**Core Java basics:**

**\*What is Java?**

[Java](https://www.javatpoint.com/java-tutorial) is the high-level, [object-oriented](https://www.javatpoint.com/java-oops-concepts), robust, secure programming language, platform-independent, high performance, Multithreaded, and portable programming language. It was developed by [**James Gosling**](https://www.javatpoint.com/james-gosling-father-of-java) in June 1991. It can also be known as the platform as it provides its own JRE and API.

**\*Difference between final and finally and finalize?**

**final**

It is a *keyword* that is used to apply restrictions on a class, method, or variable.

* The class with this keyword *cannot* be inherited.
* The method with this keyword *cannot* be overridden.
* The variable with this keyword *cannot* be changed.

class HelloWorld {

    public static void main( String args[] ) {

        final int val=150;

        val=100;

   }

}

**finally**

* In Java, **finally** is a block used to place important code that will be executed whether or not an exception is handled.

class HelloWorld {

    public static void main( String args[] ) {

      try{

      int val=150;

      }catch(Exception e){

System.out.println(e);}

      finally{System.out.println("finally block!");}

    }

}

**finalize**

**finalize()** is used to perform clean-up processing just before the object is collected by the garbage collector. In Java, the finalize method in a class is used for freeing up the heap’s memory, just like destructors in C++.

Note: finalize is deprecated in Java 9.

class HelloWorld {

    public static class test

    {

        int val = 50;

        @Override

        protected void finalize() throws Throwable

        {

            System.out.println("Finalize Method");

        }

    }

   public static void main(String[] args)

   {

      test a1 = new test();

      test a2 = new test();

      a1 = a2;

      // calling finalize method

      System.gc();

   }

}

**\*Difference == & equals method?**

In java both == and equals() method is used to check the equality of two variables or objects.

| **Sr. No.** | **Key** | **==** | **equals() method** |
| --- | --- | --- | --- |
| 1 | Type | == is an operator. | equals() is a method of Object class. |
| 2 | Comparision | == should be used during reference comparison. == checks if both references points to same location or not. | equals() method should be used for content comparison. equals() method evaluates the content to check the equality. |
| 3 | Object | == operator can not be overriden. | equals() method if not present and Object.equals() method is utilized, otherwise it can be overridden. |

public class JavaTester {

   public static void main(String args[]) {

      String s1 = new String("TUTORIALSPOINT");

      String s2 = new String("TUTORIALSPOINT");

      //Reference comparison

      System.out.println(s1 == s2);

      //Content comparison

      System.out.println(s1.equals(s2));

      // integer-type

      System.out.println(10 == 10);

      // char-type

      System.out.println('a' == 'a');

   }

}

### Output

false

true

true

true

**\*What is immutable?**

**Immutable** objects are instances whose state doesn't change after it has been initialized. For **example**, String is an **immutable** class and once instantiated its value never changes.

**\*How can we create object as immutable?**

* Don't add any setter method.
* Declare all fields final and private.
* If a field is a **mutable object create** defensive copies of it for getter methods.
* If a **mutable object** passed to the constructor must be assigned to a field **create** a defensive copy of it.

**Immutable class** means that once an **object** is created, **we** cannot change its content. In **Java**, all the wrapper **classes** (like Integer, Boolean, Byte, Short) and String **class** is **immutable**. **We can create** our own **immutable class** as well. ... The **class** must be declared as final (So that child **classes can**'t **be created**)

Benefits of immutable object

* They are much easier to cache since the modification on the object is not going affect it's original state.
* Truly immutable objects are always thread-safe
* They help to avoid temporal coupling

**\*What is aggregation and composition?**

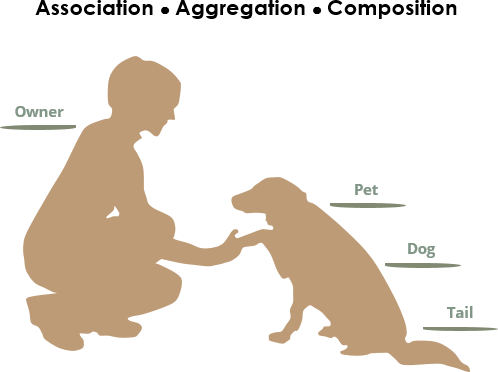
Aggregation represents the weak relationship whereas composition represents the strong relationship. For example, the bike has an indicator (aggregation), but the bike has an engine (composition).

**If two classes in a model need to communicate with each other, there must be a link between them, and that can be represented by an association (connector).**

**Aggregation** implies a relationship where the child can exist independently of the parent. Example: Class (parent) and Student (child). Delete the Class and the Students still exist.

**Composition** implies a relationship where the child cannot exist independent of the parent. Example: House (parent) and Room (child).

Consider the differences and similarities between the classes of the following objects: **pets**, **dogs**, **tails**, **owners**.



We see the following relationships:

* owners feed pets, pets please owners (**association**)
* a tail is a part of both dogs and cats (**aggregation** / **composition**)
* a cat is a kind of pet (**inheritance** / **generalization**)

**\*What is atomic and non-atomic operation?**

**Atomic operations** are take place in one step. ... Like read and write **operation** of variable. **atomic operations** cannot be interrupted and they are thread safe, In **java** read and write **operations** are **atomic** for all variables which are less or equal to 32 bit

It's **not atomic** because it's a multiple-step operation at the machine code level. That is, longs and doubles are longer than the processor's word length. **Java long and double are not atomic** in 32 bit machines, but **atomic** in 64 bit machines with some of the 64 bit JVMs

**\*What is mean by purely object oriented programming language?**

**Pure Object Oriented Language** or Complete **Object Oriented Language** are **Fully Object Oriented Language** which supports or have features which treats everything inside program as **objects**. ... All user defined types are **objects**. All operations performed on **objects** must be only through methods exposed at the **objects**.

**Java** is **purely object oriented programming language** because without class and **object** it is impossible to write any **Java** program. **Java** is not **pure object oriented programming language**. because **java** supports non-primitive datatypes like int ,float ,boolean,double,long etc. It compulsory need a **object**.

**\*Describe all oops concepts?**

**Abstraction.**

Abstraction means using simple things to represent complexity. We all know how to turn the TV on, but we don’t need to know how it works in order to enjoy it. In Java, abstraction means simple things like **objects**, **classes**, and **variables** represent more complex underlying code and data. This is important because it lets avoid repeating the same work multiple times.

abstract class Animal {

// abstract methods

abstract void move();

abstract void eat();

// concrete method

void label() {

System.out.println("Animal's data:");

}

}

interface Animal {

public void eat();

public void sound();

}

interface Bird {

int numberOfLegs = 2;

String outerCovering = "feather";

public void fly();

}

**Encapsulation.**

This is the practice of keeping fields within a class private, then providing access to them via public methods. It’s a protective barrier that keeps the data and code safe within the class itself. This way, we can re-use objects like code components or variables without allowing open access to the data system-wide.

class Animal {

private String name;

// Getter methods

public String getName() {

return name;

}

// Setter methods

public void setName(String name) {

this.name = name;

}

}

**Inheritance.**

This is a special feature of Object Oriented Programming in Java. It lets programmers create new classes that share some of the attributes of existing classes. This lets us build on previous work without reinventing the wheel.

class Bird {

public String reproduction = "egg";

public String outerCovering = "feather";

public void flyUp() {

System.out.println("Flying up...");

}

public void flyDown() {

System.out.println("Flying down...");

}

}

class Eagle extends Bird {

public String name = "eagle";

public int lifespan = 15;

}

**Polymorphism.**

This Java OOP concept lets programmers use the same word to mean different things in different contexts. One form of polymorphism in Java is **method overloading**. That’s when different meanings are implied by the code itself. The other form is **method overriding**. That’s when the different meanings are implied by the values of the supplied variables. See more on this below.

Static Polymorphism: don’t have to extend Animal class just implement bird class methods

Dynamic Polymorphism: can extend class and use both bird and animal class methods.

class Animal {

public void eat() {

System.out.println("This animal eats insects."); }}

class Bird extends Animal {

public void eat() {

System.out.println("This bird eats seeds.");

}

}

**\* What is object?**

**Object** − **Objects** have states and behaviors. **Example**: A dog has states - color, name, breed as well as behaviors – wagging the tail, barking, eating.

An **object** is an instance of a class. Class − A class can be defined as a template/blueprint that describes the behavior/state that the **object** of its type support.

**\*What is string buffer?**

**String buffer** is mutable classes which can be used to do operation on **string** objects such as reverse of **string**, concating **string** and etc.

We can modify **string** without creating a new object of the **string**.

A **string buffer** is thread-safe.

**\*String buffer vs string builder vs string?**

The **String** class is an immutable.

**String buffer and StringBuilder** both are mutable classes, which can be used to do operation on **string** objects such as reverse of **string**, concating **string** and etc.

We can modify **string** without creating a new object of the **string**.

A **string buffer** is thread-safe whereas **string builder** is not thread-safe

**\*What is Static?**

The **static** is keyword in java, indicates that the particular member belongs to a type itself, rather than to an instance of that type. This means that only one instance of that **static** member is created which is shared across all instances of the class.

**\*How serialization works internally?**

**Serialization** is a mechanism of converting the state of an object into a byte stream.

**Deserialization** is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object



The byte stream created is platform independent. So, the object serialized on one platform can be deserialized on a different platform.

**To make a Java object serializable we implement** the **java.io.Serializable** interface.  
The ObjectOutputStream class contains **writeObject()** method for serializing an Object.

public final void writeObject(Object obj) throws IOException

The ObjectInputStream class contains **readObject()** method for deserializing an object.

public final Object readObject() throws IOException, ClassNotFoundException

**\*What are marker interface?**

A **marker interface** is an **interface** that has no methods or constants inside it. It provides run-time type information about objects, so the compiler and JVM have additional information about the object. A **marker interface** is also called a tagging **interface**.

Examples of marker interface are Serializable, Clonnable and Remote interface. All these interfaces are empty interfaces. Examples of Marker Interface which are used in real-time applications : Cloneable interface : Cloneable interface is present in **java**.

**\*What is Java Composition?**

The **composition** is a design technique in **java** to implement a has-a relationship. **Java** Inheritance is used for code reuse purposes and the same we can do by using **composition**. The **composition** is achieved by using an instance variable that refers to other objects.

**Composition** allows us to reuse the code. In **Java**, **we** can **use** multiple Inheritance by **using** the **composition** concept. The **Composition** provides better test-ability of a class. **Composition** allows us to dynamically change our program's behavior by changing the member objects at run time.

For **example**: A car has a engine, a window has a button, a zoo has a tiger. **Composition** is a special case of aggregation. In other words, a restricted aggregation is called **composition**.

When an object contains the other object and the contained object cannot exist without the other object, then it is called **composition**

**\*If there is no setter methods in immutable class then how we assign values to variables?**

Initialize all the fields via a contructor.

**\*Difference between continue and break?**

The main **difference between break** and **continue** is that **break** is used for immediate termination of loop. On the other hand, '**continue**' terminate the current iteration and resumes the control to the next iteration of the loop.

**\*What is class loader in java?**

The **Java ClassLoader** is a part of the **Java** Runtime Environment that dynamically loads **Java classes** into the **Java** Virtual Machine. The **Java** run time system does not need to know about files and file systems because of **classloaders**. **Java classes** aren't loaded into memory all at once, but when required by an application.

**Class loaders** are responsible for loading **Java** classes during runtime dynamically to the JVM (**Java** Virtual Machine). Also, they are part of the JRE (**Java** Runtime Environment). ... This is where **class loaders** come into the picture. They are responsible for loading classes into memory.

**\*What is Static import?**

**Static import** is a feature introduced in the **Java** programming language that allows members (fields and methods) which have been scoped within their container class as public **static** , to be used in **Java** code without specifying the class in which the field has been defined

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

or all the static members of a class:

**import static** **java.lang.Math.\***;

**\*What is reflection?**

**Reflection** is an API which is used to examine or modify the behavior of methods, classes, interfaces at runtime. The required classes for **reflection** are provided under **java**. lang. reflect package. ... Through **reflection** we can invoke methods at runtime irrespective of the access specifier used with them.

**Java Reflection** makes it possible to inspect classes, interfaces, fields and methods at runtime, without knowing the names of the classes, methods etc. at compile time. It is also possible to instantiate new objects, invoke methods and get/set field values using **reflection**.

**\*What is static loading and dynamic loading?**

**Static Class Loading**: Creating objects and instance using new keyword is known as static class loading. The retrieval of class definition and instantiation of the object is done at compile time.  
  
**Dynamic Class Loading**: Loading classes use Class.forName () method. Dynamic class loading is done when the name of the class is not known at compile time.

**\*What is an assertion in java?**

An **assertion** is a statement in **Java** which ensures the correctness of any assumptions which have been done in the program. When an **assertion** is executed, it is assumed to be true. If the **assertion** is false, the JVM will throw an **Assertion** error. It finds it application primarily in the testing purposes.

**\*Can we achieve runtime polymorphism by data members?**

A method is overridden, not the **data members**, so **runtime polymorphism can**'t be achieved by **data members**. ... We are accessing the **data** member by the reference variable of Parent class which refers to the subclass object.

**\*How is runtime polymorphism achieved?**

The **runtime polymorphism is achieved** when the object method is invoked at the **runtime** instead of compile time. It is **achieved** by method overriding which is also known as dynamic binding.

**\*How is polymorphism achieved in Java?**

In **Java**, static **polymorphism** is **achieved** through method overloading. Method overloading means there are several methods present in a class having the same name but different types/order/number of parameters. ... For the third and fourth methods, there is a change of order of parameters.

**\*How do you achieve dynamic polymorphism in Java?**

**Dynamic polymorphism** is **achieved** by a concept called as method overriding in OOP languages like C++ and **Java**. Method overriding is redefining the same method again and again in the derived classes in an inheritance hierarchy of the classes.

### \*Categories Java Design patterns?

Based on problem analysis, we can categorize design patterns into the following categories.

**Creational patterns:**

* Factory method/Template
* Abstract Factory
* Builder
* Prototype
* Singleton

**Structural patterns:**

* Adapter
* Bridge
* Filter
* Composite
* Decorator
* Facade
* Flyweight
* Proxy

**Behavioral patterns:**

* Interpreter
* Template method/ pattern
* Chain of responsibility
* Command pattern
* Iterator pattern
* Strategy pattern
* Visitor pattern

**J2EE patterns:**

* MVC Pattern
* Data Access Object pattern
* Front controller pattern
* Intercepting filter pattern
* Transfer object pattern

**\*Explain the advantages of Java design pattern?**

* The Design Patterns are reusable in multiple projects.
* The Design Patterns provide a solution that helps to define the system architecture.
* The Design Patterns capture software engineering experiences.
* The Design Patterns provide transparency to the design of an application.
* They are testified and well-proved since they have been built upon the knowledge and experience of expert software developers.

**\*What is an agile or agile methodology?**

Agile is an iterative approach of software development methodology using short iterations of 1 to 4 weeks. Due to the agile methodology, the development process is aligned to deliver the changing business requirement.

**\*What are some quality strategies of agile?**

Some quality strategies of agile are:

* Iteration
* Re-factoring
* Dynamic code analysis
* Short feedback cycles
* Reviews and inspection
* Standards and guidelines
* Milestone reviews

### \*What is AJAX?

AJAX stands for Asynchronous JavaScript and XML. It is a group of related technologies used to display data asynchronously. In other words, it sends and retrieves data without reloading the web page.

**Heap :**

* It is a shared runtime data area and stores the actual object in a memory. It is instantiated during the virtual machine startup.
* This memory is allocated for all class instances and array. Heap can be of fixed or dynamic size depending upon the system’s configuration.
* JVM provides the user control to initialize or vary the size of heap as per the requirement. When a new keyword is used, object is assigned a space in heap, but the reference of the same exists onto the stack.
* There exists one and only one heap for a running JVM process.

*Scanner sc = new Scanner(System.in);*

The above statement creates the object of Scanner class which gets allocated to heap whereas the reference ‘sc’ gets pushed to the stack.

***Note:****Garbage collection in heap area is mandatory.*

**Method Area:**

* It is a logical part of the heap area and is created on virtual machine startup.
* This memory is allocated for class structures, method data and constructor field data, and also for interfaces or special method used in class. Heap can be of fixed or dynamic size depending upon the system’s configuration.
* Can be of a fixed size or expanded as required by the computation. Needs not to be contiguous.

***Note:****Though method area is logically a part of heap, it may or may not be garbage collected even if garbage collection is compulsory in heap area.*

**JVM Stacks:**

* A stack is created at the same time when a thread is created and is used to store data and partial results which will be needed while returning value for method and performing dynamic linking.
* Stacks can either be of fixed or dynamic size. The size of a stack can be chosen independently when it is created.
* The memory for stack needs not to be contiguous.

**Native method Stacks:**

Also called as C stacks, native method stacks are not written in Java language. This memory is allocated for each thread when its created. And it can be of fixed or dynamic nature.

**Program counter (PC) registers:**

Each JVM thread which carries out the task of a specific method has a program counter register associated with it. The non native method has a PC which stores the address of the available JVM instruction whereas in a native method, the value of program counter is undefined. PC register is capable of storing the return address or a native pointer on some specific platform.