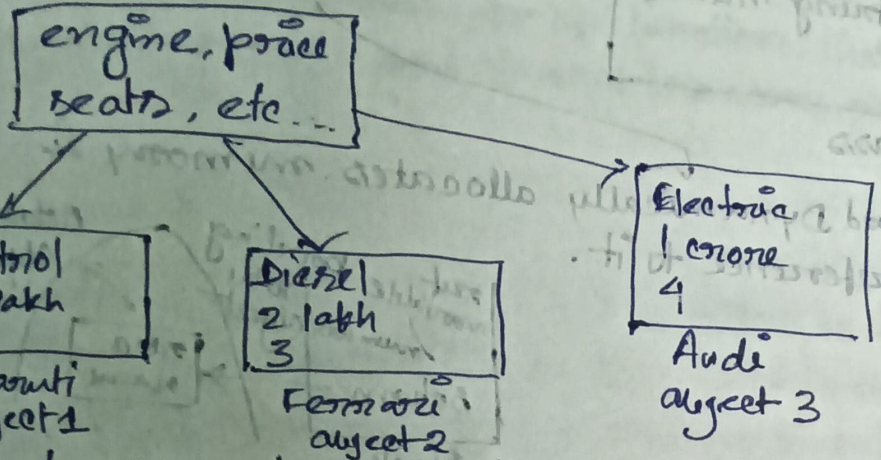
Object Oriented Programming (OOPS)

Class



- A class is a template for an object, and an object is an instance of a class.

→ Simple way class is a collection of properties and functions or methods.

When you declare an object of a class, you are creating an instance of that class.

→ Class is a logical construct. An object has physical reality. (That is, an object occupies space in memory.

Objects are categorised by three essential properties state, identity and behaviour.

State – The state of an object is a value from its data type.

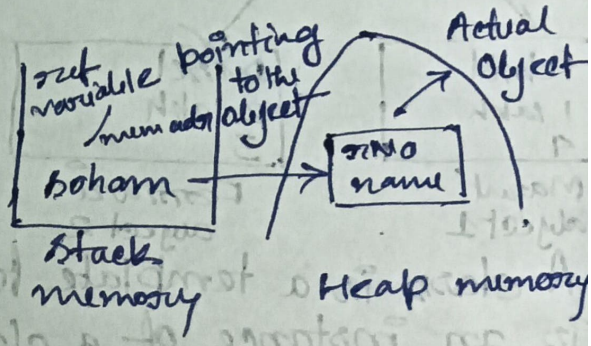
Identity – The identity of an object distinguishes one object from another.

Behaviour – The effect of data-type operations.

```
Class Student {
    int rNo;
    String name;
}
// Student Class
```

```
// Declaring a ref variable
Student soham;
// Allocating a Student Object;
soham = new Student();
```

The new keyword Dynamically allocates memory & return a reference to it.




```
// Declaring reference to object
Box mybox;
// Allocates a Box Object
mybox = new Box();
```

The first line declares mybox as a reference to an object of type Box, at this point, mybox doesn't yet refer to an actual object. The next line allocates an object and assigns a reference to it to mybox. After the second line executes, you can use mybox as if it were a Box object. But in reality, mybox simply holds the memory address of the actual object.

```
Box b1 = new Box();
Box b2 = b1;
```

b1 and b2 will both refer to the same object. It simply makes b2 refer to the same object as does b1. Thus any changes made to the object through b2 will affect to which b1 is referring, since they are the same object. When you assign one object reference variable to another object reference variable, you are not creating a copy of the object, you are making a copy of the reference.

Note

```
Bus bus = new Bus();
```

Compiler — JVM

Student soham = new Student ();

Constructor

Constructor is a special type of function, that runs when an object is created and it allocates some variables.

This Keyword

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the this keyword. "this" can be used inside any method to refer the current object. That is, this is always a reference to the object on which the method was invoked.

* This can also be used to call a constructor from another constructor

```
class Student (){
    int num;
    String name;
    Student (){
        this (0, "Default Student");
    }
    Student (int num, String name){
        this.num = num;
        this.name = name;
    }
```

A parameter is a variable defined by a method that recieves a value when the method is called. Ex. int square(int i), i is the Parameter. An argument is a value that is passed to a method when it is invoked. For example, square(100) passes 100 as an argument. Inside square(), the parameter i recieves the value.

A field can be declared as final. Doing so prevents its contents from being modified, making it, essentially, a constant. This means that you must initialize a final field when it is declared.

It is a common coding convention to choose all uppercase identifiers for final fields.

```
final int INCREASE = 2;
```

Unfortunately, final guarantees immutability only when instance variable are primitive types. If an instance variable of a reference type has the final modifier, the value of instance variable (the reference to an object) will never change. It will always refer to the same object but the value of the object itself can change.

```
final Student soham = new Student();
✓ soham.name = "Ramkrishna";
  // ↑ this is possible
✗ soham = new Student("Ramkrishna");
  // this is not possible
}
```

Finalize() Method

Sometimes an object will need to perform some action when it is destroyed. To handle such situation, Java provides a mechanism called finalization. By using finalization, you can define specific actions that will occur when an object is just about to reclaimed by the garbage collector. To add a finalizer to a class, we simply define the finalize() method. The Java run time calls that method whenever it is about to recycle an object of that class. Right before an asset is freed, the Java run time calls the finalize() method on the object.

Note in c++ Finalize() is called as Destructor.

Constructors :- Once defined, the constructor is automatically called when the object is created, before the new operator completes. Constructors look a little strange because they have no return type, not even void.

This is because the implicit return type of a constructor is the class type itself.

In the line

Box mybox1 = new Box();

new Box() is calling the Box() constructor.

* In Java, constructor of base class with no arguments gets automatically called in derived class constructor.

* Any class will have a default constructor, doesn't matter if we declare it in the class or not. If we inherit a class then the derived class must call its super class constructor. It is done by default in derived class. If it doesn't have a default constructor in derived class, the JVM will invoke its default constructor and call the super class constructor by default. In this case, if the super class doesn't have a default constructor, instead it has a parameterized constructor, then the derived class constructor should call explicitely the parameterized super class constructor

To inherit a class, we simply incorporate the definitio of one class into another class by using the "extends" keyword.

* We can only specify one super class for any subclass. Java doesn't support the inheritance of multiple superclasses into a single subclass.

* A Superclass Variable can Reference a Subclass Object?

: It is the type of the ref variable - not the type of the object that it refers to that determines what members can be accessed.

When a reference to a subclass object is assigned to a superclass ref variable, you will have access only to those parts of the object defined by the superclass.

```
Superclass ref = new Subclass();
// ref can only access methods which are available
   in Superclass.
```

Super Keyword

Whenever a subclass needs to refer to its immediate superclass, it can do by use of the keyword super. Super has two general forms.

(1) Calls the Super class Constructor

(2) Access the members of a superclass that has been hidden by a member of a subclass.

A Superclass has no knowledge of any subclass, any initialization it needs to perform by the subclass it must completed its execution first.

Note

If Super() is not used in subclass' constructor, then the default or parameterless constructor of each superclass will be executed.

Using 'final' With Inheritance

1) Prevent Overriding

To disallow a method from being overridden, specify final as a modifier at the start of its declaration. Methods declared as final cannot be overridden.

***
Methods declared as final sometimes provide a performance enhancement:- The compiler is free to inline calls to them because it "knows" they will not be overridden by a subclass. When a small final method is called, often the Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call. Inlining is an option only with final methods.

Normally Java resolves calls to methods dynamically, at run time. This is called late binding. However, since final methods cannot be overridden, a call to one can be resolved at compile time. This is called early binding.

2) To Prevent Inheritance

Sometimes we will want to prevent a class from being inherited. To do this, precede the class declaration with final.

Note: Declaring a class as final implicitly declares all of its methods as final, too.

As you might expect, it is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself & relies upon its sub-classes to provide complete implementation.

→ abstract final class Box {

}

// Not possible

Understanding Static keyword

When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object. We can define both methods and variables to be static.

The most common example of a static member is main()

→ public static void main().

It is declared as static because it must be called before any object exist. Static method in Java is a method which belongs to the class and not to the object.

A static method can access only static data. It can't access non-static data. A non static member belongs to an instance. It's meaningless without somehow resolving which instance of a class you are talking about. In a static context, we don't have an instance that's why we can't access a non-static member without explicitly mentioning an object reference. In-fact we can access a non-static member in a static context by specifying the object reference explicitly.

* A static method can call only other static methods and cannot call a non-static method from it.
* A static can be accessed directly by the class name and doesn't need any object.
* A static method cannot refer to 'this' or 'super' keyword in anyway.

Only nested classes can be static.

Static inner classes can have static variables.

We can't override the inherited static methods, as in Java overridding takes place by resolving the object type at run-time, Static methods are Class level methods, so it is always resolved during compile time.

Note A Static block runs once when the class is loaded.

Why outer classes can't be static ?

Ans. static class Test {

}

Here static not possible because 'Test' class is not dependent on another class.

class Static {

static class Test {

We have to define static here otherwise compiler

String name;

Test (String name) {

this.name = name;

}

public static void main (String[] args) {

Test a = new Test ("Soham");

Test b = new Test ("Ramkrushna");

System.out.println(a.name);

}

}

Because :- 'Test' class is dependent on the 'Static' class, so without static not available to use the 'Test' class we have to define an instance of 'Static' class.

are used to keep the class name compartmentalized. For example a package allows us to create a class named list. We can use same class Name in two different package.

To use one class of a package from a different package we have to import it by using the import keyword.

Package is both a naming and visibility control mechanism.

Method Overriding :-

In a class hierarchy, when a method in a subclass has the name and type signatures as a method in the Superclass, then the method in the subclass is said to override the method in the superclass. When an overriden method is called from within its subclasses it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

* Method overriding occurs only when the names and the type signatures of two methods are identical. If they are not, then the two methods are simply overloaded.

Dynamic Method Dispatch :- It is the mechanism by which a call to an overridden method is ~~overloaded~~ resolved at run time. Dynamic method dispatch is important because this how java implements run-time polymorphism.

A Superclass ref variable can refer to a subclass object. When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being ~~called~~ referred at the time the call occurs. Thus the determination is made at run time.

In simple words, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.

*** Superclass ref = new Subclass();

ref can access all methods or variables, if I want to access one method (same name, same argument, same return type) then the method which the Subclass means the object type will be called.

So Method overriding is a Run-time Polymorphism

Polymorphism – Poly means many morphism means way to represent

Method Overloading :–

In Java, it is possible to define two or more methods within the same class that share the same name, as long as their parameter declaration are different.

While Overloaded methods may have different return types, the return type alone is insufficient to distinguish two version of method. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameter match the arguments used in the call.

In some cases, Java automatic type conversion can play role in overload execution.

Abstract

Sometimes we will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the method that the subclass must implement.

We may have methods that must be overriden by the subclass in order for the subclass to have any meaning. In this case we want some way to ensure that a subclass does, indeed, override all necessary methods. Java's solution to this problem is the abstract method.

* Any class that contains one or more abstract method must also be declared as abstract.
* There can be no object of an abstract class ★★★
* We can't declare abstract constructors or abstract static methods.
* We can declare static method in a abstract class.

Because there can be no objects for abstract class. If they had allowed to call abstract static methods it would that mean we are calling an empty method (abstract) through the classname because it is static.

Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be declared abstract itself.

Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java's approach to run time polymorphism is implemented through the use of superclass references.

Interface

Interface is like class but not completely. It is like an abstract class. By default functions are public and abstract in interface. Variables are final and static by default in interface.

Interfaces specify ~~between classes~~ only what the class is doing, not how it is doing that.

The problem with multiple interface is that two classes may define different ways of doing the same thing, and the subclass can't choose which one to pick.

Key Difference between Classes and Interfaces: a class can maintain state information through the use of instance variable, but interface cannot.

Using interface we can specify a set of methods that can be implemented by one or more classes. Although they are similar to abstract classes, interfaces have an additional capability: A class can implement more than one interface. By contrast, a class only inherit a single superclass.

Nested interface:- An interface can be declared a member of a class or another interface. Such an interface is called a member interface or a nested interface. A nested interface can be declared as public, private or protected. This differs from a top-level interface, which must either be declared as public or use the default access level.

Interface Can be Extended: One interface can inherit another by use of the keyword extends. The syntax is the same as for inheriting classes. Any class that implements an interface must implement all methods required by that interface, including any that are inherited from another interfaces.

Default Interface Methods

A primary motivation for the default method was to provide a means by which interfaces could be expanded without breaking existing code. Suppose you add another method without body in an interface. Then they have to provide the body of that method in all classes that implement that interface.

Eg
```
default String getString(){
    return "Default String";
}
```

For example, you might have a class that implements two interfaces. In each of these interfaces provides default methods, then same behaviour is inherited from both.

* In all cases, a class implementation takes priority over an interface default implementation
* In case in which a class implements two interfaces that have the same default method, but the class doesn't override that method, then an error will result.
* In cases in which one interface inherits another, will both defining a common default method, the inheriting interfaces version will takes precedence.

Static interface methods are not inherited by either an implementing class or a sub-interface. Static interface methods should have a body! They cannot be abstract.

Note: When overriding methods, the access modifier should be same or better. i.e. if an parent class it was protected, then overriden should be either protected or public.

Abstract Class vs Interface

Feature	Abstract class	Interface
Types of Methods	Abstract classes can have abstract and not abstract methods. In Java 8, it can have default and static method also	Can have only abstract method and default method
Variables	Variable can be final and non final also	Variables can be final.
Implementation	Abstract class can provide the implementation of interface. Abstract class can be extended using keyword 'extended'.	Interface can't provide the implementation of abstract classes. A Java interface can be implemented using keyword 'implements'
Multiple implementation	An interface can extend another Java interface only.	An abstract classes can extend another Java class and implement multiple Java interfaces
Accessibility of Data members	An abstract class can have class members like private, protected etc.	Members of Java interfaces are public by default

Abstraction Vs Encapsulation

Feature	Abstraction	Encapsulation
	Abstraction is a feature of OOPs that hides unnecessary details but shows the essential information.	Encapsulation is also a features of OOPs. It hides the code and data into a single entity or unit the data can be protected from outs world.
	It solves an issue at the design level.	It solves an issu at implementation level.
	Focuses on external lookout	Focuses on interna working
	It can be implemented using abstract classes and interfaces.	It can be imple by using the acce modifiers (priv public, protecte
	It is the process of gaining information	It is the process of containing the information
	In abstraction we use abstract classes and interfaces to hide the code complentities The object are encapsulated that helps to perform encapsulation	We use getter setter method to hide the data. The object need not to abstract that result in encapsulation

Access Control

How a member can be accessed is determined by the access modifier attached to its declaration.

Java access modifiers are public, private, and protected. Java also defines a default access level protected only applies when inheritance is involved. When no access modifier is used, then by default the member of a class is public within its own package but cannot be accessed outside of its package.

	Class	Package	Subclass (Same Package)	Subclass (Different Packages)	World
public	+	+	+	+	+
Protected	+	+	+	+	
no modifier	+	+	+		
private	+				

Protected allows access from subclasses and from other classes in the samepackage.
We can use child class to use protected member outside the package but only child class object can access it.
Access to a protected methods of that instance is only allowed from objects of the same package.

Exception handling

Exception handling is a mechanism that allows a program to detect and handle errors gracefully - instead of crashing unexpectedly.

An exception is an unexpected event or error that occurs during the execution of a program and disrupts its normal flow.

Keyword Used in Exception Handling

Keyword	Description
try	Block of code where an exception may occur
catch	Block that handles the exception
finally	Block that always executes (used for clean-up)
throw	Used to manually throw an exception
throws	Declares exceptions that a method might throw.

Types of Exception

Type	Description	Example
Checked	Checked at compile time	IOException, SQL-Exception
Unchecked	Occur at runtime	ArithmaticException NullPointerException
Errors	Serious system issue	OutOfMemoryError StackOverflowError

Benefit Of Exception Handling

1) Prevent program crashes
2) Makes code more readable and robust
3) Separate error handling code from normal code
4) Allows recovery from unexpected errors.

Exception Handling — Detecting, Catching + Handling errors without stoping the program.

Wrapper Classes

In Object-Oriented Programming (OOP), a wrapper class is a class that "wraps" a primitive data type into an object, so it can be used where objects are required.

Example — Integer, Character, Float, Boolean

Example Code

```
int a = 10;
Integer obj = Integer.valueOf(a);   // wrapping
int b = obj.intValue();             // unwrapping
sout(obj);   //10
sout(b);     //10
```

* It also support null values.

Difference between is a relationship and can-do relationship?

Relationship	Keyword/Mechanism	Example	Meaning
is-a	extends (Inheritance)	Dog extends animal	Dog is a type of Animal
can-do	implements (Interface)	Fish implements Swimable	Fish can swim