**Topic 1: Main Method**

What if I create main method for java as

public static void Test(String ars[]) {

}

Code will be compile but will not run

**Topic 2: Getter and Setter**

**Getter & Setter**

* The idea of getters and setters are to control access to variables in a class.
* That way, if the value needs to be changed internally to be represented in a different way, you can do so without breaking any code outside the class.

**For example,** let's say you had a class with a distance variable, and it was measured in inches. A few months pass, you're using this class in a lot of places and you suddenly realize you needed to represent that value in centimeters. If you didn't use a getter and a setter, you would have to track down every use of the class and convert there. If you used a getter and a setter, you can just change those methods and everything that uses the class won't break.

**Topic 3: Constructor**

* Constructors are the only way to set final instance variables.
* A class with private constructor cannot be sub classed.
* If your class is a subclass and the base class doesn't have a default constructor, then you need a constructor in your class to call the super class constructor.
* In Java you don't manage the memory yourself (at least not directly anyway) so this is the purpose of the constructor.
* Essentially a constructor is just a special method that implicitly returns an object of its containing type. You should generally use constructors for creating objects

**Example**: Created a specific rule that is again enforced by the system: a new object of class Person cannot be created unless you give a first name and last name:

**Topic 4: Immutable Class**

**Q1) What is an immutable class?**

**Ans)** Immutable class is a class which once created, it’s contents cannot be changed. Immutable objects are the objects whose state cannot be changed once constructed. e.g. String class

**Q2) How to create an immutable class?**

**Ans)** To create an immutable class following steps should be followed:

1. Create a final class.
2. Set the values of properties using constructor only.
3. Make the properties of the class final and private
4. Do not provide any setters for these properties.
5. If the instance fields include references to mutable objects, don't allow those objects to be changed:
   1. Don't provide methods that modify the mutable objects.
   2. Don't share references to the mutable objects. Never store references to external, mutable objects passed to the constructor; if necessary, create copies, and store references to the copies. Similarly, create copies of your internal mutable objects when necessary to avoid returning the originals in your methods.

public final class FinalPersonClass {

private final String name;

private final int age;

public FinalPersonClass(final String name, final int age) {

this.name = name;

this.age = age;

}

public int getAge() {

return age;

}

public String getName() {

return name;

}

}

**Q3) Immutable objects are automatically thread-safe –true/false?**

**Ans)** True. Since the state of the immutable objects cannot be changed once they are created they are automatically synchronized/thread-safe.

**Q4) Which classes in java are immutable?**

**Ans)** All wrapper classes in java.lang are immutable –   
String, Integer, Boolean, Character, Byte, Short, Long, Float, Double, BigDecimal, BigInteger

**Q5) What are the advantages of immutability?**

Ans)

* Immutable objects are automatically thread-safe, the overhead caused due to use of synchronisation is avoided.
* Once created the state of the immutable object cannot be changed so there is no possibility of them getting into an inconsistent state.
* The references to the immutable objects can be easily shared or cached without having to copy or clone them as there state cannot be changed ever after construction.
* The best use of the immutable objects is as the keys of a map.

**Topic 5: Abstract Class/Method**

1. It can have abstract and non-abstract methods (method with body).
2. Abstract class provide 0 to 100 % abstraction
3. It cannot be instantiated.
4. If you are extending any abstract class that have abstract method, you must either provide the implementation of the method or make this class abstract.
5. If there is any abstract method in a class, that class must be abstract.
6. Java Abstract class can implement interfaces without even providing the implementation of interface methods.
7. We can run abstract class in java like any other class if it has main() method.

**Ques Why Abstract Class is used in Java?**

1. Its providing Data hiding and forces the programmer to implement all the methods in Abstract class.
2. The abstract class will hold common functionality for all classes that extend it.
3. **Example**: All animals move and breathe and reproduce so these can be put into the Animal Class.
4. Abstract classes permit providing a partial set of default implementations of methods in a class. Since they're incomplete, they can't be instantiated and used as they stand, but they can be subclassed to add the missing details in a way that's specific to that particular implementations, and those subclasses can be instantiated
5. Without abstract classes, you would have to provide dummy implementations of the methods you intend to override ... which could be done, but then there'd be the risk of forgetting to implement one of them. Having some methods remain entirely abstract ensures that the real implementations have to fill in the gaps, or continue to be abstract themselves and force their descendents to do so.
6. When you make things implement the same abstract class, you are saying, these things are alike at a high level, but they vary in the particulars of how they do things. An abstract method is how you say, "Here's something that all the things that extend this class have to do, but they each get to specify how exactly they will do it." example Audi>>class drive method and Volvo also has drive method

Ques **Can we declare a final method in abstract class?**

Ans. It will give "illegal combination of modifiers: abstract and final" error. ... If we extend the abstract class then only it has meaning, so whatever methods we declare or defined in Abstract call it will over ride in subclass. So we can declare a method as final in Abstract class, and it will be over ridden in subclass.

Ques **Can an abstract class have a static method?**

Ans. However, for the same reason, you can't declare a static method to be abstract. Normally, the compiler can guarantee that an abstract method will have a real implementation any time that it is called, because you can't create an instance of an abstract class. ... You cannot create an abstract static method

**Topic 6: Singleton class**

* We can make constructor as private. So that We cannot create an object outside of the class.
* This property is useful to create singleton class in java.
* Singleton pattern helps us to keep only one instance of a class at any time.
* The purpose of singleton is to control object creation by keeping private constructor.

**For example**, if you have a license for only one connection for your database or your JDBC driver has trouble with multithreading, the Singleton makes sure that only one connection is made or that only one thread can access the connection at a time.

It is used in scenarios when a user wants to restrict instantiation of a **class** to only one object. This is helpful usually when a single object is required to coordinate actions across a system. The **singleton pattern** is used in programming languages such as **Java** and .NET to **define** a global variable.

Different approaches of Singleton pattern implementation and design concerns with the implementation:

* 1. Eager Initialization.
  2. Static Block initialization
  3. Lazy Initialization
  4. Thread Safe Initialization
  5. Enum Singletone

**Eager Initialization:** the instance of Singleton Class is created at the time of class loading, this is the easiest method to create a singleton class but it has a drawback that instance is created even though client application might not be using it.

**Static Block initialization**: [Static block](https://www.journaldev.com/1365/static-keyword-in-java) initialization implementation is similar to eager initialization, except that instance of class is created in the static block that provides option for [exception handling](https://www.journaldev.com/1696/exception-handling-in-java).

Both eager initialization and static block initialization creates the instance even before it’s being used and that is not the best practice to use.

**Lazy** **Initialization:** The above implementation works fine in case of single threaded environment but when it comes to multithreaded systems, it can cause issues if multiple threads are inside the if loop at the same time. It will destroy the singleton pattern and both threads will get the different instances of singleton class.

The Singleton design pattern addresses all of these **concerns**. With the Singleton design pattern you can:

* Ensure that only one instance of a class is created
* Provide a global point of access to the object
* Allow multiple instances in the future without affecting a singleton class's clients

Sometimes it's appropriate to have exactly one instance of a class: window managers, print spoolers, and filesystems are prototypical examples. Typically, those types of objects—known as singletons—are accessed by disparate objects throughout a software system, and therefore require a global point of access

*I've a properties file containing some keys value pairs, which is need across the application, that is why I was thinking about a singleton class. This class will load the properties from a file and keep it and you can use it from anywhere in the application*

**Topic 7: Interface**

* 1. An **interface in java** is a blueprint of a class. It has static constants and abstract methods.
  2. The interface in java is **a mechanism to achieve abstraction**. There can be only abstract methods in the java interface not method body

**Ques Why use Interface?**

* It is used to achieve abstraction.
* By interface, we can support the functionality of multiple inheritance.
* It can be used to achieve loose coupling.

**Marked or Tagged interface**

An interface that have no member is known as marker or tagged interface. For example: Serializable, Cloneable, Remote etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

public interface Serializable{

}

**Nested Interface**

An interface can have another interface i.e. known as nested interface.

interface printable{

 void print();

 interface MessagePrintable{

   void msg();

 }

}

**Inner interface** is also called **nested interface**, which means declare an **interface** inside of another **interface**. For example, the Entry **interface** is declared in the Map **interface**. There are several compelling reasons for using **inner interface**: It is a way of logically grouping interfaces that are only used in one place.

**Ques Why Use Inner Interface?**

There are several compelling reasons for using inner interface:

* It is a way of logically grouping interfaces that are only used in one place.
* It increases encapsulation.
* Nested interfaces can lead to more readable and maintainable code.

One example of inner interface used in java standard library is java.util.Map and Java.util.Map.Entry. Here java.util.Map is used also as a namespace. Entry does not belong to the global scope, which means there are many other entities that are Entries and are not necessary Map’s entries. This indicates that Entry represents entries related to the Map.

**Ques How Inner Interface Works?**

To figure out how inner interface works, we can compare it with nested classes. Nested classes can be considered as a regular method declared in outer class. Since a method can be declared as static or non-static, similarly nested classes can be static and non-static. Static class is like a static method, it can only access outer class members through objects. Non-static class can access any member of the outer class.

Ques **Can we define variables in interface in Java?**

**Ans : interface** doesn't allow you to **declare** any instance **variables**. Using a **variable declared** in an **interface** as an instance **variable** will return a compile time error. You **can declare** a constant **variable**, using static final which is different from an instance **variable**.

Ques [**Why are interface variables static and final by default?**](https://stackoverflow.com/questions/2430756/why-are-interface-variables-static-and-final-by-default)

Ans: Interface variables are static because Java interfaces cannot be instantiated in their own right; the value of the variable must be assigned in a static context in which no instance exists. The final modifier ensures the value assigned to the interface variable is a true constant that cannot be re-assigned by program code.

**Ques Can you define private and protected modifiers for variables in interfaces?**

Ans: You should use an abstract class instead of an **interface** if you want to **declare** its fields. as you have **declare variable** in test **interface** with **private and protected** it **will** give error. if you do not specify the **modifier** the compiler **will** add public static final automatically.

**Note:**

* 1. Java 8, we can have method body in interface. But we need to make it default method.
  2. Java 8, we can have static method in interface.

**Topic 8: Coupling**

1. Tight coupling is when a group of classes are highly dependent on one another.

2. This scenario arises when a class assumes too many responsibilities, or when one concern is spread over many classes rather than having its own class.

3. Loose coupling is achieved by means of a design that promotes single-responsibility and separation of concerns.

4. A loosely-coupled class can be consumed and tested independently of other (concrete) classes.

5. Interfaces are a powerful tool to use for decoupling.

6. how often do changes in class A force related changes in class B

6. Example of tight coupling:

class CustomerRepository

{

private readonly Database database;

public CustomerRepository(Database database)

{

this.database = database;

}

public void Add(string CustomerName)

{

database.AddRow("Customer", CustomerName);

}

}

class Database

{

public void AddRow(string Table, string Value)

{

}

}

7. Example of loose coupling:

class CustomerRepository

{

private readonly IDatabase database;

public CustomerRepository(IDatabase database)

{

this.database = database;

}

public void Add(string CustomerName)

{

database.AddRow("Customer", CustomerName);

}

}

interface IDatabase

{

void AddRow(string Table, string Value);

}

class Database : IDatabase

{

public void AddRow(string Table, string Value)

{

}

}

Explanation of the concept without any code:

**Tight coupling**

Think of your skin. It's stuck to your body. It fits like a glove. But what if you wanted to change your skin colour from say white to black? Can you imagine just how painful it would be to peel off your skin, dye it, and then to paste it back on etc? Changing your skin is difficult because it is tightly coupled to your body. You just can't make changes easily. You would have to fundamentally redesign a human being in order to make this possible. *Key Point #1* In other words, **if you want to change the skin, you would also HAVE TO change the design of your body** as well because the two are joined together - they are tightly coupled. God was not a good object oriented programmer.

**Loose coupling**

Now think of getting dressed in the morning. You don't like blue? No problems: you can put a red shirt on instead. You can do this easily and effortlessly because the shirt is not really connected to your body the same way as your skin. *The shirt doesn't know or care about what body it is going on*. In other words, you can change your clothes, without really changing your body. That's key point #2.***If you change your shirt, then you are not forced to change your body*** - when you can do that, then you have loose coupling. When you can't do that, then you have tight coupling.

That's the basic concept in a nutshell.

**Why is all of this important?**

It's important because software changes all the time. Generally speaking you want to be able to easily modify your code. If somebody wants their car in black, you shouldn't have to reconfigure the engine in order to make that change. That's why you want things loosely coupled. Because then it's easy to change.

**Topic 9: Abstraction**

**1. Abstraction** is the process of **abstraction in Java** is used to hide certain details and only show the essential features of the object

2. Another way, it shows only important things to the user and hides the internal details for example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.

3. Ways to achieve Abstraction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

4.  An abstract class can have data member, abstract method, method body, constructor and even main() method.

5. Advantages of Abstraction By using abstraction, we can separate the things that can be grouped to another type.  
  
Frequently changing properties and methods can be grouped to a separate type so that the main type need not undergo changes. This adds strength to the OOAD principle -"Code should be open for Extension but closed for Modification".

6. Summary:

-    Use abstraction if you know something needs to be in class but implementation of that varies.  
-     In Java you cannot create instance of abstract class , its compiler error.  
-    abstract is a keyword in java.  
-    A class automatically becomes abstract class when any of its method declared as abstract.  
-     abstract method doesn't have method body.  
-    Variable cannot be made abstract, its only behavior or methods which would be abstract.  
-    If a class extends an abstract class or interface it has to provide implementation to all its abstract method to be a concrete class. Alternatively this class can also be abstract.

**Topic 10: Encapsulation**

**Encapsulation** is one of the four fundamental OOP concepts. The other three are inheritance, polymorphism, and abstraction. **Encapsulation in Java** is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. ... Declare the variables of a class as private.

**Ques** **How do we achieve encapsulation in Java?**

**Encapsulation in java is achieved by using private members and public methods.**

1. Make members of a class as private.
2. Define public setter and getter methods to modify and view the variables' values and access them outside the class only through getters and setters.

it is also known as **data hiding**.

## Benefits of Encapsulation

* The fields of a class can be made read-only or write-only.
* A class can have total control over what is stored in its fields.
* The users of a class do not know how the class stores its data. A class can change the data type of a field and users of the class do not need to change any of their code.

**Encapsulation is:**

* Binding the data with the code that manipulates it.
* It keeps the data and the code safe from external interference

**Example** a power steering mechanism of a car. Power steering of a car is a complex system, which internally have lots of components tightly coupled together, they work synchronously to turn the car in the desired direction. It even controls the power delivered by the engine to the steering wheel. But to the external world there is only one interface is available and rest of the complexity is hidden. Moreover, the steering unit in itself is complete and independent. It does not affect the functioning of any other mechanism.

Similarly, same concept of encapsulation can be applied to code. Encapsulated code should have following characteristics:

* Everyone knows how to access it.
* Can be easily used regardless of implementation details.
* There shouldn’t any side effects of the code, to the rest of the application.

The idea of encapsulation is to keep classes separated and prevent them from having tightly coupled with each other.

**A live example** of encapsulation is the class of java.util.Hashtable. User only knows that he can store data in the form of key/value pair in a Hashtable and that he can retrieve that data in the various ways. But the actual implementation like, how and where this data is actually stored, is hidden from the user. User can simply use Hashtable wherever he wants to store Key/Value pairs without bothering about its implementation.

**Advantages of Encapsulation**:

* **Data Hiding:** The user will have no idea about the inner implementation of the class. It will not be visible to the user that how the class is storing values in the variables. He only knows that we are passing the values to a setter method and variables are getting initialized with that value.
* **Increased Flexibility:** We can make the variables of the class as read-only or write-only depending on our requirement. If we wish to make the variables as read-only then we have to omit the setter methods like setName(), setAge() etc. from the above program or if we wish to make the variables as write-only then we have to omit the get methods like getName(), getAge() etc. from the above program
* **Reusability:** Encapsulation also improves the re-usability and easy to change with new requirements.
* **Testing code is easy:** Encapsulated code is easy to test for unit testing.

## Benefits of encapsulation

* Provides abstraction between an object and its clients.
* Protects an object from unwanted access by clients.
* Example: A bank application forbids (restrict) a client to change an Account's balance.

## Advantage of Encapsulation in Java and OOPS

Here are few advantages of using **Encapsulation** while writing code in Java or any Object oriented programming language:

1. Encapsulated Code is more flexible and easy to change with new requirements.

2. Encapsulation in Java makes unit testing easy.

3. Encapsulation in Java allows you to control who can access what.

4. Encapsulation also helps to write immutable class in Java which is a good choice in multi-threading environment.

5. Encapsulation reduces coupling of modules and increases cohesion inside a module because all piece of one thing is encapsulated in one place.

6. Encapsulation allows you to change one part of code without affecting other parts of code.

### Important points about encapsulation in Java.

1. "Whatever changes encapsulate it" is a famous design principle.

2. Encapsulation helps in loose coupling and high cohesion of code.

3. Encapsulation in Java is achieved using access modifier private, protected and public.

4. Factory pattern , Singleton pattern in Java makes good use of Encapsulation.

**Topic 11 Throws (Exception)**

## Use of throws keyword in Java

1. The [**throws keyword**](https://beginnersbook.com/2013/04/difference-between-throw-and-throws-in-java/) is used in method declaration, in order to explicitly specify the exceptions that a particular method might throw. When a method declaration has one or more exceptions defined using throws clause then the method-call must handle all the defined exceptions.

2. When defining a method you must include a throws clause to [**declare those exceptions**](https://beginnersbook.com/2013/04/java-exception-handling/) that might be thrown but doesn’t get caught in the method.

3. If a method is using throws clause along with few exceptions then this implicitly tells other methods that – “ If you call me, you must handle these exceptions that I throw”.

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

### Ques Which exception should be declared

**Ans)** checked exception only, because:

* **unchecked Exception:** under your control so correct your code.
* **error:** beyond your control e.g. you are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

### Advantage of Java throws keyword

Now Checked Exception can be propagated (forwarded in call stack).

It provides information to the caller of the method about the exception.

**Note:** If you are calling a method that declares an exception, you must either caught or declare the exception

**There are two cases:**

1. **Case1:** You caught the exception i.e. handle the exception using try/catch.
2. **Case2:** You declare the exception i.e. specifying throws with the method.

### You declare the exception

* A) In case you declare the exception, if exception does not occur, the code will be executed fine.
* B) In case you declare the exception if exception occurs, an exception will be thrown at runtime because throws does not handle the exception.

# [When to use throws in a Java method declaration?](https://stackoverflow.com/questions/4392446/when-to-use-throws-in-a-java-method-declaration)

Ans--- ?

**Throws** keyword is used to declare that a method may throw one or some exceptions. The caller has to catch the exceptions (catching is optional if the exceptions are of type unchecked exceptions).

* A concrete method can declare **throws** clause if only if its body throws checked exceptions. Otherwise a compile error occurs.
* An interface’s method can declare **throws** clause freely.
* The **throws** clause can declare exceptions which are super types of the exception thrown by the **throw** statements, but not sub types.

### Ques Can we rethrow an exception?

Yes, by throwing same exception in catch block.

Any method that is capable of causing exceptions must list all the exceptions possible during its execution, so that anyone calling that method gets a prior knowledge about which exceptions are to be handled. A method can do so by using the **throws** keyword.

**Difference between throw and throws**

|  |  |
| --- | --- |
| **Throw** | **throws** |
| throw keyword is used to throw an exception explicitly. | throws keyword is used to declare an exception possible during its execution. |
| throw keyword is followed by an instance of Throwable class or one of its sub-classes. | throws keyword is followed by one or more Exception class names separated by commas. |
| throw keyword is declared inside a method body. | throws keyword is used with method signature (method declaration). |
| We cannot throw multiple exceptions using throw keyword. | We can declare multiple exceptions (separated by commas) using throws keyword. |

## Topic 12: Cohesion

## Cohesion is the OO principle most closely associated with making sure that a class is designed with a single, well-focused purpose

**Benefits of Higher Cohesion:**

* Highly cohesive classes are much easier to maintain and less frequently changed.
* Such classes are more usable than others as they are designed with a well-focused purpose.
* Cohesion is all about how a single class is designed.
* The term cohesion is used to indicate the degree to which a class has a single, well-focused purpose.
* Keep in mind that cohesion is a subjective concept.
* The more focused a class is, the higher its cohesiveness - a good thing.

**Benefit or Advantages of Cohesion**

* 1. The key benefit of high cohesion is that such classes are typically much easier to maintain (and less frequently changed) than classes with low cohesion.
  2. Another benefit of high cohesion is that classes with a well-focused purpose tend to be more reusable than other classes.

**Cohesion** is the degree to which components of a class belong together to fit a particular role. What we want to avoid is low cohesion where a class incorporates several different aspects. A class that tries to do many things comes with higher maintenance and lower reusability.

**Cohesion**  
1. While coupling has to do with how classes interact with each other, cohesion is all about how a single class is designed.

2. The term cohesion is used to indicate the degree to which a class has a single, well-focused purpose. Keep in mind that cohesion too is a subjective concept.

3. The more focused a class is, the higher its cohesiveness.

4. The key benefit of high cohesion is that such classes are typically much easier to maintain (and less frequently changed) than classes with low cohesion.

5. Another benefit of high cohesion is that classes with a well-focused purpose tend to be more reusable than other classes.

6. Let’s take a look at a **pseudocode** example:  
  
class SalesReport {  
void connectToDb(){ }  
void generateSalesReport() { }  
void saveAsFile() { }  
void print() { }  
}  
  
**Example** Now imagine your manager comes along and says, “Hey you know that accounting application we’re working on? The clients just decided that they’re also going to want to generate a revenue projection report, oh and they want to do some inventory reporting also. They do like our reporting features however, so make sure that all of these reports will let them choose a database, choose a printer, and save generated reports to data files...”   
  
Rather than putting all the printing code into one report class, we probably would have been better off with the following design right from the start:  
  
class SalesReport {  
Options getReportingOptions() { }  
void generateSalesReport(Options o) { }  
}  
  
class ConnectToDb {  
DBconnection getDb() { }  
}  
  
class PrintStuff {  
PrintOptions getPrintOptions() { }  
}  
  
class FileSaver {  
SaveOptions getFileSaveOptions() { }  
}  
  
This design is much more cohesive. Instead of one class that does everything, we’ve broken the system into four main classes, each with a very specific, or cohesive, role. Because we’ve built these specialized, reusable classes, it’ll be much easier to write a new report, since we’ve already got the database connection class, the printing class, and the file saver class, and that means they can be reused by other classes that might want to print a report. Again, as in Coupling, you may not get too many questions about cohesion but if you are (un)lucky you may get one or two…

**Cohesion** means that the whole of a class sticks together (well, roughly). A class should be responsible for itself, should do one thing and as far as possible do everything for that one thing. For example: A Car class should remember its make, colour, speed. It is responsible for changing speed; the speedUp() and slowDown() methods should be in the Car class; no other class should make your Car go faster or slower.  
  
**Cohesion**

* 1. Cohesion is another OO concept like coupling, however cohesion refers to the degree in which a class has a single, well defined purpose.
  2. **Example** a class called Car, it would maintain state such as numberOfWindows, numberOfWheels, and have behaviour (methods) such as drive(), brake(), stop() etc. That would indicate that the class is cohesive, meaning that it serves a single, defined purpose.
  3. Now, exploring a little, if the Car class suddenly decided to have methods such as switchOnTheTV(), or hangTheWashingOut(), then its not really doing Car-like behaviour. If a class is trying to do more than one thing, it is considered a low cohesive class.

**Cohesion**

* Cohesion refers to the degree in which a class has a single, well defined purpose
* High cohesion is GOOD as a class does only what it should
* Low cohesion is BAD because a class is trying to solve everything and not just its specific purposes

**Topic 13 Wrapper classed in java**

1. All the wrapper classes (**Integer**, Long, Byte, Double, Float, Short) are subclasses of the abstract class Number.
2. The **object** of the wrapper class contains or wraps its respective primitive data type. Converting primitive data types into **object** is called **boxing**, and this is taken care by the compiler.

**Ques** **Are wrapper objects immutable?**

Ans Firstly, **all** of the **java**.lang package **wrapper classes** are **immutable**: Boolean, Byte, Character, Double, Float, Integer, Long, Short, String

**Ques Is string is a wrapper class in Java?**

Ans That's different for the primitive **wrappers**: they have been designed only to wrap a primitive, to adapt a java primitive to java.lang.Object . No. **String** is not a **wrapper class**, simply because there is no parallel primitive type that it wraps

**Need of Wrapper Classes**

1. They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).
2. The classes in java.util package handles only objects and hence wrapper classes help in this case also.
3. Data structures in the Collection framework, such as [ArrayList](http://www.geeksforgeeks.org/arraylist-in-java/" \t "_blank) and [Vector](http://www.geeksforgeeks.org/vector-vs-arraylist-java/), store only objects (reference types) and not primitive types.
4. An object is needed to support synchronization in multithreading.

## How to use wrapper class methods

All the wrapper class methods are static in nature so we need to call these method using class.methodName().

* for Integer: int x=Integer.parseInt(String);
* for float: float x=Float.parseFloat(String);
* for double: double x=Double.parseDouble(String);

**Wrapper Class**:

1. java is not 100% oops because we cannot make object of primitives data types [int] so to overcome this we need something who make object of these data types java introudce concept of wrapper class .
2. we can make object of data types.
3. Wrapper classes are used to convert any data type into an object. The primitive data types are not objects; they do not belong to any class; they are defined in the language itself. Sometimes, it is required to convert data types into objects in Java.
4. Primitive types can’t be null but wrapper classes can be null.
5. Wrapper classes can be used to achieve polymorphism.

**Several possible reasons:**

* So that a null value is possible
* To include in a Collection
* To treat generically / polymorphically as an Object along with other Objects

**Wrapper Class** Java is an object-oriented language and can view everything as an object. A simple file can be treated as an object , an address of a system can be seen as an object , an image can be treated as an object (with java.awt.Image) and a simple data type can be converted into an object (with wrapper classes). This tutorial discusses wrapper classes. Wrapper classes are used to convert any data type into an object.

**Features of the Java wrapper Classes.**

**1)** Wrapper classes convert numeric strings into numeric values.

**2)** The way to store primitive data in an object.

**3)** The valueOf() method is available in all wrapper classes except Character

**4)** All wrapper classes have typeValue() method. This method returns the value of the object as its primitive type.

### Methods and Examples

A really cool feature of the wrapper classes is that they include their own methods that can be used with each wrapper class. For example, you can show the double value of an integer variable, compare values, or even convert values to a string!

Let's look at a code example for the heavy hitter Integer.

#### Int Primitive = Integer Wrapper

The wrapper class for the int data type is the Integer class. Let's expand upon the previous example of the Integer and use one of the methods to convert it to a Double. The method to do this is doubleValue(), and the code looks like:

int thisInt=10;

Integer myInteger=new Integer(thisInt);

Double doubleMe=myInteger.dobuleValue();

Now that we've created a new instance of Integer, let's look at some of the methods that come with these powerful tools in the Java toolbox.

**Topic 14: Overhiding**

If a subclass defines a static method with the same signature as a static method in the superclass, then the method in the subclass hides the one in the superclass.

The distinction between hiding a static method and overriding an instance method has important implications:

## The version of the overridden instance method that gets invoked is the one in the subclass.

## The version of the hidden static method that gets invoked depends on whether it is invoked from the superclass or the subclass.

# [Why doesn't Java allow overriding of static methods?](https://stackoverflow.com/questions/2223386/why-doesnt-java-allow-overriding-of-static-methods)

Overriding depends on having an instance of a class. The point of polymorphism is that you can subclass a class and the objects implementing those subclasses will have different behaviors for the same methods defined in the superclass (and overridden in the subclasses). A static method is not associated with any instance of a class so the concept is not applicable.

**Hiding**: Parent class methods that are static are not part of a child class (although they are accessible), so there is no question of overriding it. Even if you add another static method in a subclass, identical to the one in its parent class, this subclass static method is unique and distinct from the static method in its parent class.

Static methods cannot be overridden because they are not dispatched on the object instance at runtime. The compiler decides which method gets called.

Static methods cannot be overridden because they are not part of the object's state

**Topic 15 Data binding**

There are two types of binding

1. static binding (also known as early binding).
2. dynamic binding (also known as late binding).

### static binding

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

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

**Example:**

1. **class** Dog{
2. **private** **void** eat(){System.out.println("dog is eating...");}
4. **public** **static** **void** main(String args[]){
5. Dog d1=**new** Dog();
6. d1.eat();
7. }
8. }

### Dynamic binding

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

1. **class** Animal{
2. **void** eat(){System.out.println("animal is eating...");}
3. }
4. **class** Dog **extends** Animal{
5. **void** eat(){System.out.println("dog is eating...");}
7. **public** **static** **void** main(String args[]){
8. Animal a=**new** Dog();
9. a.eat();
10. }
11. }

**Why binding of static, final and private methods is always a static binding?**  
Static binding is better performance wise (no extra overhead is required). Compiler knows that all such methods **cannot be overridden** and will always be accessed by object of local class. Hence compiler doesn’t have any difficulty to determine object of class (local class for sure). That’s the reason binding for such methods is static.

**Dynamic Binding:**In Dynamic binding compiler doesn’t decide the method to be called. Overriding is a perfect example of dynamic binding. In overriding both parent and child classes have same method

**Important Points**

* private, final and static members (methods and variables) use static binding while for virtual methods (In Java methods are virtual by default) binding is done during run time based upon run time object.
* Static binding uses Type information for binding while Dynamic binding uses Objects to resolve binding.
* Overloaded methods are resolved (deciding which method to be called when there are multiple methods with same name) using static binding while overridden methods using dynamic binding, i.e, at run time.

### Static Binding or Early Binding

The binding which can be resolved at compile time by compiler is known as static or early binding. All the static, private and final methods have always been bonded at [**compile-time**](https://beginnersbook.com/2013/04/runtime-compile-time-polymorphism/) . **Why binding of Static, final and private methods is always a static binding?**You would understand it better after reading dynamic binding. Still let me explain this – Compiler knows that all such methods cannot be overridden and will always be accessed by object of local class. Hence compiler doesn’t have any difficulty to determine object of class (local class for sure). That’s the reason binding for such methods is static.

### Dynamic Binding or Late Binding

When compiler is not able to resolve the call/binding at compile time, such binding is known as Dynamic or late Binding. Overriding is a perfect example of dynamic binding as in overriding both parent and child classes have same method. Thus while calling the overridden method, the compiler gets confused between parent and child class method(since both the methods have same name).

## Static Binding vs Dynamic Binding

Lets discuss the **difference between static and dynamic binding in Java**.

1. Static binding happens at compile-time while dynamic binding happens at runtime.
2. Binding of private, static and final methods always happen at compile time since these methods cannot be overridden. Binding of overridden methods happen at runtime.
3. Java uses static binding for [**overloaded methods**](https://beginnersbook.com/2013/05/method-overloading/) and dynamic binding for overridden methods.

**Topic 16: Marker Interface**

Ques 1. **What is Marker interfaces in Java and why required?**  
**Ans 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 marker interface is Serializable, Clonnable and Remote interface.

### Ques 2 What Marker or Tag interface do in Java?

Ans 1) Looking carefully on marker interface in Java e.g. [**Serializable**](http://javarevisited.blogspot.com/2011/04/top-10-java-serialization-interview.html)**, Clonnable and Remote** it looks they are **used to indicate something to compiler or JVM**. So if JVM sees a Class is Serializable it done some special operation on it, similar way if JVM sees one Class is implement Clonnable it performs some operation to support cloning. Same is true for **RMI and Remote interface**. So in short Marker interface indicate, signal or a command to Compiler or [JVM](http://javarevisited.blogspot.com/2011/12/jre-jvm-jdk-jit-in-java-programming.html).

### Ques 3 Where Should I use Marker interface in Java?

Ans

* 1. Apart from using built in marker interface for making a class Serializable or Clonnable.
  2. One can also develop his own marker interface.
  3. Marker interface is a good way to classify code.
  4. You can create marker interface to logically divide your code and if you have your own tool than you can perform some pre-processing operation on those classes.
  5. Particularly useful for developing API and framework like [Spring](http://javarevisited.blogspot.com/2011/09/spring-interview-questions-answers-j2ee.html) or [Struts](http://javarevisited.blogspot.com/2011/11/struts-interview-questions-answer-j2ee.html).

After introduction of Annotation on Java5, Annotation is better choice than marker interface and JUnit is a perfect example of using Annotation

**Ques 4. Use of marker interface in Java?**

Ans

1. One more use of marker interface in Java can be commenting.
2. A marker interface called Thread Safe can be used to communicate other developers that classes implementing this marker interface gives thread-safe guarantee and any modification should not violate that.
3. Marker interface can also help code coverage or code review tool to find bugs based on specified behavior of marker interfaces.

Again Annotations are better choice @ThreadSafe looks lot better than implementing ThraedSafe marker interface.

**Ques 5. Runnable interface being marker or not?**

Ans

1. Runnable interface is not marker because Runnable interface has the public void run() method declared inside it.
2. A very good example of marker interface is Serializable where the class implements can be used with ObjectOutputStream and ObjectInputStream classes.

**Ques 5. can we create a marker interface or not?**

Ans

1. We can't create marker interface similar to Serializable or Cloneable but we can simulate the functionality by writing extra code around the custom marker interface.
2. Hence, an empty interface in java is called a marker interface. Marker interface is also called tag interface by some java gurus. In java we have the following major marker interfaces as under:

* Searilizable interface
* Cloneable interface
* Remote interface
* ThreadSafe interface

1. The marker interface can be described as a design pattern which is used by many languages to provide run-time type information about the objects.
2. The marker interface provides a way to associate metadata with the class where the language support is not available.

**Ques 6. Create own marker interface**

Ans

We have no. of colleges from which some colleges are of A grade.

1. We have created a AGradeCollegeMarker interface which contains no method and only inform the JVM that it is a A grade college.
2. Every A grade college have to implement AGradeCollegeMarker. In TestCollege class, tester method will print “A grade college.” if object belongs to A grade college.

## Example:

**AGradeCollegeMarker.java**

|  |
| --- |
| **package** com.javawithease.business;    **/\*\***  **\* This is marker interface for A grade college.**  **\* @author javawithease**  **\*/**  **public** **interface** AGradeCollegeMarker {    } |

**College1.java**

|  |
| --- |
| **package** com.javawithease.business;    **/\*\***  **\* This class represents a A grade college.**  **\* @author javawithease**  **\*/**  **public** **class** College1 **implements** AGradeCollegeMarker{  //Do something  } |

**College2.java**

|  |
| --- |
| **package** com.javawithease.business;    **/\*\***  **\* This class represents a, non A grade college.**  **\* @author javawithease**  **\*/**  **public** **class** College2 {  //Do something  } |

**TestCollege.java**

|  |
| --- |
| **package** com.javawithease.business;    **/\*\***  **\* This class is used to test the custom marker interface functionality.**  **\* @author javawithease**  **\*/**  **public** **class** TestCollege {  **static** **void** tester(Object obj){  **if** (obj **instanceof** AGradeCollegeMarker) {  System.out.println("A grade college.");  }  }    **public** **static** **void** main(String args[]){  College1 obj1 = **new** College1();  College2 obj2 = **new** College2();    //test college objects  tester(obj1);  tester(obj2);  }  } |

## Output:

|  |
| --- |
| A grade college. |

**Topic 17: Checked Exception**

1. The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions e.g.IOException, SQLException etc. Checked exceptions are checked at compile-time.
2. Checked exceptions are checked at compile-time. It means if a method is throwing a checked exception then it should handle the exception using try-catch block or it should declare the exception using throws keyword, otherwise the program will give a compilation error. It is named as checked exception because these exceptions are checked at Compile time.
3. **Example:** In this example we are reading the file myfile.txt and displaying its content on the screen. In this program there are three places where an checked exception is thrown as mentioned in the comments below. FileInputStream which is used for specifying the file path and name, throws FileNotFoundException. The read() method which reads the file content throws IOException and the close() method which closes the file input stream also throws IOException.
4. **Why this compilation error?** As I mentioned in the beginning that checked exceptions gets checked during compile time. Since we didn’t handled/declared the exceptions, our program gave the compilation error.
5. **How to resolve the error?** There are two ways to avoid this error. We will see both the ways one by one.  
     
   **Method 1:** Declare the exception using throws keyword.  
   As we know that all three occurrences of checked exceptions are inside main() method so one way to avoid the compilation error is: Declare the exception in the method using throws keyword. You may be thinking that our code is throwing FileNotFoundException and IOException both then why we are declaring the IOException alone. The reason is that IOException is a parent class of FileNotFoundException so it by default covers that. If you want you can declare that too like this public static void main(String args[]) throws IOException, FileNotFoundException.
6. However, I think checked exceptions are useful - they are used when you want to force the user of your API to think how to handle the exceptional situation (if it is recoverable). It's just that checked exceptions are overused in the Java platform, which makes people hate them.
7. Checked Exceptions are subclasses of java.lang.Exception. The checked exception classes named in the throws clause are part of the contract between the implementor and user of the method or constructor.
8. Checked exceptions are checked at compile time by the JVM and its related to resources (files/db/stream/socket etc). The motive of checked exception is that at compile time if the resources are not available the application should define an alternative behaviour to handle this in the catch/finally block.
9. Checked Exception in Java is all those Exception which requires being catches and handled during compile time. If Compiler doesn’t see try or catch block handling a Checked Exception, it throws Compilation error. Now Which Exception is checked Exception and Why Checked Exception are introduced in first place? All the Exception which are direct sub Class of Exception but not inherit RuntimeException are Checked Exception.
10. **When to use Checked Exception in Java**?  
     Ans Knowing Checked Exception is not that useful until you know how to use Checked Exception in Java. Java has often been criticized for its Checked Exception strategy, arguments given are that checked Exception adds lot of boiler plate code and makes whole class or function unreadable. Somewhat I agree with this and java also recognize this by introducing improved Exception handling mechanism in Java7 but Checked Exception does have its real purpose.
11. **Example** of checked Exception in Java API  
    Ans Following are some Examples of Checked Exception in Java library:  
    IOException  
    SQLException  
    DataAccessException  
    ClassNotFoundException  
    InvocationTargetException
12. Checked Exception is required to be handled by compile time while Unchecked Exception doesn't.
13. Checked Exception is direct sub-Class of Exception while Unchecked Exception are of RuntimeException.
14. CheckedException represent scenario with higher failure rate
15. **What are Checked exceptions?**Ans The exceptions that are subtypes of Exception (exclude subtypes of RuntimeException) are categorized as checked exceptions. When we use code that can throw checked exceptions, we must handle them, otherwise the compiler will complain.
16. Checked Exceptions in Java: Are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using throws keyword.
17. **Example,** consider the following Java program that opens file at location “C:\test\a.txt” and prints first three lines of it. The program doesn’t compile, because the function main() uses FileReader() and FileReader() throws a checked exception FileNotFoundException. It also uses readLine() and close() methods, and these methods also throw checked exception IOException.
18. Checked exceptions − A checked exception is an exception that occurs at the compile time, these are also called as compile time exceptions. These exceptions cannot simply be ignored at the time of compilation, the programmer should take care of (handle) these exceptions.
19. **Checked exceptions:** Represent invalid conditions in areas outside the immediate control of the program (invalid user input, database problems, network outages, absent files), are subclasses of Exception, a method is obliged to establish a policy for all checked exceptions thrown by its implementation (either pass the checked exception further up the stack, or handle it somehow).
20. If a checked exception is thrown in a method, it must be handled in one of three ways:  
        1. By using a try block and catching the exception in a handler and dealing with it.  
        2. By using a try block and catching the exception in a handler, but throwing another exception that is either unchecked or declared in its throws clause.  
        3. By explicitly allowing propagation of the exception to its caller by declaring it in the throws clause of its method header.
21. Checked exceptions result in multiple throws clause declarations.
22. The problem with checked exceptions is they encourage people to swallow important details (namely, the exception class). If you choose not to swallow that detail, then you have to keep adding throws declarations across your whole app. This means 1) that a new exception type will affect lots of function signatures, and 2) you can miss a specific instance of the exception you actually -want- to catch (say you open a secondary file for a function that writes data to a file. The secondary file is optional, so you can ignore its errors, but because the signature throws IOException, it's easy to overlook this).
23. Checked exceptions are not really exceptions.
24. The thing about checked exceptions is that they are not really exceptions by the usual understanding of the concept. Instead, they are API alternative return values.
25. The whole idea of exceptions is that an error thrown somewhere way down the call chain can bubble up and be handled by code somewhere further up, without the intervening code having to worry about it.
26. Checked exceptions, on the other hand, require every level of code between the thrower and the catcher to declare they know about all forms of exception that can go through them. This is really little different in practice to if checked exceptions were simply special return values which the caller had to check for.
27. Are checked exceptions good or are they bad? In other words, should programmers be forced to handle checked exceptions or given the opportunity to ignore them? I like the idea of enforcing more robust software. However, I also think that Java's exception-handling mechanism needs to evolve to make it more programmer-friendly.
28. Checked exceptions denote error scenarios which are outside the immediate control of the program. They occur usually interacting with outside resources/ network resources e.g. database problems, network connection errors, missing files etc.
29. Example of mostly faced checked exceptions are : ClassNotFoundException, IOException, SQLException and so on. Please refer official java documentation for all such classes.
30. Checked exceptions are exceptions that the designers of Java feel that your programs absolutely must provide for, one way or another.
31. Whenever you code a statement that could throw a checked exception, your program must do one of two things:  
    a) Catch the exception by placing the statement within a try statement that has a catch block for the exception.
    1. Specify a throws clause on the method that contains the statement to indicate that your method doesn’t want to handle the exception, so it’s passing the exception up the line.
    2. But it is not that simple – checked exceptions are making the code “uglier”. They force the developer to write try/catch blocks or to rethrow the exception. But rethrowing hides another problem – some exceptions should not cross module boundaries. The most common thing to do when you are forced to catch a checked exception, which you have no idea what to do with, is to simply wrap it in a RuntimeException and rethrow it.
32. **Expected but unpreventable**: The caller did everything within their power to validate the input parameters, but some condition outside their control has caused the operation to fail. For example, you try reading a file but someone deletes it between the time you check if it exists and the time the read operation begins. By declaring a checked exception, you are telling the caller to anticipate this failure.
33. According to this paradigm, I would say we must:

* always use checked exceptions.
* never throw/use unchecked exceptions.
* use only Exception, without any sub-types.
* always declare one exception type in the throws block.
* never catch without rethrowing; read [more about that here](http://www.yegor256.com/2015/07/09/catch-if-cant-otherwise.html).

This paradigm diverges from many other articles I've found on this subject. Let's compare and discuss.

1. Checked exceptions:

* represent invalid conditions in areas outside the immediate control of the program (invalid user input, database problems, network outages, absent files)
* are subclasses of [Exception](http://docs.oracle.com/javase/7/docs/api/java/lang/Exception.html)
* a method is *obliged* to establish a policy for all checked exceptions thrown by its implementation (either pass the checked exception further up the stack, or handle it somehow)

## Why Checked Exceptions?

* From the JLS: "This compile-time checking for the presence of exception handlers is designed to reduce the number of exceptions which are not properly handled."
* If you are throwing an exception for an abnormal condition that you feel client programmers should consciously decide how to handle, throw a checked exception.
* Unchecked exceptions indicate software bugs.
* Precisely because unchecked exceptions usually represent software bugs, they often *can't* be handled somewhere with more context.

**Topic 18: String**

1. In java, string is basically an object that represents sequence of char values. An array of characters works same as java string.
2. The CharSequence interface is used to represent sequence of characters. It is implemented by String, StringBuffer and StringBuilder classes. It means, we can create string in java by using these 3 classes.
3. The java String is immutable i.e. it cannot be changed. Whenever we change any string, a new instance is created. For mutable string, you can use StringBuffer and StringBuilder classes.
4. In java, string is an object that represents a sequence of characters. The java.lang.String class is used to create string object.
5. There are two ways to create String object:
6. By string literal: String s="welcome";  //will not create new instance
7. By new keyword: String s=new String("Welcome");//creates 2 objects and one reference variable
8. **Why String objects are immutable?**

Ans

* **Security**: parameters are typically represented as String in network connections, database connection urls, usernames/passwords etc. If it were mutable, these parameters could be easily changed.
* **Synchronization** **and concurrency:** making String immutable automatically makes them thread safe thereby solving the synchronization issues.
* **Caching**: when compiler optimizes your String objects, it sees that if two objects have same value (a="test", and b="test") and thus you need only one string object (for both a and b, these two will point to the same object).
* **Class loading**: String is used as arguments for class loading. If mutable, it could result in wrong class being loaded (because mutable objects change their state).

1. **What is string constant pool?**

Ans. Now lets know what happens here step by step

* The class is loaded when JVM is invoked.
* JVM will look for all the string literals in the program
* First, it finds the variable s which refers to the  literal “prasad” and it will be created in the memory
* A reference for the literal “prasad” will be placed in the string constant pool memory.
* Then it finds another variable s2 which is referring to the same string literal “prasad“.
* Now that JVM has already found a string literal “prasad“, both the variables s and s2 wil refer to the same object i.e. “prasad“

1. **compareTo()** method compares values and returns an int which tells if the string compared is less than, equal to or greater than the other string. It compares the String based on natural ordering i.e alphabetically.
2. **Why Use String?**

**Ans.** One of the primary functions of modern computer science, is processing human language.

Similarly to how numbers are important to math, language symbols are important to meaning and decision making. Although it may not be visible to computer users, computers process language in the background as precisely and accurately as a calculator. Help dialogs provide instructions. Menus provide choices. And data displays show statuses, errors, and real-time changes with language.

As a Java programmer, one of your main tools for storing and processing language is going to be the String class.

1. **Important Points to Note:**
   1. Once created the value cannot be altered. Thus String objects are called immutable.
   2. The Java Virtual Machine(JVM) creates a memory location especially for Strings called **String Constant Pool**. That’s why String can be initialized without ‘new’ key word.
   3. String class falls under**java.lang.String hierarchy**. But there is no need to import this class. Java platform provides them automatically.
   4. String reference can be overridden but that does not delete the content; i.e., if

String h1 = "hello";

h1 = "hello"+"world";

then "hello" String does not get deleted. It just looses its handle.

* 1. Multiple references can be used for same String but it will occur in the same place; i.e., if

String h1 = "hello";

String h2 = "hello";

String h3 = "hello";

then only one pool for String “hello” is created in the memory with 3 references-h1,h2,h3

* 1. If a **number is quoted in “ ”** then it**becomes a string**, not a number any more. That means if

String S1 ="The number is: "+ "123"+"456";

System.out.println(S1);

then it will print: The number is: 123456

If the initialization is like this:

String S1 = "The number is: "+(123+456);

System.out.println(S1);

then it will print: The number is:579 That's all to Strings.

* 1. Strings receive *special treatment* in Java, because they are used frequently in a program. Hence, efficiency (in terms of computation and storage) is crucial.
  2. The designers of Java decided to retain primitive types in an object-oriented language, instead of making everything an object, so as to improve the performance of the language.
  3. Primitives are stored in the call stack, which require less storage spaces and are cheaper to manipulate. On the other hand, objects are stored in the program heap, which require complex memory management and more storage spaces.
  4. For performance reason, Java's String is designed to be in between a primitive and a class.
  5. The special features in String include:
     + The '+' operator, which performs addition on primitives (such as int and  double), is overloaded to operate on  String objects.  '+' performs concatenation for two String operands.
     + Java does not support *operator overloading* for software engineering consideration.
     + The '+' operator is the *only* operator that is internally overloaded to support string concatenation in Java.
     + Take note that '+' does not work on any two arbitrary objects, such as Points or Circles.

1. **String Literal vs. String Object:** 
   1. As mentioned, there are two ways to construct a string: implicit construction by assigning a string literal or explicitly creating a String object via the new operator and constructor. For example

String s1 = "Hello"; // String literal

String s2 = "Hello"; // String literal

String s3 = s1; // same reference

String s4 = new String("Hello"); // String object

String s5 = new String("Hello"); // String object

* 1. Java has provided a special mechanism for keeping the String literals - in a so-called *string common pool*. If two string literals have the same contents, they will share the same storage inside the common pool. This approach is adopted to *conserve storage* for frequently-used strings.
  2. On the other hand, String objects created via the new operator and constructor are kept in the heap. Each String object in the heap has its own storage just like any other object. There is no sharing of storage in heap even if two String objects have the same contents.
  3. You can use the method equals() of the String class to compare the contents of two Strings. You can use the relational equality operator '==' to compare the references (or pointers) of two objects. Study the following codes:

s1 == s1; // true, same pointer

s1 == s2; // true, s1 and s1 share storage in common pool

s1 == s2; // true, s1 and s1 share storage in common pool

s1 == s3; // true, s3 is assigned same pointer as s1

s1.equals(s3); // true, same contents

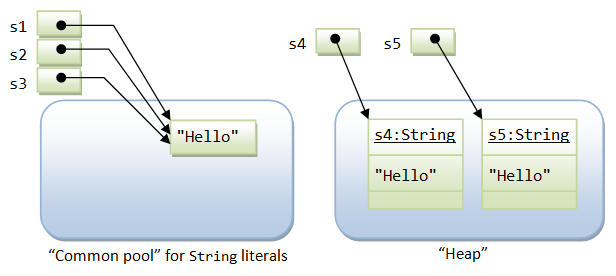
s1 == s4; // false, different pointers

s1.equals(s4); // true, same contents

s4 == s5; // false, different pointers in heap

s4.equals(s5); // true, same contents

* 1. Important Notes:
* In the above example, I used relational equality operator '==' to compare the references of two String objects. This is done to demonstrate the differences between string literals sharing storage in the common pool and String objects created in the heap. It is a logical error to use (str1 == str2) in your program to compare the contents of two Strings.
* String can be created by directly assigning a String literal which is shared in a common pool. It is uncommon and not recommended to use the new operator to construct a String object in the heap.



## Creating Format Strings:

* You have printf() and format() methods to print output with formatted numbers. The String class has an equivalent class method, format(), that returns a String object rather than a PrintStream object.
* Using String's static format() method allows you to create a formatted string that you can reuse, as opposed to a one-time print statement. **For example, instead of-**
* System.out.printf("The value of the float variable is "+ "%f, while the value of the integer "+ "variable is %d, and the string "+ "is %s", floatVar, intVar, stringVar);
* **You can write-**

String fs;

fs = String.format("The value of the float variable is "+ "%f, while the value of the integer "+

"variable is %d, and the string "+"is %s", floatVar, intVar, stringVar);

System.out.println(fs);

**Topic 19: String Buffer**

* 1. Java StringBuffer class is used to create mutable (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.
  2. **Important Constructors of StringBuffer class**

StringBuffer()

StringBuffer(String str)

StringBuffer(int capacity)

* 1. StringBuffer is fast and consumes less memory when you concat strings.
  2. StringBuffer class doesn't override the equals() method of Object class.
  3. The java.lang.StringBuffer class is a thread-safe, mutable sequence of characters. Following are the important points about StringBuffer −
* A string buffer is like a String, but can be modified.
* It contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls.
* They are safe for use by multiple threads.
* Every string buffer has a capacity.
  1. public final class StringBuffer extends Object implements Serializable, CharSequence
  2. This class inherits methods from the following classes –
* java.lang.Object
  1. Main difference between the StringBuffer and StringBuilder is that StringBuilders methods are not thread safe (not synchronised).
  2. StringBuffer class is used to create a mutable string object i.e its state can be changed after it is created. It represents growable and writable character sequence. As we know that String objects are immutable, so if we do a lot of changes with String objects, we will end up with a lot of memory leak.
  3. So StringBuffer class is used when we have to make lot of modifications to our string. It is also thread safe i.e multiple threads cannot access it simultaneously. StringBuffer defines 4 constructors.
* StringBuffer ().
* StringBuffer ( int size).
* StringBuffer ( String str ).
* StringBuffer ( charSequence [ ]ch ).
  1. StringBuffer is a peer class of String that provides much of the functionality of strings. String represents fixed-length, immutable character sequences while StringBuffer represents growable and writable character sequences.
  2. StringBuffer may have characters and substrings inserted in the middle or appended to the end. It will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth.
  3. **StringBuffer( ):**It reserves room for 16 characters without reallocation.
  4. **StringBuffer( int size)**It accepts an integer argument that explicitly sets the size of the buffer.
  5. **StringBuffer(String str):**It accepts a **String**argument that sets the initial contents of the StringBuffer object and reserves room for 16 more characters without reallocation.
  6. **delete( ) and deleteCharAt( ):** It can delete characters within a StringBuffer by using the methods delete( ) and deleteCharAt( ).The delete( ) method deletes a sequence of characters from the invoking object. Here, start Index specifies the index of the first character to remove, and end Index specifies an index one past the last character to remove. Thus, the substring deleted runs from start Index to endIndex–1. The resulting StringBuffer object is returned. The   deleteCharAt( ) method deletes the character at the index specified by loc.
  7. **Some Interesting facts:**

* + - * lang.Object is extended by java.lang.StringBuffer.
      * All Implemented Interfaces of StringBuffer class:Serializable, Appendable, CharSequence.
      * public final class StringBuffer extends Object implements Serializable, CharSequence.
      * String buffers are safe for use by multiple threads. The methods can be synchronized wherever necessary so that all the operations on any particular instance behave as if they occur in some serial order.
      * Whenever an operation occurs involving a source sequence (such as appending or inserting from a source sequence) this class synchronizes only on the string buffer performing the operation, not on the source.
      * It inherits some of the methods from Object class which are clone, equals, finalize, getClass, hashCode, notify, notifyAll.
  1. The significant performance difference between these two classes is that StringBuffer is faster than String when performing simple concatenation.
  2. String buffers are used by the compiler to implement the binary string concatenation operator +
  3. The principal operations on a StringBuffer are the append and insert methods, which are overloaded so as to accept data of any type. Each effectively converts a given datum to a string and then appends or inserts the characters of that string to the string buffer. The append method always adds these characters at the end of the buffer; the insert method adds the characters at a specified point.
  4. StringBuffer class is a mutable class unlike the String class which is immutable. Both the capacity and character string of a StringBuffer Class.
  5. StringBuffer can be changed dynamically. String buffers are preferred when heavy modification of character strings is involved (appending, inserting, deleting, modifying etc).
  6. Strings can be obtained from string buffers. Since the StringBuffer class does not override the equals() method from the Object class, contents of string buffers should be converted to String objects for string comparison.
  7. A StringIndexOutOfBoundsException is thrown if an index is not valid when using wrong index in String Buffer manipulations.
  8. **When to use String and when StringBuffer?**

Ans. If there is a need to change the contents frequently, StringBuffer should be used instead of String because StringBuffer concatenation is significantly faster than String concatenation.

**Topic 20: Collection Framework Basics**

* 1. Collections in java are a framework that provides architecture to store and manipulate the group of objects.
  2. All the operations that you perform on a data such as searching, sorting, insertion, manipulation, deletion etc. can be performed by Java Collections.
  3. Java Collection simply means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque etc.) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet etc).

#### What is Collection in java ?

Ans. Collection represents a single unit of objects i.e. a group.

#### What is framework in java

* provides readymade architecture.
* represents set of classes and interface.
* is optional.

#### What is Collection framework?

Ans. Collection framework represents a unified architecture for storing and manipulating group of objects. It has:

1. Interfaces and its implementations i.e. classes
2. Algorithm

The Collection interface (**java.util.Collection**) and Map interface (**java.util.Map**) are two main root interfaces of Java collection classes.

* 1. **Advantages of Collection Framework?**

1. **Consistent API:** The API has basic set of interfaces like Collection, Set, List, or Map. All those classes (such as ArrayList, LinkedList, Vector etc) which implements, these interfaces have some common set of methods.
2. **Reduces programming effort:** The programmer need not to worry about design of Collection rather than he can focus on its best use in his program.
3. **Increases program speed and quality:** Increases performance by providing high-performance implementations of useful data structures and algorithms.

## What Is a Collections Framework?

Ans. A collections framework is a unified architecture for representing and manipulating collections. All collections frameworks contain the following:

* **Interfaces**: These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
* **Implementations**: These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
* **Algorithms**: These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface. In essence, algorithms are reusable functionality.

## Benefits of the Java Collections Framework

Ans. The Java Collections Framework provides the following benefits:

* **Reduces programming effort:** By providing useful data structures and algorithms, the Collections Framework frees you to concentrate on the important parts of your program rather than on the low-level "plumbing" required to make it work. By facilitating interoperability among unrelated APIs, the Java Collections Framework frees you from writing adapter objects or conversion code to connect APIs.
* **Increases program speed and quality:** This Collections Framework provides high-performance, high-quality implementations of useful data structures and algorithms. The various implementations of each interface are interchangeable, so programs can be easily tuned by switching collection implementations. Because you're freed from the drudgery of writing your own data structures, you'll have more time to devote to improving programs' quality and performance.
* **Allows interoperability among unrelated APIs:** The collection interfaces are the vernacular by which APIs pass collections back and forth. If my network administration API furnishes a collection of node names and if your GUI toolkit expects a collection of column headings, our APIs will interoperate seamlessly, even though they were written independently.
* **Reduces effort to learn and to use new APIs:** Many APIs naturally take collections on input and furnish them as output. In the past, each such API had a small sub-API devoted to manipulating its collections. There was little consistency among these ad hoc collections sub-APIs, so you had to learn each one from scratch, and it was easy to make mistakes when using them. With the advent of standard collection interfaces, the problem went away.
* **Reduces effort to design new APIs:** This is the flip side of the previous advantage. Designers and implementers don't have to reinvent the wheel each time they create an API that relies on collections; instead, they can use standard collection interfaces.
* **Fosters software reuse:** New data structures that conform to the standard collection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces.
  1. The [Java Collections Framework](https://docs.oracle.com/javase/tutorial/collections/intro/index.html) is a collection of interfaces and classes which helps in storing and processing the data efficiently. This framework has several useful classes which have tons of useful functions which makes a programmer task super easy. I have written several tutorials on Collections and below are the links of those. All the tutorials are shared with examples and source codes to help you understand better.
  2. **The collections framework was designed to meet several goals, such as −**
* The framework had to be high-performance. The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hashtables) were to be highly efficient.
* The framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.
* The framework had to extend and/or adapt a collection easily.
  1. **A collections framework is a unified architecture for representing and manipulating collections. All collections frameworks contain the following −**
* **Interfaces** − These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
* **Implementations, i.e., Classes** − These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
* **Algorithms** − These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface.

In addition to collections, the framework defines several map interfaces and classes. Maps store key/value pairs. Although maps are not collections in the proper use of the term, but they are fully integrated with collections.

* 1. The Java collections framework gives the programmer access to prepackaged data structures as well as to algorithms for manipulating them.
  2. A collection is an object that can hold references to other objects. The collection interfaces declare the operations that can be performed on each type of collection.
  3. The classes and interfaces of the collections framework are in package java.util.
  4. Collections are like containers that groups multiple items in a single unit. For example; a jar of chocolates, list of names etc. Collections are used almost in every programming language and when Java arrived, it also came with few Collection classes; Vector, Stack, Hashtable, Array. Java 1.2 provided Collections framework that is architecture to represent and manipulate Collections in java in a standard way.
  5. Java Collections Framework consists of following parts:
* **Interfaces**: Java Collections Framework interfaces provides the abstract data type to represent collection. java.util.Collection is the root interface of Collections Framework. It is on the top of Collections framework hierarchy. It contains some important methods such as size(), iterator(), add(), remove(), clear() that every Collection class must implement. Some other important interfaces are [java.util.List](http://www.journaldev.com/11444/java-list), java.util.Set, java.util.Queue and [java.util.Map](http://www.journaldev.com/11641/java-map). Map is the only interface that doesn’t inherits from Collection interface but it’s part of Collections framework. All the collections framework interfaces are present in java.util package.
* **Implementation Classes**: Collections in Java provides core implementation classes for collections. We can use them to create different types of collections in java program. Some important collection classes are  ArrayList,  LinkedList,  HashMap,  TreeMap,  HashSet,  TreeSet. These classes solve most of our programming needs but if we need some special collection class, we can extend them to create our custom collection class.

Java 1.5 came up with thread-safe collection classes that allowed to modify Collections while iterating over it, some of them are  CopyOnWriteArrayList,  ConcurrentHashMap,  CopyOnWriteArraySet. These classes are in java.util.concurrent package. All the collection classes are present in java.util and java.util.concurrent package.

* **Algorithms**: Algorithms are useful methods to provide some common functionalities, for example searching, sorting and shuffling.

### [What are collections](http://www.vogella.com/tutorials/JavaCollections/article.html#javacollections1) ?

Ans. The Java language supports arrays to store several objects. An array is initialized with an predefined size during instantiation. To support more flexible data structures the core Java library provides the collection framework. A collection is a data structure which contains and processes a set of data. The data stored in the collection is encapsulated and the access to the data is only possible via predefined methods. For example the developer can add elements to an collection via a method. Collections use internally arrays for there storage but hide the complexity of managing the dynamic size from the developer.

For example if your application saves data in an object of type People, you can store several People objects in a collection.

* 1. **How to iterate collection by many ways**

## Ans. Iterable.forEach method

|  |  |
| --- | --- |
| Collection<String> collection = Arrays.asList("How", "To", "Iterate", "In", "Java");   collection.forEach(s -> System.out.println(s)); Java “foreach” loop (Java 5) for (Type var : Iterable<Type>) {  // do something with "var"  } java.util.Iterator (Java 2) Using Iterator is still very popular way to iterate over collections, mainly due to addtional methods it provide to manipulate the collection elements.   |  | | --- | | Collection<String> collection = Arrays.asList("How", "To", "Iterate", "In", "Java");    Iterator<String> itr = collection.iterator();    while(itr.hasNext()) {      System.out.println(itr.next());  } Traditional for loop for (init; test; change) {  // do something  } | |

## Disadvantages of collection framework:

1. 1. It must cast to correct type.
2. 2. It can’t be done compile-time type checking.

**Iterable Interface:**

The Iterable interface (java.lang.Iterable) is one of the root interfaces of the Java collection classes. The Collection interface extends Iterable, so all subtypes of Collection also implement the Iterableinterface.

The Iterable interface has only one method:

public interface Iterable<T> {

public Iterator<T> iterator();

}

When a class implements the Iterable interface, it is telling other classes that you can get an Iterator object to use to iterate over (i.e., traverse) the data in the object.

### Iterator interface

|  |
| --- |
| Iterator interface provides the facility of iterating the elements in forward direction only. |

#### Methods of Iterator interface

There are only three methods in the Iterator interface. They are:

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | Public boolean hasNext() | It returns true if iterator has more elements. |
| 2 | public Object next() | It returns the element and moves the cursor pointer to the next element. |
| 3 | public void remove() | It removes the last elements returned by the iterator. It is rarely used. |

**Comparble and Comparator Interface**

# *==, .equals(), compareTo(), and compare()*

## Equality comparison: One way for primitives, Four ways for objects

|  |  |  |
| --- | --- | --- |
| Comparison | Primitives | Objects |
| a == b, a != b | Equal values | **Compares references, not values.** The use of == with object references is generally limited to the following:   * Comparing to see if a reference is null. * Comparing two enum values. This works because there is only one object for each enum constant. * You want to know if two references are to the *same object* |
| a.equals(b) | N/A | Compares values for equality. Because this method is defined in the Object class, from which all other classes are derived, it's automatically defined for every class. However, it doesn't perform an intelligent comparison for most classes unless the class overrides it. It has been defined in a meaningful way for most Java core classes. If it's not defined for a (user) class, it behaves the same as ==.  It turns out that defining equals() isn't trivial; in fact it's moderately hard to get it right, especially in the case of subclasses. The best treatment of the issues is in Horstmann's *Core Java Vol 1*. [TODO: Add explanation and example] |
| a.compareTo(b) | N/A | **Comparable interface.** Compares values and returns an int which tells if the values compare less than, equal, or greater than. If your class objects have a natural order, implement the *Comparable<T>* interface and define this method. All Java classes that have a natural ordering implement this (String, Double, BigInteger, ...). |
| compare(a, b) | N/A | **Comparator interface.** Compares values of two objects. This is implemented as part of the *Comparator<T>* interface, and the typical use is to define one or more small utility classes that implement this, to pass to methods such as sort() or for use by sorting data structures such as TreeMap and TreeSet. You might want to create a Comparator object for the following.   * **Multiple comparisons.** To provide several different ways to sort something. For example, you might want to sort a Person class by name, ID, age, height, ... You would define a Comparator for each of these to pass to the sort()method. * **System class.** To provide comparison methods for classes that you have no control over. For example, you could define a Comparator for Strings that compared them by length. * **Strategy pattern.** To implement a *strategy* pattern, which is a situation where you want to represent an *algorithm* as an object that you can pass as a parameter, save in a data structure, etc.   If your class objects have one natural sorting order, you may not need this. |

## Comparing Object references with the == and != Operators

The two operators that can be used with object references are comparing for equality (**==**) and inequality (**!=**). These operators compare two values to see if they **refer to the same object**. Although this comparison is very fast, it is often not what you want.

Usually you want to know if the objects have the same *value*, and not whether two objects are a *reference* to the same object. For example,

if (name == "Mickey Mouse") // Legal, but ALMOST SURELY WRONG

This is true only if name is a reference to the *same object* that "Mickey Mouse" refers to. This will be false if the String in name was read from input or computed (by putting strings together or taking the substring), even though name really does have exactly those characters in it.

Many classes (eg, String) define the equals() method to compare the *values* of objects.

## Comparing Object values with the equals() Method

Use the equals() method to compare object values. The equals() method returns a boolean value. The previous example can be fixed by writing:

if (name.**equals**("Mickey Mouse")) // Compares values, not references.

Because the equals() method makes a == test first, it can be fairly fast when the objects are identical. It only compares the values if the two references are not identical.

## Other comparisons - Comparable<T> interface

The equals method and == and != operators test for equality/inequality, but do not provide a way to test for relative values. Some classes (eg, String and other classes with a natural ordering) implement the *Comparable<T>*interface, which defines a compareTo method. You will want to implement *Comparable<T>* in your class if you want to use it with Collections.sort() or Arrays.sort() methods.

## Defining a Comparator object

As described in the table above on compare(), you can create Comparators to sort any arbitrary way for any class. For example, the String class defines the CASE\_INSENSITIVE\_ORDER comparator.

## If you override equals, you should also override hashCode()

**Overriding hashCode()**. The hashCode() method of a class is used for *hashing* in library data structures such as HashSet and HashMap. If you override equals(), you should override hashCode() or your class will not work correctly in these (and some other) data structures.

## Shouldn't .equals and .compareTo produce same result?

The general advice is that if a.equals(b) is true, then a.compareTo(b) == 0 should also be true. Curiously, BigDecimal violates this. Look at the Java API documentation for an explanation of the difference. This seems wrong, although their implementation has some plausibility.

## Other comparison methods

String has the specialized equalsIgnoreCase() and compareToIgnoreCase(). String also supplies the constant String.CASE\_INSENSITIVE\_ORDER Comparator.

## The === operator (Doesn't exist - yet?)

Comparing objects is somewhat awkward, so a === operator has been proposed. One proposal is that   
a === b would be the same as ((a == b) || ((a != null) && a.equals(b)))

## Common Errors

**Using == instead of equals() with Objects**

When you want to compare objects, you need to know whether you should use == to see if they are the *same object*, or equals() to see if they may be a different object, but have the *same value*. This kind of error can be very hard to find.

**Comparators and comparable** in Java are two interfaces which is used to implement sorting in Java. It’s often required to sort objects stored in any collection classes like ArrayList, HashSet or in Array and that time we need to use either  compare() or  compareTo() method defined in java.util.Comparator and java.lang.Comparable. In this Java tutorial we will see example of  Comparator and Comparable to sort object in Java and discuss some best practices around when to use Comparator interface etc. Any way before moving ahead Let’s see some important differences between Comparable and Comparator in Java.

1) Comparator in Java is defined in java.util package while Comparable interface in Java is defined in java.lang package, which very much says that Comparator should be used as an utility to sort objects which Comparable should be provided by default.

2) Comparator interface in Java has method public int compare (Object o1, Object o2) which returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second. While Comparable interface has method public int compareTo(Object o) which returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

3) If you see then logical difference between these two is *Comparator in Java* compare two objects provided to him, while Comparable interface compares this" reference with the object specified.

4) Comparable in Java is used to implement **natural ordering of object**. In Java API String, Date and wrapper classes implements Comparable interface. Its always good practice to override compareTo() for value objects.

5) If any class implement Comparable interface in Java then collection of that object either [List](http://javarevisited.blogspot.sg/2012/04/difference-between-list-and-set-in-java.html) or Array can be sorted automatically by using  Collections.sort()  or Arrays.sort() method and object will be sorted based on there natural order defined by CompareTo method.

6)Objects which implement *Comparable in Java*  can be used as keys in a SortedMap like [TreeMap](http://javarevisited.blogspot.sg/2011/12/treemap-java-tutorial-example-program.html" \t "_blank) or elements in a SortedSet  for example TreeSet, without specifying any Comparator.

**How to Compare String in Java**

[String is immutable in Java](http://javarevisited.blogspot.sg/2010/10/why-string-is-immutable-in-java.html) and one of the most used value class. For comparing String in Java we should not be worrying because String implements Comparable interface and provides a lexicographic implementation for CompareTo method which compare two strings based on contents of characters or you can say in lexical order. You just need to call String.compareTo(AnotherString) and Java will determine whether specified String is greater than , equal to or less than current object.

**How to Compare Dates in Java**

Dates are represented by java.util.Date class in Java and like String,  Date also implements Comparable in Java so they will be automatically sorted based on there natural ordering if they got stored in any sorted collection like TreeSet or TreeMap. If you explicitly wants to compare two dates in Java you can call  Date.compareTo(AnotherDate)  method in Java and it will tell whether specified date is greater than , equal to or less than current String

**When to use Comparator and Comparable in Java**

At last let’s see some best practices and recommendation on when to use Comparator or Comparable in Java:

1) If there is a natural or default way of sorting Object already exist during development of Class than use Comparable. This is intuitive and you given the class name people should be able to guess it correctly like Strings are sorted chronically, Employee can be sorted by there Id etc. On the other hand if an Object can be sorted on multiple ways and client is specifying on which parameter sorting should take place than use Comparator interface. for example Employee can again be sorted on name, salary or department and clients needs an API to do that. Comparator implementation can sort out this problem.

2) Some time you write code to sort object of a class for which you are not the original author, or you don't have access to code. In these cases you can not implement Comparable and Comparator is only way to sort those objects.

3) Beware with the fact that How those object will behave if stored in SorteSet or SortedMap like TreeSet and [TreeMap](http://javarevisited.blogspot.sg/2011/12/treemap-java-tutorial-example-program.html" \t "_blank). If an object doesn't implement Comparable than while putting them into SortedMap, always provided corresponding Comparator which can provide sorting logic.

4) Order of comparison is very important while implementing Comparable or Comparator interface. for example if you are sorting object based upon name than you can compare first name or last name on any order, so decide it judiciously. I have shared more detailed tips on compareTo on my post how to implement CompareTo in Java.

5) Comparator has a distinct advantage of being self descriptive  for example if you are writing Comparator to compare two Employees based upon there salary than name that comparator as SalaryComparator, on the other hand compareTo()

**How to Override Equals () and hashCode() Example**

class Complex {

    private double re, im;

    public Complex(double re, double im) {

        this.re = re;

        this.im = im;

    }

}

public class Main {

    public static void main(String[] args) {

        Complex c1 = new Complex(10, 15);

        Complex c2 = new Complex(10, 15);

        if (c1 == c2) {

            System.out.println("Equal ");

        } else {

            System.out.println("Not Equal ");

        }

    }

}

Output: Not Equal

The reason for printing “Not Equal” is simple: when we compare c1 and c2, it is checked whether both c1 and c2 refer to same object or not ([object variables are always references in Java](http://www.geeksforgeeks.org/archives/8926)). c1 and c2 refer to two different objects, hence the value (c1 == c2) is false. If we create another reference say c3 like following, then (c1 == c3) will give true.

Complex c3 = c1;  // (c3 == c1) will be true

So, how do we check for equality of values inside the objects? All classes in Java inherit from the Object class, directly or indirectly (See point 1 of [this](http://www.geeksforgeeks.org/archives/15055)). The [Object class](http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Object.html) has some basic methods like clone(), toString(), equals(),.. etc. We can override the equals method in our class to check whether two objects have same data or not.

 @Override

    public boolean equals(Object o) {

        // If the object is compared with itself then return true

        if (o == this) {

            return true;

        }

        /\* Check if o is an instance of Complex or not

          "null instanceof [type]" also returns false \*/

        if (!(o instanceof Complex)) {

            return false;

        }

        // typecast o to Complex so that we can compare data members

        Complex c = (Complex) o;

        // Compare the data members and return accordingly

        return Double.compare(re, c.re) == 0

                && Double.compare(im, c.im) == 0;

    }

}

// Driver class to test the Complex class

public class Main {

    public static void main(String[] args) {

        Complex c1 = new Complex(10, 15);

        Complex c2 = new Complex(10, 15);

        if (c1.equals(c2)) {

            System.out.println("Equal ");

        } else {

            System.out.println("Not Equal ");

        }

    }

}

**Output: Equal**

As a side note, when we override equals(), it is recommended to also override the hashCode() method. If we don’t do so, equal objects may get different hash-values; and hash based collections, including HashMap, HashSet, and Hashtable do not work properly

public int hashCode()

19. {

20. int hash = 7;

21. hash = 31 \* hash + num;

22. hash = 31 \* hash + (null == data ? 0 : data.hashCode());

23. return hash;

24. }

## How to implement compareTo in Java

There are certain rules and important points to remember while overriding compareTo method:

1) CompareTo method must return negative number if current object is less than other object, positive number if current object is greater than other object and zero if both objects are equal to each other.

2) CompareTo must be in consistent with [equals method](http://javarevisited.blogspot.com/2011/02/how-to-write-equals-method-in-java.html) e.g. if two objects are equal via equals() , there compareTo() must return zero otherwise if those objects are stored in SortedSet or SortedMap they will not behave properly. Since SortedSet or SortedMap use compareTo() to check the object if two unequal object are returned equal by compareTo those will not be added into Set or Map if they are not using external Comparator.  One example where compareTo is not consistent with equals in JDK is BigDecimal class. two BigDecimal number for which compareTo returns zero, equals returns false as clear from following BigDecimal comparison example:

**BigDecimal** bd1 = **new** **BigDecimal**("2.0");  
**BigDecimal** bd2 = **new** **BigDecimal**("2.00");  
        
**System**.out.println("comparing BigDecimal using equals: " + bd1.equals(bd2));  
**System**.out.println("comparing BigDecimal using compareTo: " + bd1.compareTo(bd2));  
  
Output:  
comparing **BigDecimal** using equals: **false**  
comparing **BigDecimal** using compareTo: 0

How does it affect BigDecimal ? well if you store these two BigDecimal in HashSet you will end up with duplicates (violation of Set Contract) i.e. two elements while if you store them in TreeSet you will end up with just 1 element because HashSet uses equals to check duplicates while TreeSet uses compareTo to check duplicates. That's why its suggested to keep **compareTo consistent with equals method in java**.

3) CompareTo() must throw NullPointerException if current object get compared to null object as opposed to equals() which return false on such scenario.

4) Another important point to note is **don't use subtraction for comparing integral values** because result of subtraction can overflow as every int operation in Java is modulo 2^32. use either Integer.compareTo()  or logical operators for comparison. There is one scenario where you can use subtraction to reduce clutter and improve performance. As we know compareTo doesn't care magnitude, it just care whether result is positive or negative. While comparing two integral fields you can use subtraction if you are absolutely sure that both operands are positive integer or more precisely there different must be less than Integer.MAX\_VALUE. In this case there will be no overflow and your compareTo will be concise and faster.

5. Use relational operator to compare integral numeric value i.e. < or > but use Float.compareTo() or Double.compareTo() to compare [floating point number](http://javarevisited.blogspot.com/2011/10/convert-double-to-string-example.html) as relational operator doesn't obey contract of compareTo for floating point numbers.

6. CompareTo() method is for comparison so **order in which you compare two object matters**. If you have more than one significant field to compare than always *start comparing from most significant field* to least significant field. here **compareTo is different with equals** because in case of equality check order doesn't matter. like in above *example of compareTo* if we don't consider Id and compare two student by its name and age than name should be first compare and than age, so if two student have same name one that has higher age should result in greater.

Student john12 = **new** Student(1001, "John", 12);  
Student john13 = **new** Student(1002, "John", 13);  
        
*//compareTo will return -1 as age of john12 is less than john13*  
**System**.out.println("comparing John, 12 and John, 13 with compareTo :" + john12.compareTo(john13));  
  
Output:  
comparing John, 12 and John, 13 with compareTo :-1

7. Another important point while comparing String using compareTo is to consider case. just like equals() doesn't consider case, compareTo also do not consider case, if you want to compare regardless of case than use String.compareToIgnoreCase() as we have used in above example.

**Where compareTo() method used in Java**

---------------------------------------------------

In Java API compareTo() method is used in SortedSet e.g. TreeSet and SortedMap e.g. TreeMap for sorting elements on natural order if no explicit Comparator is passed to Collections.sort() method e.g.

[**List**](http://www.google.com/search?sitesearch=java.sun.com&q=allinurl%3Aj2se%2F1+5+0%2Fdocs%2Fapi+List) stocks = getListOfStocks();   
[**Collections**](http://java.sun.com/j2se/1.5.0/docs/api/java/util/Collections.html).sort(stocks);

as mentioned earlier if compareTo is not consistent with equals then it could produce strange result. let took another example you put Stock A and Stock B on StockSet which is a TreeSet. Both Stock A and Stock B are equal by equals() method but compareTo return non zero values for it which makes that StockB will also be landed into TreeSet which was voilation of Set itself because it is not supposed to allow duplicates.

### Example of compareTo() in Java --------------------------------------

Let’s see an example of **how to override compareTo method in Java**. This method is very similar to equals and hashcode, key thing is compareTo should provide **natural ordering** e.g. in this example order of object based on Student ID.

**public** **class** Student **implements** [**Comparable**](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Comparable.html) {   
    **private** **int** id;  
    **private** **String** name;  
    **private** **int** age;  
    
    */\*  
     \*Compare a given Student with current(this) object.  
     \*If current Student id is greater than the received object,  
     \*then current object is greater than the other.  
     \*/*     
    **public** **int** compareTo(Student otherStudent) {  
       *// return*[*this.id*](http://this.id/)*- otherStudent.id ; //result of this operation can overflow*  
       **return** (**this**.id &lt; otherStudent.id ) ? -1: (**this**.id &gt; otherStudent.id) ? 1:0 ;  
  
    }   
}

here is another example of compareTo method in Java on which compareTo uses two significant field to compare objects:

**public** **class** Student **implements** [**Comparable**](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Comparable.html)<Student> {  
   .....      
    ***/\*\*  
     \* Compare a given Student with current(this) object.  
     \* first compare name and than age  
     \*/***  
    @**Override**  
    **public** **int** compareTo(Student otherStudent) {         
        *//compare name*  
        **int** nameDiff = name.compareToIgnoreCase(otherStudent.name);  
        **if**(nameDiff != 0){  
            **return** nameDiff;  
        }  
        *//names are equals compare age*  
        **return** age - otherStudent.age;  
    }  
    
}

That’s all on implementing compareTo method in Java. Please add any other fact which you think important to note while overriding compareTo. In summary compareTo should provide natural ordering and compareTo must be consistent with equals() method in Java.

**Map.Entry interface**

# Map.Entry interface in Java with example

Map.Entry interface in Java provides certain methods to access the entry in the Map. By gaining access to the entry of the Map we can easily manipulate them. Map.Entry is a generic and is defined in the java.util package.

**Declaration :**

Interface Map.Entry

k -> Key

V -> Value

**Methods:**

1. **equals (Object o)** – It compares the object (invoking object) with the Object o for equality.  
   **Syntax :**
2. boolean equals(Object o)
3. **Parameters :**
4. o -> Object to which we want to compare
5. **Returns:**
6. true: if both objects are equals
7. false: otherwise
8. **K getKey()**– Returns the key for the corresponding map entry.  
   **Syntax :**
9. K getKey()
10. **Parameters :**
11. -------------
12. **Returns:**
13. K -> Returns the corresponding Key of a entry on which it is invoked

**Exception –**

* + **IllegalSateException**is thrown when entry has been removed from the map.

1. **V getValue()** – Returns the value for the corresponding map entry.  
   **Syntax :**
2. V getValue()
3. **Parameters :**
4. -------------
5. **Returns:**
6. V -> Returns the corresponding value of a entry on which it is invoked
7. **int hashcode()** – Returns the hashcode for the corresponding map entry.  
   **Syntax :**
8. int hashcode()
9. **Parameters :**
10. -------------
11. **Returns:**
12. int -> Returns the hash-code of entry on which it is invoked
13. **V setValue(V v)** – Sets the value of the map with specified value v
14. V setValue(V v)
15. **Parameters :**
16. v -> Value which was earlier stored in the entry on which it is invoked
17. **Returns:**

int -> Returns the hash-code of a entry on which it is invoked

**Exception :**

* + **ClassCastException**is thrown if the class of the value ‘v’ is not a correct type for map.
  + **NullPointerException**is thrown if ‘null’ is stored in ‘v’, and ‘null ’ is not supported by map.
  + **UnsupportedOperationException**is thrown if we cannot manipulate the map or the put operation is not supported by the map.
  + **IllegalArgumetException**is thrown If there is some problem with the argument i.e v
  + **IllegalStateException** is thrown when entry has been removed from the map

**Set<Map.Entry> entrySet() –**Returns the Set view of the entire map.  
**Note :** This is not a method of Map.entry interface but it is discussed here because this method is useful while working with Map.Entry interface.

## Example and Usage of Map.Entry

In this example, we have a Map collection class TreeMap and we are iterating and displaying its key & value pairs using Map.Entry interfaces. Here we have used getKey() and getValue() methods of Map.Entry interface in order to get the key & value pairs.

import java.util.\*;

class TreeMapExample {

public static void main(String args[]) {

// Creating TreeMap object

TreeMap<String, Integer> tm = new TreeMap<String, Integer>();

// Adding elements to the Map

tm.put("Chaitanya", 27);

tm.put("Raghu", 35);

tm.put("Rajeev", 37);

tm.put("Syed", 28);

tm.put("Hugo", 32);

// Getting a set of the entries

Set set = tm.entrySet();

// Get an iterator

Iterator it = set.iterator();

// Display elements

while(it.hasNext()) {

Map.Entry me = (Map.Entry)it.next();

System.out.print("Key: "+me.getKey() + " & Value: ");

System.out.println(me.getValue());

}

}

}

**Output:**

Key: Chaitanya & Value: 27

Key: Hugo & Value: 32

Key: Raghu & Value: 35

Key: Rajeev & Value: 37

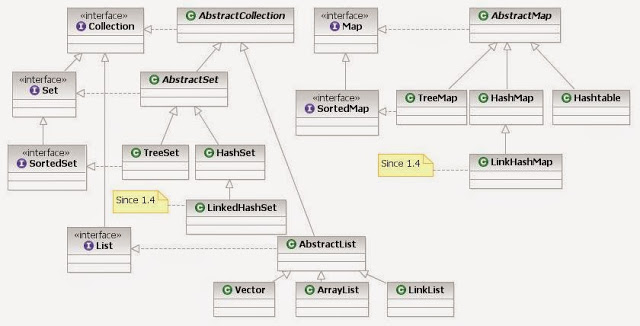
Key: Syed & Value: 28

**Collection questions**

**Q1  What is Collection ? What is a Collections Framework ? What are the benefits of Java Collections Framework ?**  
  
**Collection :** A collection (also called as container) is an object  that groups multiple elements into a single unit.  
  
**Collections Framework :**Collections framework provides unified architecture for manipulating and representing collections.  
  
**Benefits of Collections Framework :**  
1. Improves program quality and speed  
2. Increases the chances of reusability of software  
3. Decreases programming effort.  
  
**Q2 What is the root interface in collection hierarchy ?**  
  
Root interface in collection hierarchy is **Collection interface .**Few interviewer may argue that   
Collection interface extends **Iterable interface**. So iterable should be the root interface. But you should reply iterable interface present in java.lang package not in java.util package .It is clearly mentioned in [Oracle Collection  docs](http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html) , that Collection interface is a member of the Java Collections framework.  For [Iterable interface Oracle doc](https://docs.oracle.com/javase/7/docs/api/java/lang/Iterable.html" \t "_blank) , iterable interface is not mentioned as a part of the Java Collections framework .So if the question includes  collection hierarchy , then you should answer the question as Collection interface (which is found in java.util package).  
 **Q3 What is the difference between Collection and Collections ?**  
  
Collection is  an interface while Collections is a java class , both are present in java.util package and  part of java collections framework.

**Q4 Which collection classes are synchronized or thread-safe ?**  
  
Stack, Properties , Vector and Hashtable can be used in multi threaded environment because they are synchronized classes (or thread-safe).

**Q5 Name the core Collection  interfaces ?**

[](http://1.bp.blogspot.com/-ifC30f-ZZ7M/VU7wWRwqo3I/AAAAAAAAAmQ/b1CIhOP5NU8/s1600/Java_collection_framework.jpg)

The list of core collection interfaces are : just mention the important ones  
  
Important : Collection , Set , Queue , List , Map  
  
Other interface also in the list :  SortedSet, SortedMap , Deque, ListIterator etc.  
 **Q6 What is the difference between List and Set ?**  
  
Set contain only unique elements while List can contain duplicate elements.  
Set is unordered while List is ordered . List maintains the order in which the objects are added .  
  
**Q7 What is the difference between Map and Set ?**  
  
Map object has unique keys each containing some value, while Set contain only unique values.  
  
**Q8 What are the classes implementing List and Set interface ?**  
  
***Class implementing List interface :***  ArrayList , Vector , LinkedList ,  
  
***Class implementing Set interface :***HashSet , TreeSet

**Q9 What is an iterator ?**  
  
Iterator is an interface . It is found in java.util package. It provides methods to iterate over any Collection.  
  
  
**Q10 What is the difference between Iterator and Enumeration ?**  
  
The main difference between Iterator and Enumeration is that Iterator has remove() method while Enumeration doesn't.  
Hence , using Iterator we can manipulate objects by adding and removing the objects from the collections. Enumeration behaves like a read only interface as it can only traverse the objects and fetch it .  
  
**Q11 Which design pattern followed by Iterator ?**  
  
It follows iterator design pattern. Iterator design pattern provides us to navigate through the collection of objects by using a common interface without letting us know about the underlying implementation.  
  
Enumeration is an example of Iterator design pattern.  
 **Q12 Which methods you need to override to use any object as key in HashMap ?**  
  
To use any object as key in HashMap , it needs to implement equals() and hashCode() method .  
  
**Q13  What is the difference between Queue and Stack ?**  
  
Queue is a data structure which is based on FIFO ( first in first out ) property . An example of Queue in real world is buying movie tickets in the multiplex or cinema theaters.  
  
Stack is a data structure which is based on LIFO (last in first out) property . An example of Stack in real world is  insertion or removal of CD  from the CD case.  
  
**Q14 How to reverse the List in Collections ?**  
  
There is a built in reverse method in Collections class . reverse(List list) accepts list as parameter.  
  
**Collections.reverse(listobject);**  
  
**Q15 How to convert the array of strings into the list ?**  
  
Arrays class of java.util package contains the method asList() which accepts the array as parameter.  
So,  
  
**String[]  wordArray =  {"Love Yourself"  , "Alive is Awesome" , "Be in present"};**  
**List wordList =  Arrays.asList(wordArray);**  
  
  
***Intermediate Level (1-3 yrs): Java Collections Interview Questions  and Answers***

**Q16 What is the difference between ArrayList and Vector ?**

It is one of the frequently asked collection interview question , the main differences are  
Vector is synchronized while ArrayList is not . Vector is slow while ArrayList is fast . Every time when needed, Vector increases the capacity twice of its initial size while ArrayList increases its ArraySize by 50%. find detailed explanation   [ArrayList vs Vector](http://javahungry.blogspot.co.uk/2013/12/difference-between-arraylist-and-vector-in-java-collection-interview-question.html" \t "_blank)  .

**Q17 What is the difference between HashMap and Hashtable ?**  
  
It is one of the most popular collections interview question for java developer . Make sure you go through this once before appearing for the interview .  
Main differences between HashMap and Hashtable are :  
  
a. HashMap allows one null key and any number of null values while Hashtable does not allow null keys and null values.  
b. HashMap is not synchronized or thread-safe while Hashtable is synchronized or thread-safe .  
find detailed explanation here [Hashtable vs HashMap in Java](http://javahungry.blogspot.co.uk/2014/03/hashmap-vs-hashtable-difference-with-example-java-interview-questions.html" \t "_blank)  
  
**Q18 What is the difference between peek(),poll() and remove() method of the Queue interface ?**  
  
Both poll() and remove() method is used to remove head object of the Queue. The main difference lies when the Queue is empty().  
If Queue is empty then poll() method will return null . While in similar case , remove() method will throw NoSuchElementException .  
peek() method retrieves but does not remove the head of the Queue. If queue is empty then peek() method also returns null.

**Q19 What is the difference between Iterator and ListIterator.**  
  
Using Iterator we can traverse the list of objects in forward direction . But ListIterator can traverse the collection in both directions that is forward as well as backward.  
  
**Q20 What is the difference between Array and ArrayList in Java ?**  
  
This question checks whether student understand the concept of static and dynamic array. Some main differences between Array and ArrayList are :  
a. Array is static in size while ArrayList is dynamic in size.  
b. Array can contain primitive data types while ArrayList can not contain primitive data types.

find detailed explanation [ArrayList vs Array in Java](http://javahungry.blogspot.ca/2015/03/difference-between-array-and-arraylist-in-java-example.html" \t "_blank)

**Q21 What is the difference between HashSet and TreeSet ?**  
  
Main differences between HashSet and TreeSet are :  
a.  HashSet maintains the inserted elements in random order while TreeSet maintains elements in the sorted order  
b. HashSet can store null object while TreeSet can not store null object.  
find detailed explanation here [TreeSet vs HashSet in Java](http://javahungry.blogspot.co.uk/2014/03/difference-between-hashset-and-treeset-similarities-and-example.html" \t "_blank)  
  
  
**Q22 Write java code showing insertion,deletion and retrieval of HashMap object ?**  
  
Do it yourself (DIY) , if found any difficulty or doubts then please mention in the comments.  
  
**Q23 What is the difference between HashMap and ConcurrentHashMap ?**  
  
This is also one of the most popular java collections interview question . Make sure this question is in your to do list before appearing for the interview .  
Main differences between HashMap and ConcurrentHashMap are :  
a. HashMap is not synchronized while ConcurrentHashMap is synchronized.  
b. HashMap can have one null key and any number of null values while ConcurrentHashMap does not allow null keys and null values .  
find detailed explanation here [ConcurrentHashMap vs HashMap in Java](http://javahungry.blogspot.co.uk/2014/02/hashmap-vs-concurrenthashmap-java-collections-interview-question.html" \t "_blank)  
  
**Q24 Arrange the following in the ascending order (performance):**  
**HashMap , Hashtable , ConcurrentHashMap and Collections.SynchronizedMap**  
  
Hashtable  <  Collections.SynchronizedMap  <  ConcurrentHashMap  <  HashMap

**Q25 How HashMap works in Java ?**  
  
This is one of the most important question for java developers. HashMap  works on the principle of Hashing . Find detailed information here to understand [what is hashing and how hashmap works in java](http://javahungry.blogspot.co.uk/2013/08/hashing-how-hash-map-works-in-java-or.html) .  
  
**Q26 What is the difference between LinkedList and ArrayList in Java ?**  
  
Main differences between LinkedList and ArrayList are :  
a. LinkedList is the doubly linked list implementation of list interface , while , ArrayList is the resizable array implementation of list interface.  
b. LinkedList can be traversed in the reverse direction using descendingIterator() method  provided by the Java Api developers , while , we need to implement our own method to traverse ArrayList in the reverse direction . find the detailed explanation here [ArrayList vs LinkedList in java](http://javahungry.blogspot.co.uk/2015/04/difference-between-arraylist-and-linkedlist-in-java-example.html" \t "_blank).  
  
**Q27 What are Comparable and Comparator interfaces ? List the difference between them ?**

We already explained what is comparable and comparator interface in detail along with examples here,  [Comparable vs Comparator in Java](http://javahungry.blogspot.com/2013/08/difference-between-comparable-and.html" \t "_blank)

**Q28 Why Map interface does not extend the Collection interface in Java Collections Framework ?**  
One liner answer : **Map interface is not compatible with the Collection interface.**  
Explanation : Since Map requires key as well as value , for example , if we want to add key-value pair then we will use put(Object key , Object value) . So there are two parameters required to add element to the HashMap object  . In Collection interface add(Object o) has only one parameter.   
The other reasons are Map supports valueSet , keySet as well as other appropriate methods which have just different views from the Collection interface.  
  
**Q29 When to use ArrayList and when to use LinkedList in application?**  
ArrayList has constant time search operation O(1) .Hence, ArrayList is preferred when there are more get() or search operation .  
  
Insertion , Deletion operations take constant time O(1) for LinkedList. Hence, LinkedList is preferred when there are more insertions or deletions involved in the application.  
  
**Q30 Write the code for iterating the list in different ways in java ?**  
There are two ways to iterate over the list in java :  
a. using Iterator  
b. using for-each loop

**Topic 21: Java Heap and Slack Memory**

1. Heap memory is used by all the parts of the application whereas stack memory is used only by one thread of execution.
2. Whenever an object is created, it's always stored in the Heap space and stack memory contains the reference to it.
3. **Java Heap Space:** Java Heap space is used by java runtime to allocate memory to Objects and JRE classes. Whenever we create any object, it’s always created in the Heap space.
4. Garbage Collection runs on the heap memory to free the memory used by objects that doesn’t have any reference. Any object created in the heap space has global access and can be referenced from anywhere of the application.
5. **Java Stack Memory:** Java Stack memory is used for execution of a thread. They contain method specific values that are short-lived and references to other objects in the heap that are getting referred from the method.
6. Stack memory is always referenced in LIFO (Last-In-First-Out) order. Whenever a method is invoked, a new block is created in the stack memory for the method to hold local primitive values and reference to other objects in the method. As soon as method ends, the block becomes unused and become available for next method.
7. Stack memory size is very less compared to Heap memory.
8. **Difference between Java Heap Space and Stack Memory**

Ans. Following differences between Heap and Stack memory.

* 1. Heap memory is used by all the parts of the application whereas stack memory is used only by one thread of execution.
  2. 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.
  3. Objects stored in the heap are globally accessible whereas stack memory can’t be accessed by other threads.
  4. Memory management in stack is done in LIFO manner whereas it’s more complex in Heap memory because it’s used globally. Heap memory is divided into Young-Generation, Old-Generation etc, more details at Java Garbage Collection.
  5. Stack memory is short-lived whereas heap memory lives from the start till the end of application execution.
  6. We can use -Xms and -Xmx JVM option to define the startup size and maximum size of heap memory. We can use -Xss to define the stack memory size.
  7. When stack memory is full, Java runtime throws java.lang.StackOverFlowError whereas if heap memory is full, it throws java.lang.OutOfMemoryError: Java Heap Space error.
  8. Stack memory size is very less when compared to Heap memory. Because of simplicity in memory allocation (LIFO), stack memory is very fast when compared to heap memory.
  9. The main difference between heap and stack is that stack memory is used to store local variables and function call while heap memory is used to store objects in Java. No matter, where the object is created in code e.g. as a member variable, local variable or class variable,  they are always created inside heap space in Java.
  10. If you are using Recursion, on which method calls itself, You can quickly fill up stack memory. Another difference between stack and heap is that size of stack memory is a lot lesser than the size of  heap memory in Java.
  11. Variables stored in stacks are only visible to the owner Thread while objects created in the heap are visible to all thread. In other words, stack memory is kind of private memory of Java Threads while heap memory is shared among all threads.

1. The JVM divided the memory into following sections.

1. Heap

2. Stack

3. Code

4. Static

This division of memory is required for its effective management.

* The **code**section contains your **bytecode**.
* The **Stack**section of memory contains **methods, local variables and reference variables.**
* The **Heap**section contains **Objects**(may also contain reference variables).
* The **Static**section contains **Static data/methods**.

1. **Instance variable** are declared **inside a class but not inside a method**

class Student{

int num; // num is  instance variable

public void showData{}

1. **Local variable** are declared **inside**a **method including**method **arguments**.

public void sum(int a){

int x = int a +  3;

// a , x are local variables</strong>

}

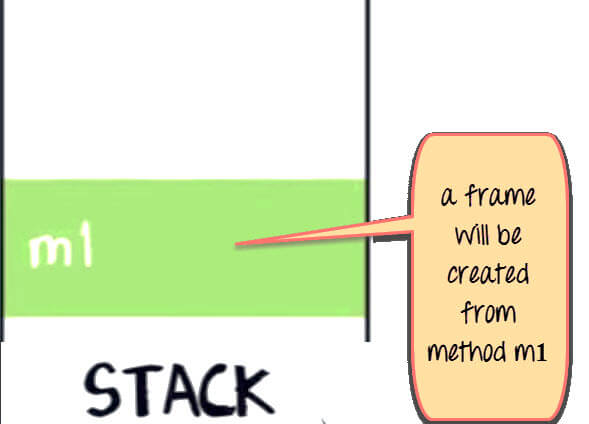
1. Consider that your main method calling method m1

public void m1{

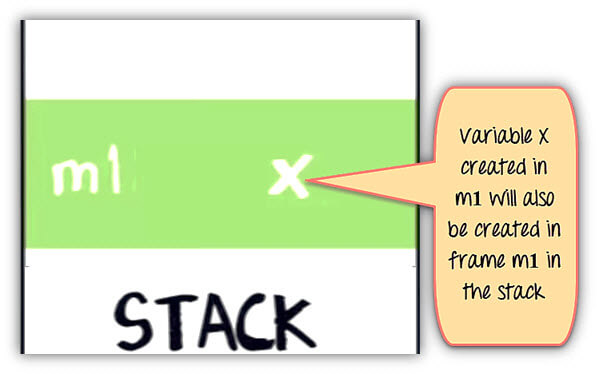
int x=20

}

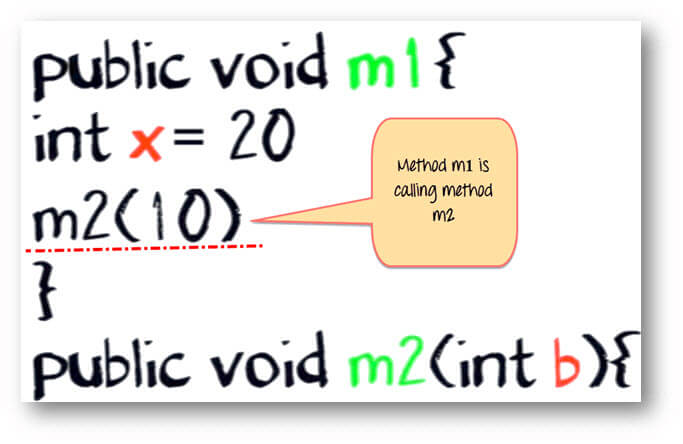
In the stack java, a frame will be created from method m1.

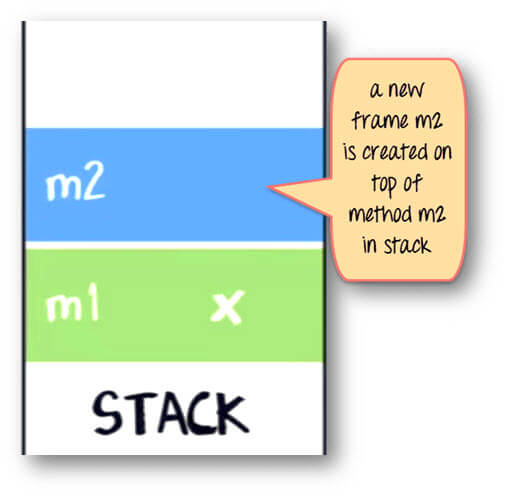
[](https://cdn.guru99.com/images/java/052016_0744_JavaStackan1.jpg)

The variable X in m1 will also be created in the frame for m1 in the stack. (See image below).

[](https://cdn.guru99.com/images/java/052016_0744_JavaStackan2.jpg)

Method m1 is calling method m2. In the stack java, a new frame is created for m2 on top of the frame m1.

[](https://cdn.guru99.com/images/java/052016_0744_JavaStackan3.jpg)

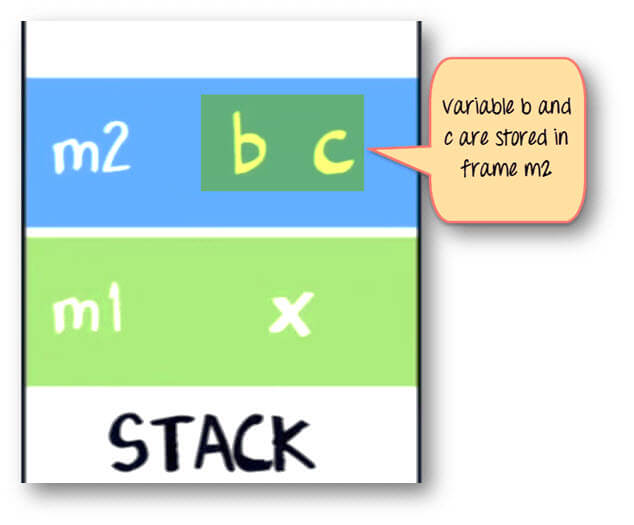
[](https://cdn.guru99.com/images/java/052016_0744_JavaStackan4.jpg)

Variable b and c will also be created in a frame m2 in a stack.

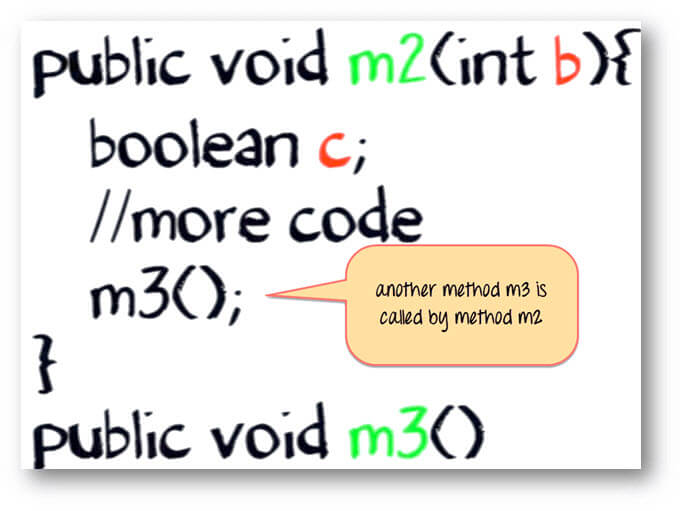
public void m2(int b){

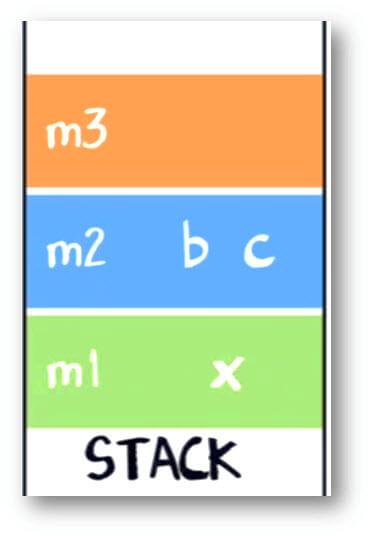
boolean c;

}

[](https://cdn.guru99.com/images/java/052016_0744_JavaStackan5.jpg)

Same method m2 is calling method m3. Again a frame m3 is created on the top of the stack (see image below).

[](https://cdn.guru99.com/images/java/052016_0744_JavaStackan6.jpg)

[](https://cdn.guru99.com/images/java/052016_0744_JavaStackan7.jpg)

Now let say our method m3 is creating an object for class "Account," which has two instances variable int p and int q.

Account {

Int p;

Int q;

}

Here is the code for m3

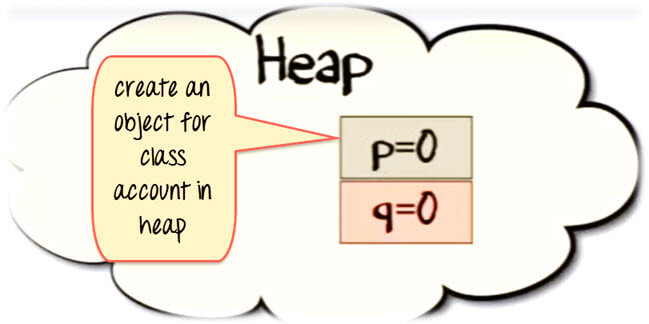
public void m3(){

Account ref = new Account();

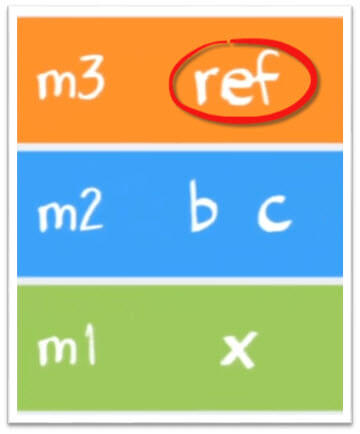
// more code

}

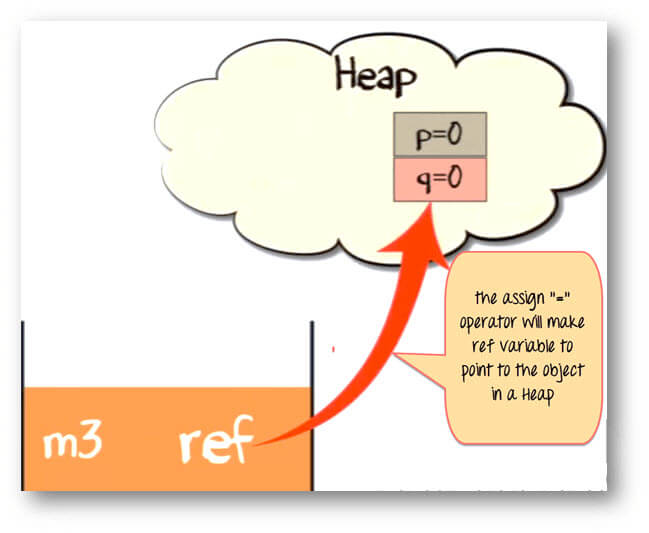
The statement new Account() will create an object of account in heap.

[](https://cdn.guru99.com/images/java/052016_0744_JavaStackan8.jpg)

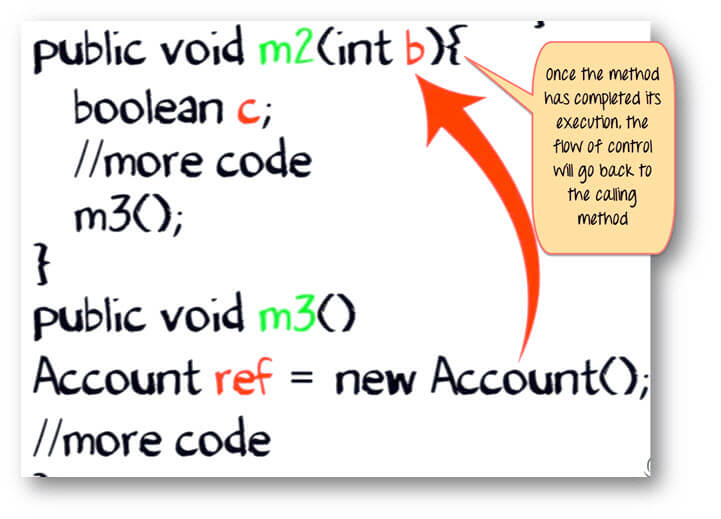
The reference variable "ref" will be created in a stack java.

[](https://cdn.guru99.com/images/java/052016_0744_JavaStackan9.jpg)

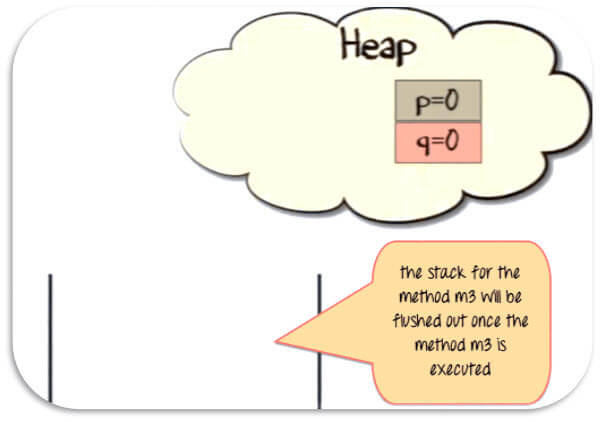
The assign "=" operator will make a reference variable to point to the object in the Heap.

[](https://cdn.guru99.com/images/java/052016_0744_JavaStackan10.jpg)

Once the method has completed its execution. The flow of control will go back to the calling method. Which in this case is m2.

[](https://cdn.guru99.com/images/java/052016_0744_JavaStackan11.jpg)

The stack from method m3 will be flushed out.

[](https://cdn.guru99.com/images/java/052016_0744_JavaStackan12.jpg)

Since the reference variable will no longer be pointing to the object in the heap, it would be eligible for garbage collection.

[](https://cdn.guru99.com/images/java/052016_0744_JavaStackan13.jpg)

Once m2 has completed its execution. It will be poped out of the stack, and all its variable will be flushed and no longer be available for use.

Likewise for method m1.

Eventually, the flow of control will return to the startpoint of the program. Which usually, is the "main" method.

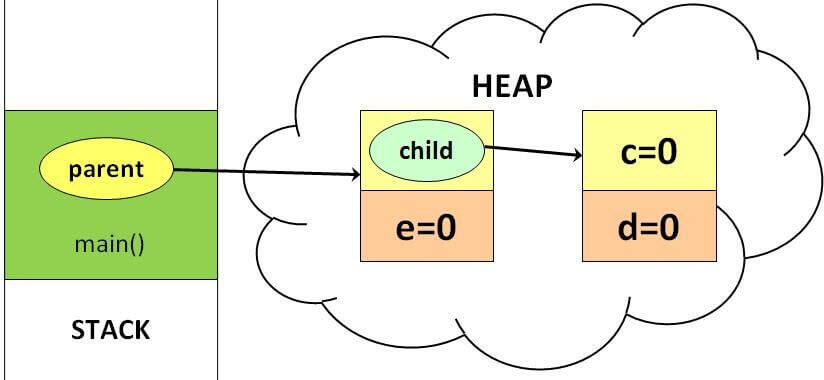
**Summary:**

* When a method is called , a frame is created on the top of stack.
* Once a method has completed execution , flow of control returns to the calling method and its corresponding stack frame is flushed.
* Local variables are created in the stack
* Instance variables are created in the heap & are part of the object they belong to.
* Reference variables are created in the stack.

**Point to Ponder:**What if Object has a reference as its instance variable?

public static void main(String args[]){ A parent = new A(); //more code } class A{ B child = new B(); int e; //more code } class B{ int c; int d; //more code }

In this case , the reference variable "child" will be created in heap ,which in turn will be pointing to its object, something like the diagram shown below.

[](https://cdn.guru99.com/images/java/052016_0744_JavaStackan14.jpg)

## What Is Java Stack?

A Java stack is part of your computer’s memory where temporary variables, which are created by all functions you do, are stored. It is used to execute a thread and may have certain short-lived values as well as references to other objects. It uses LIFO data structure, or [last in first out](http://www.journaldev.com/4098/java-heap-space-vs-stack-memory" \t "_blank).

What does this mean? When a method is invoked, it creates a new block in the stack for that particular method. The new block will have all the local values, as well as references to other objects that are being used by the method. When the method ends, the new block will be erased and will be available for use by the next method. The objects you find here are only accessible to that particular function and will not live beyond it.

This makes it very easy to keep track of the stack, where the latest reserved block is also the first to be freed. The variables created for the method are directly stored in the memory, allowing for fast access.

The memory size of a Java stack is generally much less than in a Java heap space because when a method ends, all the variables created on the stack are erased forever.

## What Is Java Heap?

Java objects are in an area, which is called the heap. It is created when the program is run, and its size may decrease or increase as your program runs. It can easily become full, and when it does, garbage collection is initiated. This is when objects that are no longer used are deleted to make way for new objects.

Unlike in a Java stack where memory allocation is done when your program is compiled, in a heap it is allocated as your program is run. Accessing variables placed here is a bit slower compared to a stack’s direct and fast access.

Heap is likened to a [global memory pool](https://www.quora.com/How-is-java-memory-pool-divided" \t "_blank). A method or function will use the heap for memory allocation if you need the data or variables to live longer than the method or function in question. The objects you find here are accessible to all the functions.

Also, there is no specific order in reserving blocks in a heap. You can allocate blocks at any time, and then you can free it when you wish. As you can imagine, it is much more complex to keep track of the parts that are free and can be allocated, but it can also be divided into two generations or sub-areas.

These sub-areas are called the young space (or nursery) and the old space. The young space is typically earmarked for the memory allocation for new objects. When the young space becomes full, [garbage collection](https://stackify.com/what-is-java-garbage-collection/" \t "_blank) happens. Short-lived or temporary objects typically use the young space. This help makes garbage collection faster when compared to a heap without any divisions.

## Similarities and Differences Between Stack and Heap

Both are ways that [Java allocates memory](http://net-informations.com/faq/net/stack-heap.htm" \t "_blank) and both are stored in the RAM. However, to make things easier to remember, heap is used for dynamic memory allocation, while stack is for static allocations.

**Where is it stored?**Variables that are allocated on the stack are accessible directly from memory, and as such, these can run very fast. Accessing objects on the heap, on the other hand, takes more time.

**When does the allocation happen?**On the stack, memory allocation happens when the program is compiled. Meanwhile, on the heap, it begins when the program is run.

And since this is the case, you would need to know just how much data and memory you are going to need before compiling if you want to use the stack. Another limitation that the stack has is that it cannot handle big chunks of variables that need a lot of memory. If you do not know how much data you are going to need at run time or if you need memory for a lot of data, then you need to use heap.

## In a Nutshell…

### ****Stack****

* The size of the stack will vary as methods and functions create and delete local variables as needed.
* Memory is allocated and then subsequently freed without you needing to manage the memory allocation.
* Stack has size limits, which can vary according to the operating system you use.
* Variables that are stored on the stack exist for as long as the function that created them are running.

### ****Heap****

* Memory is not managed automatically nor is it as tightly managed by the central processing unit the way stack is managed. You would need to free allocated memory yourself when these blocks are no longer needed.
* The heap is prone to memory leaks, where memory is allocated to unused objects and will not be available to processes other than that.
* There is no size limit in the heap.
* Compared to stack, objects in the heap are much slower to access. It is also slower to write to the memory on the heap.

Stack is easier and faster to use, but it comes with a lot of limitations that you can ignore if you use heap.

**[When do you use stack?](https://softwareengineering.stackexchange.com/questions/65281/stack-and-heap-memory-in-java" \t "_blank)** Stack can only be used for local variables that use up small amounts of memory. The good news is that memory allocation and management is not going to be your problem and access to these objects is very fast. It does suffer from size limitations and the fact that you cannot resize variables on the stack.

**[When do you use heap?](https://softwareengineering.stackexchange.com/questions/65281/stack-and-heap-memory-in-java" \t "_blank)** You use the heap to allocate memory if there are variables that you need to be accessed globally, as opposed to just being available only to the methods and functions that created it. Heap is also good when you have a need for a lot of memory since it has no limit on memory size. You can also resize the variables on the heap.

**Topic: 23 Class Loader in java**

1. Because class loaders exist, the Java run time does not need to know anything about files and file systems when running Java programs.
2. Classes are introduced into the Java environment when they are referenced by name in a class that is already running. There is a bit of magic that goes on to get the first class running (which is why you have to declare the *main()* method as static, taking a string array as an argument), but once that class is running, future attempts at loading classes are done by the class loader.
3. At its simplest, a class loader creates a flat name space of class bodies that are referenced by a string name. The method definition is:

Class r = loadClass(String className, Boolean resolveIt)

1. The variable *className* contains a string that is understood by the class loader and is used to uniquely identify a class implementation. The variable *resolveIt* is a flag to tell the class loader that classes referenced by this class name should be resolved (that is, any referenced class should be loaded as well)
2. All Java virtual machines include one class loader that is embedded in the virtual machine. This embedded loader is called the primordial class loader. It is somewhat special because the virtual machine assumes that it has access to a repository of *trusted classes* which can be run by the VM without verification.
3. The primordial class loader implements the default implementation of *loadClass()*. Thus, this code understands that the class name **java.lang.Object** is stored in a file with the prefix java/lang/Object.class somewhere in the class path. This code also implements both class path searching and looking into zip files for classes. The really cool thing about the way this is designed is that Java can change its class storage model simply by changing the set of functions that implements the class loader.
4. The primordial class loader is implemented primarily in the functions *FindClassFromClass* and *ResolveClass*.

## Building a SimpleClassLoader

A class loader starts by being a subclass of java.lang.ClassLoader. The only abstract method that must be implemented is loadClass(). The flow of loadClass() is as follows:

* Verify class name.
* Check to see if the class requested has already been loaded.
* Check to see if the class is a "system" class.
* Attempt to fetch the class from this class loader's repository.
* Define the class for the VM.
* Resolve the class.
* Return the class to the caller.

**Topic 24: Threads in java**

Definition - What does Thread mean?

A thread, in the context of Java, is the path followed when executing a program. All Java programs have at least one thread, known as the main thread, which is created by the JVM at the program’s start, when the main() method is invoked with the main thread. In Java, creating a thread is accomplished by implementing an interface and extending a class. Every Java thread is created and controlled by the java.lang.Thread class.

When a thread is created, it is assigned a priority. The thread with higher priority is executed first, followed by lower-priority threads.

>>>The JVM stops executing threads under either of the following conditions:

>>>If the exit method has been invoked and authorized by the security manager

All the daemon threads of the program have died

Java is a multi-threaded application that allows multiple thread execution at any particular time.

In a single-threaded application, only one thread is executed at a time because the application or program can handle only one task at a time.

For example, a single-threaded application may allow for the typing of words. However, this single thread requires an additional single thread allowing for the recording of keystrokes in order to type the words.

Thus, a single-threaded application records the keystrokes, allowing the next single-threaded application (the typing of words) to follow.

However, a multi-threaded application allows for the handling of both tasks (recording and typing the keystrokes) within one application.

Multithreading refers to two or more tasks executing concurrently within a single program.

A thread is an independent path of execution within a program. Many threads can run concurrently within a program. Every thread in Java is created and controlled by the java.lang.Thread class.

Multithreading has several advantages over Multiprocessing such as;

Threads are lightweight compared to processes

Threads share the same address space and therefore can share both data and code

Context switching between threads is usually less expensive than between processes

Cost of thread intercommunication is relatively low that that of process intercommunication

Threads allow different tasks to be performed concurrently.

The following figure shows the methods that are members of the Object and Thread Class.

There are two ways to create thread in java;

Implement the Runnable interface (java.lang.Runnable)

By Extending the Thread class (java.lang.Thread)

By definition, multitasking is when multiple processes share common processing resources such as a CPU. Multi-threading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads.

Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application.

Life Cycle of a Thread

======================

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. The following diagram shows the complete life cycle of a thread.

Following are the stages of the life cycle −

New − A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.

Runnable − After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.

Waiting − Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.

Timed Waiting − A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.

Terminated (Dead) − A runnable thread enters the terminated state when it completes its task or otherwise terminates

Thread Priorities

=================

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Java thread priorities are in the range between MIN\_PRIORITY (a constant of 1) and MAX\_PRIORITY (a constant of 10). By default, every thread is given priority NORM\_PRIORITY (a constant of 5).

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and are very much platform dependent.

Thread Objects

Each thread is associated with an instance of the class Thread. There are two basic strategies for using Thread objects to create a concurrent application.

To directly control thread creation and management, simply instantiate Thread each time the application needs to initiate an asynchronous task.

To abstract thread management from the rest of your application, pass the application's tasks to an executor.

But we use multithreading than multiprocessing because threads share a common memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation etc.

Advantages of Java Multithreading

================================

1) It doesn't block the user because threads are independent and you can perform multiple operations at same time.

2) You can perform many operations together so it saves time.

3) Threads are independent so it doesn't affect other threads if exception occur in a single thread.

Life cycle of a Thread (Thread States)

======================================

Life cycle of a thread

New

Runnable

Running

Non-Runnable (Blocked)

Terminated

A thread can be in one of the five states. According to sun, there is only 4 states in thread life cycle in java new, runnable, non-runnable and terminated. There is no running state.

But for better understanding the threads, we are explaining it in the 5 states.

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

New

Runnable

Running

Non-Runnable (Blocked)

Terminated

1) New

The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

2) Runnable

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

3) Running

The thread is in running state if the thread scheduler has selected it.

4) Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

5) Terminated

A thread is in terminated or dead state when its run() method exits.

# Thread Scheduler in Java

**Thread scheduler** in java is the part of the JVM that decides which thread should run.

There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.

Only one thread at a time can run in a single process.

The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads

Under preemptive scheduling, the highest priority task executes until it enters the waiting or dead states or a higher priority task comes into existence. Under time slicing, a task executes for a predefined slice of time and then reenters the pool of ready tasks. The scheduler then determines which task should execute next, based on priority and other factors.

**Thread Scheduling**. ... Hence, **threads** run one at a time in such a way as to provide an illusion of concurrency. Execution of multiple **threads** on a single CPU in some order is called **scheduling**. The Java runtime environment supports a very simple, deterministic **scheduling** algorithm called fixed-priority **scheduling**.

JAVA Thread Scheduling

Features :

1. The JVM schedules using a preemptive , priority � based scheduling algorithm.
2. All Java threads have a priority and the thread with he highest priority is scheduled to run by the JVM.
3. In case two threads have the same priority a FIFO ordering is followed.

 A different thread is invoked to run in case one of the following events occur:

1.The currently running thread exits the Runnable state ie either blocks or terminates.

2. A thread with a higher priority than the thread currently running enters the Runnable state. The lower priority thread is preempted and the higher priority thread is scheduled to run.

Time Slicing is dependent on the implementation.

A thread can voluntarily yield control through the yield() method. Whenever a thread yeilds control of the CPU another thread of the same priority is scheduled to run. A thread voluntarily yielding control of the CPU is called Cooperative Multitasking.

Sometimes we need to execute a task periodically or after specific delay. Java provides [**Timer Class**](https://www.journaldev.com/1050/java-timer-timertask-example)through which we can achieve this but sometimes we need to run similar tasks in parallel. So creating multiple Timer objects will be an overhead to the system and it’s better to have a thread pool of scheduled tasks.

Java provides scheduled thread pool implementation through ScheduledThreadPoolExecutor class that implements ScheduledExecutorService interface. ScheduledExecutorService defines the contract methods to schedule a task with different options.

**Thread Scheduler in Java**. **Thread scheduler in java** is the part of the JVM that decides which **thread** should run. ... Only one **thread** at a time can run in a single process. The **thread scheduler** mainly uses preemptive or time slicing **scheduling**to schedule the **threads**.

In Java program, you create threads but they are not executed by Java alone. Java takes the help of the underlying OS to execute them. To allocate microprocessor time and to supervise all the threads' execution, the OS comes with **Thread Scheduler**. The entire responsibility of maintaining the sequence of execution of threads, where which thread should be given first preference than the other, lies with the thread scheduler. The scheduling depends on the **algorithm** of the scheduler. Many types of algorithms exist like **preemptive** and **time slicing**with **round robin** etc. It is a very complex algorithm that executes many times in a given time.

###### Java Thread Scheduler

The scheduler maintains a pool of threads. When Java thread is started calling**start()** method, it joins the pool of waiting threads. For deciding processor allocation for each waiting thread, the scheduler takes many aspects into consideration.

**1.** Priority of thread  
**2.** Waiting time of thread  
**3.**Nature of thread

The JVM is based on **preemptive and priority based** scheduling algorithm. The thread with more priority is given first preference than the thread with less priority. The thread with more priority relinquishes (empties) the thread with less priority that is being executed. If the threads of equal priority are in the pool, the waiting time is taken in consideration. Nature of threads sometimes affects. The **daemon threads**are given less importance and are executed only when no other thread is available for execution.

When JVM starts executing a Java program, it creates a few threads for the execution.

1. **main** is a method for us, but main is a thread for JVM. The execution starts with main thread.

2. **Garbage collector** is a daemon thread that comes into action before every thread.

3. **Event Dispatcher** is a thread which will take care of events raised by the components like click of a button etc.

4. There is one more, **Timer** thread, which maintains the time for methods like **sleep()** etc.

When a thread is started with **start()** method, the Java thread joins the **pool of waiting threads**. It does not mean the scheduler should give the processor time immediately. "Joins" means, it is guaranteed of getting processor time, but at what time depends on the decision of the scheduler managed by the OS. When a thread joins, the scheduler is executed by the OS and makes a table of threads' sequence to allocate processor time. When a **sleep()**method is called, the thread is removed from the active pool, then again the scheduler is executed to make the table as there comes imbalance in the waiting threads. And executed again when the thread joins the pool when the **sleep()**time is over. For this reason, the scheduler executes a number of times when a thread joins, **yields**, **stops** or leaves (when blocked or dies) the pool.

# Thread Scheduling

In our introduction to [how threads work](http://www.javamex.com/tutorials/threads/how_threads_work.shtml" \t "_blank), we introduced the **thread scheduler**, part of the OS (usually) that is responsible for sharing the available CPUs out between the various threads. How exactly the scheduler works depends on the individual platform, but various modern operating systems (notably Windows and Linux) use largely similar techniques that we'll describe here. We'll also mention some key varitions between the platforms.

Note that we'll continue to talk about a single thread scheduler. On multiprocessor systems, there is generally some kind of scheduler *per processor*, which then need to be coordinated in some way. (On some systems, switching on different processors is staggered to avoid contention on shared scheduling tables.) Unless otherwise specified, we'll use the term *thread scheduler* to refer to this overall system of coordinated per-CPU schedulers.

Across platforms, thread scheduling1 tends to be based on at least the following criteria:

* a **priority**, or in fact usually multiple "priority" settings that we'll discuss below;
* a **quantum**, or number of allocated **timeslices** of CPU, which essentially determines the amount of CPU time a thread is allotted before it is forced to yield the CPU to another thread of the same or lower priority (the system will keep track of the *remaining* quantum at any given time, plus its *default* quantum, which could depend on thread type and/or system configuration);
* a **state**, notably "runnable" vs "waiting";
* **metrics** about the behaviour of threads, such as recent CPU usage or the time since it last *ran* (i.e. had a share of CPU), or the fact that it has "just received an event it was waiting for".

Most systems use what we might dub **priority-based round-robin** scheduling to some extent. The general principles are:

* a thread of **higher priority** (which is a function of base and local priorities) will **preempt** a thread of lower priority;
* otherwise, threads of **equal priority** will essentially **take turns** at getting an allocated slice or **quantum** of CPU;
* there are a few extra "tweaks" to make things work.

### States

Depending on the system, there are various **states** that a thread can be in. Probably the two most interesting are:

* **runnable**, which essentially means "ready to consume CPU"; being runnable is generally the **minimum** requirement for a thread to actually be scheduled on to a CPU;
* **waiting**, meaning that the thread currently cannot continue as it is waiting for a resource such as a lock or I/O, for memory to be paged in, for a signal from another thread, or simply for a period of time to elapse (*sleep*).

Other states include *terminated*, which means the thread's code has finished running but not all of the thread's resources have been cleared up, and a *new* state, in which the thread has been created, but not all resources necessary for it to be runnable have been created. Internally, the OS may distinguish between various different types of wait states2 (for example "waiting for a signal" vs "waiting for the stack to be paged in"), but this level of granularity is generally not available or so important to Java programs. (On the other hand, Java generally exposes to the programmer things the JVM can reasonly know about, for example, if a thread is waiting to acquire the lock on a Java object— roughly speaking, "entering a synchronized block".)