**Encapsulation:** binding the data(state) with method(behavior). Sensitive data is hidden from the users.

To achieve:

1. Must declare the variables/attributes as private.
2. Provide public get and set methods to access and update the values of a private variables.

[Note: get method returns value, set method set the values.]

**Inheritance:**

**Polymorphism:** ability to **perform** **single action** in **many ways.** It occurs when we have many classes that are related to each other by inheritance.

It lets us inherit attributes and methods from another class. Polymorphism uses those methods to perform different task.

Why: concept is extensively used in implementing inheritance. It plays an important role in allowing objects having different internal structures to share the same external interface.

The benefit of **overriding:** ability to define a behavior that’s specific to the subclass type, which means a subclass can implement a parent class method based on its requirement.

In OOPS. Overriding means to override the functionality of an existing method.

**Overloading:** is feature that allows a class to have more than one method having same name, if their arguments lists (signature) are different with in the same class.

[**Compile time, Static polymorphism. Static/Early binding**].

**Overriding:** is a feature that allows a subclass or child class to provide specific implementation of a method that is already provided by one of its super classes or parent classes.

(**Run-time, Dynamic polymorphism, Dynamic/Late binding**).

[Note: **child class should extend the parent class**]

**Abstraction:**

**Interfaces in Java**

Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body). Interfaces specify what a class **must do** and **not how**. It is the blueprint of the class.

An Interface specifies a set of methods that the class has to implement.

If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.

A Java library example is, Comparator Interface. If a class implements this interface, then it can be used to sort a collection.

To declare an interface, use interface keyword. It is used to provide total abstraction. That means all the methods in an interface are declared with an empty body and are public and all fields are public, static and final by default.

A class that implement interface must implement all the methods declared in the interface. To implement interface use implements keyword.

**Why do we use interface ?**

It is used to achieve total abstraction.

Since java does not support multiple inheritance in case of class, but by using interface it can achieve multiple inheritance.

Interfaces are used to implement abstraction. So the question arises why use interfaces when we have abstract classes?

The reason is, abstract classes may contain non-final variables, whereas variables in interface are final, public and static.

**Abstract classes:**

Abstract classes cannot be instantiated and are designed to be sub-classed. They are used to provide some common functionality across a set of related classes while also allowing default method implementations.

Rules:

* Abstract classes cannot be instantiated.
* If a class has at least one abstract method, then the class must be declared abstract.
* To use an abstract class, we must create a class that extends the abstract class(inheritance) and provide implementations for all abstract methods.
* Java does not support multiple inheritance, so we are only to extends on class (abstract or not). There is where interfaces become useful.

**Is-a and Has-a relationship:**

**Is-a relationship**: where an **object** can be **considered as a type of another**. **If a total functionality is required**. (means class B **extends** class A)

**Has-a relationship**: where an object contains variable of another object. **Commonly used functionality**. (means create an object with **new keyword**, write once and use anywhere. Code reusability.

**Association:** those relationships whose **object have an independent lifecycle** and where there is **no ownership between objects**.

**Aggregation:** those relationships whose **object have an independent lifecycle** and where there is **ownership**, and **child objects cannot belong to another parent object**.

**Composition:** those relationships whose **objects don’t have an independent lifecycle** and if the **parent object is deleted, all child objects will also be deleted**.

**Type casting:** assigning a value of one primitive data type to another primitive data type.

1. **Implicit/ upcasting/Widening** (automatically): smaller to larger data type. This is done by compiler.
2. **Explicit/downcasting/Narrowing** (manually: larger to smaller data type. Programmers will do it.

**Wrapper class**: is a class whose object wraps or contains a primitive data type. When we create an object to a wrapper class, it contains a filed and in this filed we can store a primitive data type.

**Use**: wrapper classes are used to convert any **primitive data type** into an **object**. The primitive data types are not objects. They do not belong to any class. A data type is to be converted into an object and then added to a Stack or Vector etc.

**Auto boxing**: Converting a **primitive data type value** into a corresponding **wrapper class**. Java compiler applies autoboxing when a **primitive value** is:

* passed as a parameter to a method that expects an **object** for the corresponding wrapper class.
* assigning to a variable of the corresponding **wrapper class**.

**Unboxing**: Converting an **object of wrapper type** to its corresponding **primitive data type value**. Java compiler applies unboxing when an **object of a wrapper class** is:

* passed as a parameter to a method that expects a **value** of corresponding primitive type.
* assigning to a variable of the corresponding **primitive data type**.

[ Note: Java complier applies when an object of a wrapper class]

**Java packages & API:** group of related classes.

1. User-defined packages.
2. Built-in packages

[Note: Library is divided into packages and classes].

**Arguments:** a value passed to a method where **method is called.**

**Parameters:** a variable used to define a particular value during a **method definition.**

**Enum:**

An **enum** is a special class that represents a group of constants (unchangeable variable, like final variable).

Can be accessed with **dot** syntax. ( enum type can be accessed by values() method in for loop).

Enum can be used inside a class.

Can be used in switch statement.

**Why can’t we override static methods?**

No, static means common for all the objects in the hierarchy. When we want to override then there is no point to declare the method as a static in the parent class.

**Why we cannot override constructor?** [ constructor in java is special method in java]

No, the constructor name of the child class should be same as the child class name then, how can you call the parent class constructor. Constructor is a special method belongs to the specific class only.

static: // memory management.

* Static variables: declared inside class and out a method
* Static methods: method is declared static
* Static blocks: using static keyword.

**Static variable, methods and static block are executed during .class file loading.**

When we execute a java class file (java Classname)

* Loads corresponding .class byte code into memory.
* After loading, it calls main method for execution.

[note: .class is loaded only once].

**Instance block depends on the object creation.**

JVM memory areas:

* Method Area: (classes byte codes are loaded)
* Heap Area: (class objects are stored, Heap memory is stored in RAM)
* Stack Area: (whenever a thread is created a frame is created here)
* PC registers:
* Native method:

Local variables:

* Declared inside method, constructor and blocks.
* Scope of the local variable specific to method, constructor and blocks.
* Memory is allocated only when the method is invoked, the memory is released with is completed.

Instance Variables:

* Declared inside the side class and outside the methods.
* Scope of the instance variable is inside the class all methods, constructors and blocks.
* Memory is allocated when the object is created, and memory destroyed when is destroyed.
* Stored in heap memory.
* Can be accessed directly if same area (instance area), if other area (static) using the object.

Static Variables:

* Declared inside the class and outside the methods with static modifier.
* Scope of the instance variable is inside the class all methods, constructors and blocks.
* Memory is allocated when the .class is loading and destroyed .class file
* Stored in non- heap memory.
* Can be assessed by using classname.

**Areas of java**:

* Instance area:
* Static area:

**Heap memory**: objects occupy memory.

**Stack memory**: references pointing to object in heap memory.

**static and non-static variables (also called as instance variables)**:

Variable is named memory location.

Anything which is common, then we have to go for static variable, free access, no restrictions.

**Static belongs to a class**.

They are accessed using Classname.variableName outside the class.

Every method of a class can access a static variable including Constructor.

**this()**: this() keyword is used to call constructor.

In constructor overloading, if we want to call same class constructor from another constructor.

**this**: this keyword used to call the variable and methods in the same class.

**super()**: super() keyword is used to call super class constructor.

Is mainly used to call parent class constructor.

**super**: super keyword is used to call variable and methods from the super class.

**Constructor:**

* Constructor is simply a method that can be called from different parts of the program to create instance of a defined class.
* Used to create a new object from the class each time it is called.
* Constructor has no explicit return type because it’s a factory method, which means it creates a new instance of the class and returns the new instance to the caller.
* Constructor of a class must have same name of the class and is case sensitive.
* Every object created from a class is referred to as an instance of the class.
* Finally, the constructor can accept information from the caller as paraeters that will be assigned to the instance fields of the instance being created.

**Serialization (Object state):**

The concept of **converting object state** into **persistent state** (**physical state**) (to write to a file).

**Three ways to read data from console: (BufferedReader, Scanner, Console objects):**

BufferedReader br=new BufferedReader(new InputStreamReader(System.*in*));  
String s=br.readLine();

Scanner src=new Scanner(System.*in*);  
String s1=src.next();

Drawbacks of the above dynamic inputs:

We have to use two instructions to get data.

Do not provide security. (password, pin etc.)

Console cs=System.*console*();  
char[] p = cs.readPassword("enter password :");  
String up=new String(p);

**Ways to create an object:**

new keyword

A o=new A();

instance factory method

static factory method

Pattern factory method

new instance method

Classc=Class.forName("com.company.A");  
Object o=c.newInstance();

**Difference between this and super in Java:**

***this*** and ***super*** are reserved [keywords](https://howtodoinjava.com/java-keywords/) in Java. this refer to current instance of a class while super refer to the parent class of that class where super keyword is used.

**Java this keyword:**

* this keyword automatically holds the reference to current instance of a class.
* It is very useful in scenarios where we are inheriting a method from parent class into child class and want to invoke method from child class specifically.
* We can use this keyword to access [static](https://howtodoinjava.com/java/basics/java-static-keyword/) fields in the class as well, but recommended approach to access static fields using class reference.

**Java super keyword:**

* Similar to this keyword, super also is a reserved keyword in Java. It always holds the reference to parent class of any given class.
* Using super keyword, we can access the fields and methods of parent class in any child class.

**Java Pass-by-Value vs. Pass-by-Reference:**

Java is pass by value and not pass by reference.

When you pass an instance to a method, its memory address are copied bit by bit to new reference variable.

Thus both pointing to same instance.

But if we change the reference inside method, original reference will not get changed.

if it was pass by reference, then it would got changes.

**Java View/Generate Bytecode of Class file:**

> javap -c <pathtoclassfile> > <pathto destiantion\sampl.bc>

**return** keyword:

The return keyword is used to return from a method when its execution is complete. When a return statement is reached in a method, the program returns to the code that invoked it.

A method can return a value or reference type or does not return a value. If a method does not return a value, the method must be declared void and it doesn’t need to contain a return statement.

If a method declared to return a value, then it must use the return statement within the body of method. The data type of the return value must match the method’s declared return type.

**return** keyword in the method:

void keyword, denotes that the method does not return any value after the method has been called and executed. (***the method have nothing to offer the JVM in memory***).

**return this** keyword, is used to refer to an object of a class or objects fields.

public Dog getDog() {

// return Dog type

return this;

}