machine must store a constant pool. A constant pool is an ordered set of constants used by the type,

including literals (string, integer, and floating point constants) and symbolic references to types, fields, and methods.

Entries in the constant pool are referenced by index, much like the elements of an array.

Because it holds symbolic references to all types, fields, and methods used by a type, the constant pool plays a central role in the dynamic linking of Java programs.

The constant pool is described in more detail later in this chapter and in Chapter 6, "The Java Class File."

#### Field Information

For each field declared in the type, the following information must be stored in the method area. In addition to the information for each field,

the order in which the fields are declared by the class or interface must also be recorded. Here's the list for fields:

- The field's name
- The field's type
- The field's modifiers (some subset of public, private, protected, static, final, volatile, transient)

#### Method Information

For each method declared in the type, the following information must be stored in the method area.

As with fields, the order in which the methods are declared by the class or interface must be recorded as well as the data. Here's the list:

- The method's name
- The method's return type (or void)
- The number and types (in order) of the method's parameters
- The method's modifiers (some subset of public, private, protected, static, final, synchronized, native, abstract)

In addition to the items listed previously, the following information must also be stored with each method that is not abstract or native:

- The method's bytecodes

- The sizes of the operand stack and local variables sections of the method's stack frame (these are described in a later section of this chapter)
- An exception table (this is described in Chapter 17, "Exceptions")

### Class Variables

Class variables are shared among all instances of a class and can be accessed even in the absence of any instance.

These variables are associated with the class--not with instances of the class--so they are logically part of the class data in the method area.

Before a Java virtual machine uses a class, it must allocate memory from the method area for each non-final class variable declared in the class.

Constants (class variables declared final) are not treated in the same way as non-final class variables.

Every type that uses a final class variable gets a copy of the constant value in its own constant pool.

As part of the constant pool, final class variables are stored in the method area--just like non-final class variables.

But whereas non-final class variables are stored as part of the data for the type that declares them,

final class variables are stored as part of the data for any type that uses them.

This special treatment of constants is explained in more detail in Chapter 6, "The Java Class File."

### A Reference to Class ClassLoader

For each type it loads, a Java virtual machine must keep track of whether or not the type was loaded via the bootstrap class loader or a user-defined class loader.

For those types loaded via a user-defined class loader, the virtual machine must store a reference to the user-defined class loader that loaded the type.

This information is stored as part of the type's data in the method area.

The virtual machine uses this information during dynamic linking. When one type refers to another type,

the virtual machine requests the referenced type from the same class loader that loaded the referencing type.

This process of dynamic linking is also central to the way the virtual machine forms separate name spaces.

To be able to properly perform dynamic linking and maintain multiple name spaces,

the virtual machine needs to know what class loader loaded each type in its method area.

The details of dynamic linking and name spaces are given in Chapter 8, "The Linking Model."

#### A Reference to Class Class

An instance of class `java.lang.Class` is created by the Java virtual machine for every type it loads.

The virtual machine must in some way associate a reference to the `Class` instance for a type with the type's data in the method area.

Your Java programs can obtain and use references to `Class` objects. One static method in class `Class`,

allows you to get a reference to the `Class` instance for any loaded class:

```
// A method declared in class java.lang.Class:  
public static Class forName(String className);
```

If you invoke `forName("java.lang.Object")`, for example, you will get a reference to the `Class` object that represents `java.lang.Object`.

If you invoke `forName("java.util Enumeration")`, you will get a reference to the `Class` object that represents the `Enumeration` interface from the `java.util` package.



You can use `forName()` to get a Class reference for any loaded type from any package,

so long as the type can be (or already has been) loaded into the current name space.

If the virtual machine is unable to load the requested type into the current name space, `forName()` will throw `ClassNotFoundException`.

An alternative way to get a Class reference is to invoke `getClass()` on any object reference.

This method is inherited by every object from class `Object` itself:

// A method declared in class `java.lang.Object`:

```
public final Class getClass();
```

If you have a reference to an object of class `java.lang.Integer`,

for example, you could get the Class object for `java.lang.Integer` simply by invoking `getClass()` on your reference to the Integer object.

Given a reference to a Class object, you can find out information about the type by invoking methods declared in class `Class`.

If you look at these methods, you will quickly realize that class `Class` gives the running application access to the information stored in the method area.

Here are some of the methods declared in class `Class`:

// Some of the methods declared in class `java.lang.Class`:

```
public String getName();
```

```
public Class getSuperClass();
```

```
public boolean isInterface();
```

```
public Class[] getInterfaces();
```

```
public ClassLoader getClassLoader();
```

These methods just return information about a loaded type. `getName()` returns the fully qualified name of the type.

`getSuperClass()` returns the `Class` instance for the type's direct superclass. If the type is class `java.lang.Object` or an interface,

none of which have a superclass, `getSuperClass()` returns null. `isInterface()` returns true if the `Class` object describes an interface,

false if it describes a class. `getInterfaces()` returns an array of `Class` objects, one for each direct superinterface.

The superinterfaces appear in the array in the order they are declared as superinterfaces by the type.

If the type has no direct superinterfaces, `getInterfaces()` returns an array of length zero. `getClassLoader()` returns a reference to the `ClassLoader` object

that loaded this type, or null if the type was loaded by the bootstrap class loader. All this information comes straight out of the method area.

## Method Tables

The type information stored in the method area must be organized to be quickly accessible. In addition to the raw type information listed previously, implementations may include other data structures that speed up access to the raw data. One example of such a data structure is a method table.

For each non-abstract class a Java virtual machine loads, it could generate a method table and include it as part of the class information it stores in the method area.

A method table is an array of direct references to all the instance methods that may be invoked on a class instance,

including instance methods inherited from superclasses. (A method table isn't helpful in the case of abstract classes or interfaces,

because the program will never instantiate these.) A method table allows a virtual machine to quickly locate an instance method invoked on an object.

## -----An Example of Method Area Use

As an example of how the Java virtual machine uses the information it stores in the method area, consider these classes:

```
// On CD-ROM in file jvm/ex2/Lava.java  
class Lava {  
  
    private int speed = 5; // 5 kilometers per hour  
  
    void flow() {  
    }  
}
```

```
// On CD-ROM in file jvm/ex2/Volcano.java  
class Volcano {  
  
    public static void main(String[] args) {  
        Lava lava = new Lava();  
        lava.flow();  
    }  
}
```

The following paragraphs describe how an implementation might execute the first instruction in the bytecodes for the main() method of the Volcano application.

Different implementations of the Java virtual machine can operate in very different ways.

The following description illustrates one way--but not the only way--a Java virtual machine could execute the first instruction of Volcano's main() method.

To run the Volcano application, you give the name "Volcano" to a Java virtual machine in an implementation-dependent manner. Given the name Volcano, the virtual machine finds and reads in file Volcano.class. It extracts the definition of class Volcano from the binary data in the imported class file

and places the information into the method area. The virtual machine then invokes the `main()` method, by interpreting the bytecodes stored in the method area.

As the virtual machine executes `main()`, it maintains a pointer to the constant pool (a data structure in the method area) for the current class (class `Volcano`).

Note that this Java virtual machine has already begun to execute the bytecodes for `main()` in class `Volcano` even though it hasn't yet loaded class `Lava`.

Like many (probably most) implementations of the Java virtual machine, this implementation doesn't wait until all classes used by the application are loaded

before it begins executing `main()`. It loads classes only as it needs them.

`main()`'s first instruction tells the Java virtual machine to allocate enough memory for the class listed in constant pool entry one.

The virtual machine uses its pointer into `Volcano`'s constant pool to look up entry one and finds a symbolic reference to class `Lava`.

It checks the method area to see if `Lava` has already been loaded.

The symbolic reference is just a string giving the class's fully qualified name: `"Lava"`.

Here you can see that the method area must be organized so a class can be located--as quickly as possible--given only the class's fully qualified name.

Implementation designers can choose whatever algorithm and data structures best fit their needs--a hash table, a search tree, anything.

This same mechanism can be used by the static `forName()` method of class `Class`, which returns a `Class` reference given a fully qualified name.

When the virtual machine discovers that it hasn't yet loaded a class named `"Lava,"` it proceeds to find and read in file `Lava.class`.

It extracts the definition of class `Lava` from the imported binary data and places the information into the method area.

The Java virtual machine then replaces the symbolic reference in `Volcano`'s constant pool entry one,

which is just the string "Lava", with a pointer to the class data for Lava.

If the virtual machine ever has to use Volcano's constant pool entry one again,

it won't have to go through the relatively slow process of searching through the method area for class Lava given only a symbolic reference, the string "Lava".

It can just use the pointer to more quickly access the class data for Lava. This process of replacing symbolic references with direct references

(in this case, a native pointer) is called constant pool resolution.

The symbolic reference is resolved into a direct reference by searching through the method area until the referenced entity is found, loading new classes if necessary.

Finally, the virtual machine is ready to actually allocate memory for a new Lava object. Once again,

the virtual machine consults the information stored in the method area.

It uses the pointer (which was just put into Volcano's constant pool entry one) to the Lava data (which was just imported into the method area)

to find out how much heap space is required by a Lava object.

A Java virtual machine can always determine the amount of memory required to represent an object by looking into the class data stored in the method area.

The actual amount of heap space required by a particular object, however, is implementation-dependent.

The internal representation of objects inside a Java virtual machine is another decision of implementation designers.

Object representation is discussed in more detail later in this chapter.

Once the Java virtual machine has determined the amount of heap space required by a Lava object,

it allocates that space on the heap and initializes the instance variable speed to zero, its default initial value.

If class Lava's superclass, Object, has any instance variables, those are also initialized to default initial values.

The first instruction of main() completes by pushing a reference to the new Java object onto the stack.

A later instruction will use the reference to invoke Java code that initializes the speed variable to its proper initial value, five.

Another instruction will use the reference to invoke the flow() method on the referenced Java object.

#### -----The Heap

Whenever a class instance or array is created in a running Java application, the memory for the new object is allocated from a single heap.

As there is only one heap inside a Java virtual machine instance, all threads share it.

Because a Java application runs inside its "own" exclusive Java virtual machine instance, there is a separate heap for every individual running application.

There is no way two different Java applications could trample on each other's heap data.

Two different threads of the same application, however, could trample on each other's heap data.

This is why you must be concerned about proper synchronization of multi-threaded access to objects (heap data) in your Java programs.

The Java virtual machine has an instruction that allocates memory on the heap for a new object, but has no instruction for freeing that memory.

Just as you can't explicitly free an object in Java source code, you can't explicitly free an object in Java bytecodes.

The virtual machine itself is responsible for deciding whether and when to free memory occupied by objects that are no longer referenced by the running application.

Usually, a Java virtual machine implementation uses a garbage collector to manage the heap

-----Object Representation

The primary data that must in some way be represented for each object is the instance variables declared in the object's class and all its superclasses.

Given an object reference, the virtual machine must be able to quickly locate the instance data for the object.

In addition, there must be some way to access an object's class data (stored in the method area) given a reference to the object.

For this reason, the memory allocated for an object usually includes some kind of pointer into the method area.

One possible heap design divides the heap into two parts: a handle pool and an object pool. An object reference is a native pointer to a handle pool entry.

A handle pool entry has two components: a pointer to instance data in the object pool and a pointer to class data in the method area.

The advantage of this scheme is that it makes it easy for the virtual machine to combat heap fragmentation.

When the virtual machine moves an object in the object pool, it need only update one pointer with the object's new address: the relevant pointer in the handle pool.

The disadvantage of this approach is that every access to an object's instance data requires dereferencing two pointers

Another design makes an object reference a native pointer to a bundle of data that contains the object's instance data and a pointer to the object's class data.

This approach requires dereferencing only one pointer to access an object's instance data, but makes moving objects more complicated.

When the virtual machine moves an object to combat fragmentation of this kind of heap, it must update every reference to that object anywhere in the runtime data areas.

The virtual machine needs to get from an object reference to that object's class data for several reasons.

When a running program attempts to cast an object reference to another type,

the virtual machine must check to see if the type being cast to is the actual class of the referenced object or one of its supertypes. .

It must perform the same kind of check when a program performs an instanceof operation.

In either case, the virtual machine must look into the class data of the referenced object.

When a program invokes an instance method, the virtual machine must perform dynamic binding: it must choose the method to invoke based not on the type of the reference

but on the class of the object. To do this, it must once again have access to the class data given only a reference to the object

No matter what object representation an implementation uses, it is likely that a method table is close at hand for each object.

Method tables, because they speed up the invocation of instance methods, can play an important role in achieving good overall performance

for a virtual machine implementation. Method tables are not required by the Java virtual machine specification and may not exist in all implementations.

Implementations that have extremely low memory requirements, for instance, may not be able to afford the extra memory space method tables occupy.

If an implementation does use method tables, however, an object's method table will likely be quickly accessible given just a reference to the object

The special structure has two components:

- A pointer to the full the class data for the object
- The method table for the object The method table is an array of pointers to the data for each instance method that can be invoked on objects of that class.

The method data pointed to by method table includes:

- The sizes of the operand stack and local variables sections of the method's stack
- The method's bytecodes
- An exception table



#### -----Array Representation

In Java, arrays are full-fledged objects. Like objects, arrays are always stored on the heap. Also like objects,

implementation designers can decide how they want to represent arrays on the heap.

Arrays have a Class instance associated with their class, just like any other object. All arrays of the same dimension and type have the same class.

The length of an array (or the lengths of each dimension of a multidimensional array) does not play any role in establishing the array's class.

For example, an array of three ints has the same class as an array of three hundred ints. The length of an array is considered part of its instance data.

The name of an array's class has one open square bracket for each dimension plus a letter or string representing the array's type.

For example, the class name for an array of ints is "[I". The class name for a three-dimensional array of bytes is "[[[B".

The class name for a two-dimensional array of Objects is "[[Ljava.lang.Object".

<https://www.artima.com/insidejvm/ed2/jvm6.html>

#### -----The Program Counter

Each thread of a running program has its own pc register, or program counter, which is created when the thread is started. The pc register is one word in size,

so it can hold both a native pointer and a returnAddress.

As a thread executes a Java method, the pc register contains the address of the current instruction being executed by the thread.

An "address" can be a native pointer or an offset from the beginning of a method's bytecodes. If a thread is executing a native method, the value of the pc register is undefined.

#### -----The Java Stack

When a new thread is launched, the Java virtual machine creates a new Java stack for the thread. As mentioned earlier,

a Java stack stores a thread's state in discrete frames. The Java virtual machine only performs two operations directly on Java Stacks: it pushes and pops frames.

The method that is currently being executed by a thread is the thread's current method. The stack frame for the current method is the current frame.

The class in which the current method is defined is called the current class, and the current class's constant pool is the current constant pool.

As it executes a method, the Java virtual machine keeps track of the current class and current constant pool.

When the virtual machine encounters instructions that operate on data stored in the stack frame, it performs those operations on the current frame.

When a thread invokes a Java method, the virtual machine creates and pushes a new frame onto the thread's Java stack. This new frame then becomes the current frame.

As the method executes, it uses the frame to store parameters, local variables, intermediate computations, and other data.

A method can complete in either of two ways. If a method completes by returning, it is said to have normal completion.

If it completes by throwing an exception, it is said to have abrupt completion.

When a method completes, whether normally or abruptly, the Java virtual machine pops and discards the method's stack frame.

The frame for the previous method then becomes the current frame.

All the data on a thread's Java stack is private to that thread. There is no way for a thread to access or alter the Java stack of another thread.

Because of this, you need never worry about synchronizing multi- threaded access to local variables in your Java programs.

When a thread invokes a method, the method's local variables are stored in a frame on the invoking thread's Java stack.

Only one thread can ever access those local variables: the thread that invoked the method.

Like the method area and heap, the Java stack and stack frames need not be contiguous in memory. Frames could be allocated on a contiguous stack, or they could be allocated on a heap, or some combination of both. The actual data structures used to represent the Java stack and stack frames is a decision of implementation designers. Implementations may allow users or programmers to specify an initial size for Java stacks, as well as a maximum or minimum size.

#### -----The Stack Frame

The stack frame has three parts: local variables, operand stack, and frame data. The sizes of the local variables and operand stack, which are measured in words, depend upon the needs of each individual method.

These sizes are determined at compile time and included in the class file data for each method. The size of the frame data is implementation dependent.

When the Java virtual machine invokes a Java method, it checks the class data to determine the number of words required by the method in the local variables

and operand stack. It creates a stack frame of the proper size for the method and pushes it onto the Java stack.

#### -----Local Variables

The local variables section of the Java stack frame is organized as a zero-based array of words.

Instructions that use a value from the local variables section provide an index into the zero-based array.

Values of type int, float, reference, and returnAddress occupy one entry in the local variables array. Values of type byte, short, a

nd char are converted to int before being stored into the local variables. Values of type long and double occupy two consecutive entries in the array.

To refer to a long or double in the local variables, instructions provide the index of the first of the two consecutive entries occupied by the value.

For example, if a long occupies array entries three and four, instructions would refer to that long by index three. All values in the local variables are word-aligned.

Dual-entry longs and doubles can start at any index.

The local variables section contains a method's parameters and local variables. Compilers place the parameters into the local variable array first, in the order in which they are declared

## MEMORY MODEL

<http://coding-geek.com/jvm-memory-model/>

<https://dzone.com/articles/understanding-the-java-memory-model-and-the-garbag>

<https://dzone.com/articles/java-memory-model-programmer%E2%80%99s>

<https://www.journaldev.com/2856/java-jvm-memory-model-memory-management-in-java>

<https://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html>

<https://blog.codecentric.de/en/2010/01/the-java-memory-architecture-1-act/>

<https://dzone.com/articles/java-memory-architecture-model-garbage-collection>

<https://blog.codecentric.de/en/2010/01/the-java-memory-architecture-1-act/>

### 302) what are data storage type?

Runtime data area in JVM can be divided as below,

1) Method Area : Storage area for compiled class files.

(One per JVM instance)

The runtime constant pool is a subset of the method area which "stores per-class structures such as the runtime constant pool, field and method data, and the code for methods and constructors, including the special methods used in class and instance initialization and interface type initialization".

2) Heap : Storage area for Objects.

(One per JVM instance)

3) Java stack: Storage are for local variables, results of intermediate operations and reference variables

(One per thread)

4) PC Register : Stores the address of the next instruction to be executed if the next instruction is native method then the value in pc register will be undefined.

(One per thread)

5) Native method stacks : Helps in executing native methods( methods written in languages other than java).

(One per thread)

### 303) What is difference between Heap and Stack Memory?

Major difference between Heap and Stack memory are as follows:

- Heap memory is used by all the parts of the application whereas stack memory is used only by one thread of execution.
- Whenever an object is created, it's always stored in the Heap space and stack memory contains the reference to it. Stack memory only contains local primitive variables and reference variables to objects in heap space.
- Memory management in stack is done in LIFO manner whereas it's more complex in Heap memory because it's used globally.

For a detailed explanation with a sample program, read [Java Heap vs Stack Memory](#).

## PROGRAMMING QUESTION

### 2. What will be the output of following programs?

#### 0. static method in class

```
1. package com.journaldev.util;
2.
3. public class Test {
4.
5.     public static String toString() {
6.         System.out.println("Test toString called");
7.         return "";
8.     }
9.
10.    public static void main(String args[]){
11.        System.out.println(toString());
12.    }
13. }
```

**Answer:** The code won't compile because we can't have an `Object` class method with static keyword. Note that `Object` class has `toString()` method. You will get compile time error as "This static method cannot hide the instance method from `Object`". The reason is that static method belongs to class and since every class base is `Object`, we can't have same method in instance as well as in class. You won't get this error if you change the method name from `toString()` to something else that is not present in super class `Object`.

#### 14. static method invocation

```
15. package com.journaldev.util;
16.
17. public class Test {
18.
19.     public static String foo(){
20.         System.out.println("Test foo called");
21.         return "";
22.     }
23.
24.    public static void main(String args[]){
25.        Test obj = null;
26.        System.out.println(obj.foo());
27.    }
```

```
27.         }  
28.     }
```

**Answer:** Well this is a strange situation. We all have seen `NullPointerException` when we invoke a method on object that is NULL. But here this program will work and prints "Test foo called".

The reason for this is the java compiler code optimization. When the java code is compiled to produced byte code, it figures out that `foo()` is a static method and should be called using class. So it changes the method call `obj.foo()` to `Test.foo()` and hence no `NullPointerException`.

I must admit that it's a very tricky question and if you are interviewing someone, this will blow his mind off. 😊

<https://www.journaldev.com/2366/core-java-interview-questions-and-answers#java-oops>

## Write a method to check if input String is Palindrome?

A String is said to be Palindrome if it's value is same when reversed. For example "aba" is a Palindrome String.

String class doesn't provide any method to reverse the String but `StringBuffer` and `StringBuilder` class has reverse method that we can use to check if String is palindrome or not.

```
private static boolean isPalindrome(String str) {  
    if (str == null)  
        return false;  
    StringBuilder strBuilder = new StringBuilder(str);  
    strBuilder.reverse();  
    return strBuilder.toString().equals(str);  
}
```

Sometimes interviewer asks not to use any other class to check this, in that case we can compare characters in the String from both ends to find out if it's palindrome or not.

```
private static boolean isPalindromeString(String str) {  
    if (str == null)  
        return false;  
    int length = str.length();  
    System.out.println(length / 2);  
    for (int i = 0; i < length / 2; i++) {  
  
        if (str.charAt(i) != str.charAt(length - i - 1))  
            return false;  
    }  
    return true;  
}
```



## Write a method that will remove given character from the String?

We can use `replaceAll` method to replace all the occurrence of a String with another String. The important point to note is that it accepts String as argument, so we will use `Character` class to create String and use it to replace all the characters with empty String.

```
private static String removeChar(String str, char c) {  
    if (str == null)  
        return null;  
    return str.replaceAll(Character.toString(c), "");  
}
```

## Write a program to print all permutations of String?

This is a tricky question and we need to use recursion to find all the permutations of a String, for example "AAB" permutations will be "AAB", "ABA" and "BAA". We also need to use Set to make sure there are no duplicate values. Check this post for complete program to [find all permutations of String](#).

## Write a function to find out longest palindrome in a given string?

A String can contain palindrome strings in it and to find longest palindrome in given String is a programming question. Check this post for complete program to find longest [palindrome in a String](#).

## String Programming Questions

1. What is the output of below program?

```
2. package com.journaldev.strings;  
3.  
4. public class StringTest {  
5.  
6.     public static void main(String[] args) {  
7.         String s1 = new String("pankaj");  
8.         String s2 = new String("PANKAJ");  
9.         System.out.println(s1 = s2);  
10.     }  
11.  
12. }
```

It's a simple yet tricky program, it will print "PANKAJ" because we are assigning s2 String to s1. Don't get confused with == comparison operator.

13. What is the output of below program?

```

14. package com.journaldev.strings;
15.
16. public class Test {
17.
18.     public void foo(String s) {
19.         System.out.println("String");
20.     }
21.
22.     public void foo(StringBuffer sb){
23.         System.out.println("StringBuffer");
24.     }
25.
26.     public static void main(String[] args) {
27.         new Test().foo(null);
28.     }
29.
30. }

```

The above program will not compile with error as "The method foo(String) is ambiguous for the type Test". For complete clarification read [Understanding the method X is ambiguous for the type Y error](#).

31. What is the output of below code snippet?

```

32. String s1 = new String("abc");
33. String s2 = new String("abc");
34. System.out.println(s1 == s2);

```

It will print **false** because we are using *new* operator to create String, so it will be created in the heap memory and both s1, s2 will have different reference. If we create them using double quotes, then they will be part of string pool and it will print true.

35. What will be output of below code snippet?

```

36. String s1 = "abc";
37. StringBuffer s2 = new StringBuffer(s1);
38. System.out.println(s1.equals(s2));

```

It will print false because s2 is not of type String. If you will look at the equals method implementation in the String class, you will find a check using **instanceof** operator to check if the type of passed object is String? If not, then return false.

39. What will be output of below program?

```

40. String s1 = "abc";
41. String s2 = new String("abc");
42. s2.intern();
43. System.out.println(s1 ==s2);

```

It's a tricky question and output will be **false**. We know that intern() method will return the String object reference from the string pool, but since we didn't assigned it back to s2, there is no change in s2 and hence both s1 and s2 are having different reference. If we change the code in line 3 to `s2 = s2.intern();` then output will be true.

44. How many String objects got created in below code snippet?

```

45. String s1 = new String("Hello");
46. String s2 = new String("Hello");

```

Answer is 3.

First – line 1, "Hello" object in the string pool.

Second – line 1, new String with value "Hello" in the heap memory.

Third – line 2, new String with value “Hello” in the heap memory. Here “Hello” string from string pool is reused.

## Reference Links

[https://www.buggybread.com/2015/02/java-interview-questions-and-answers-on\\_22.html](https://www.buggybread.com/2015/02/java-interview-questions-and-answers-on_22.html)

<http://java-questions.com/Cloning-interview-questions.html>

<https://www.artima.com/intv/bloch13.html>