Object Oriented Programming

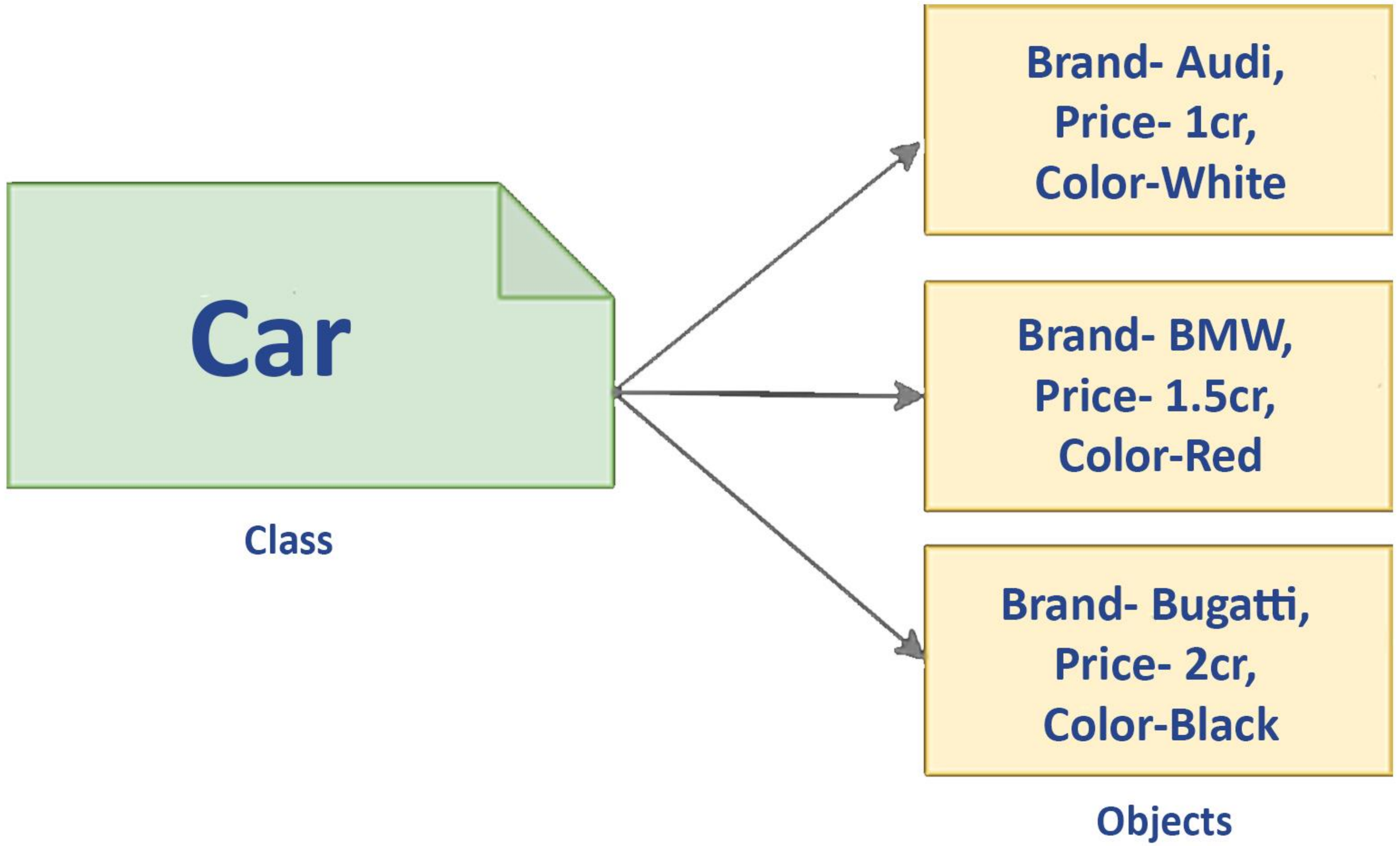
Nikhita Palla

What is Object Oriented Programming?

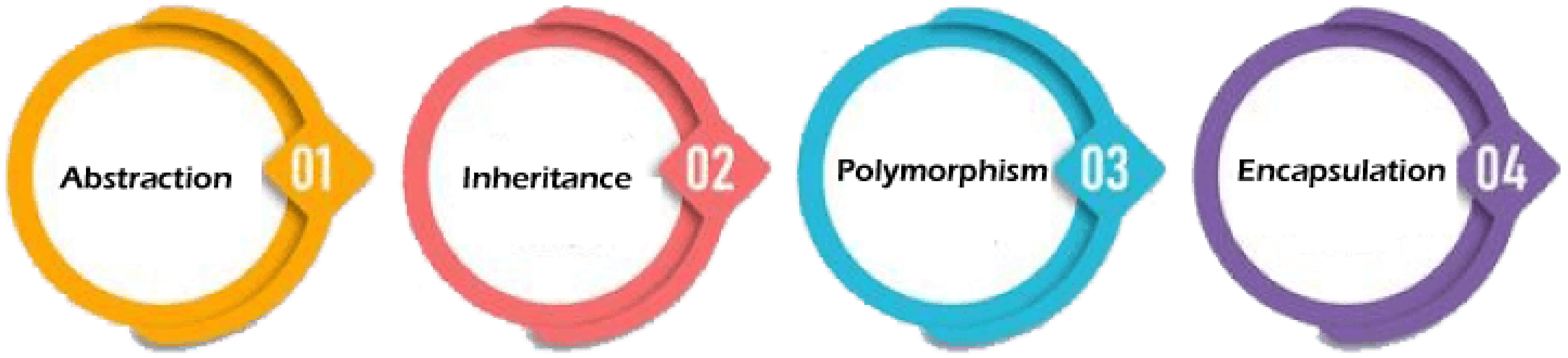
- Programming paradigm.
- Relies on the concept of **classes** and **objects**
- Structure a software program into simple, reusable pieces of code blueprints which are used to create individual instances of objects.

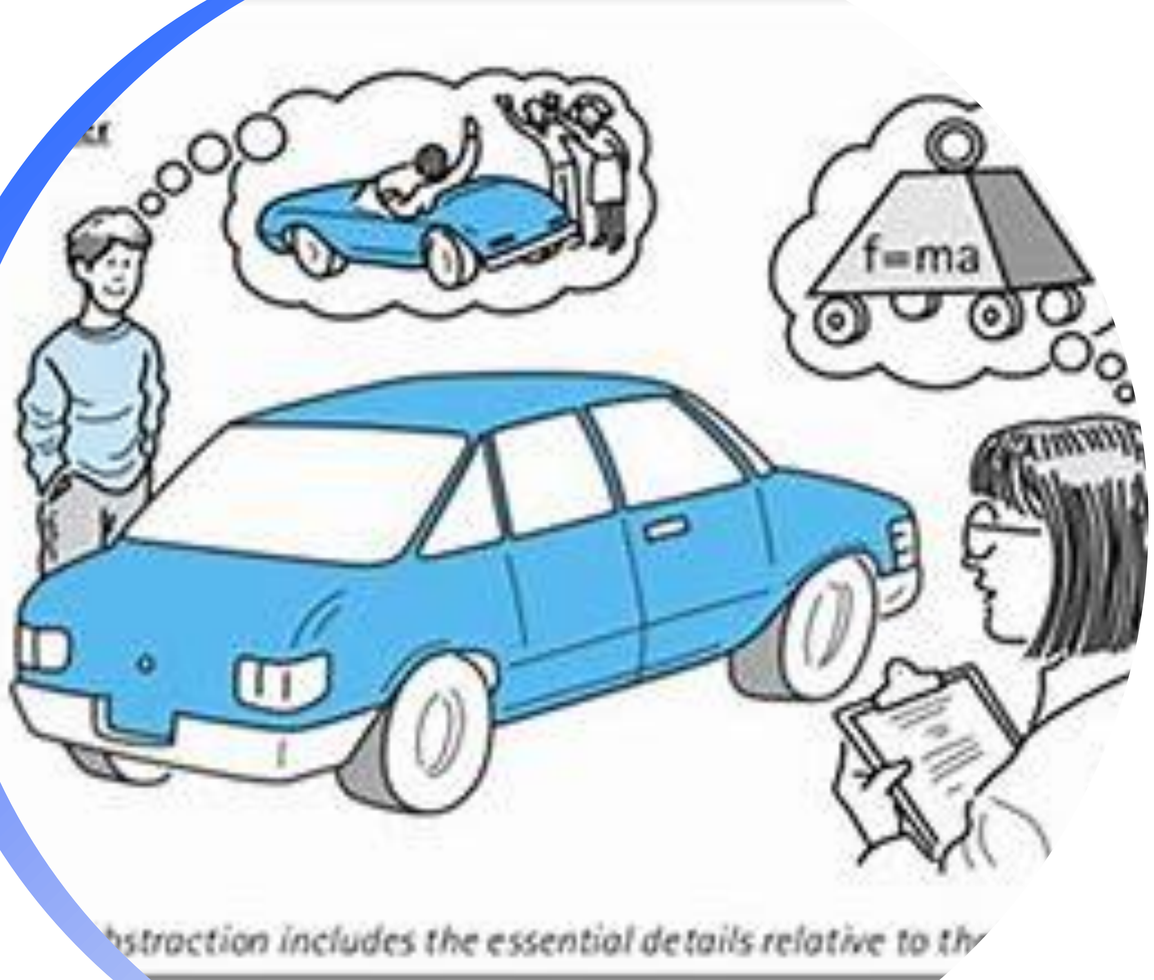
Building blocks

- Class : blueprint for the structure of methods and attributes.
- Object : Instance of Class
- Attributes : defined in class template
- Methods : represent behaviour


Car
Class
Brand- Audi,
Price- 1cr,
Color-White
Brand- BMW,
Price- 1.5cr,
Color-Red
Brand- Bugatti,
Price- 2cr,
Color-Black
Objects

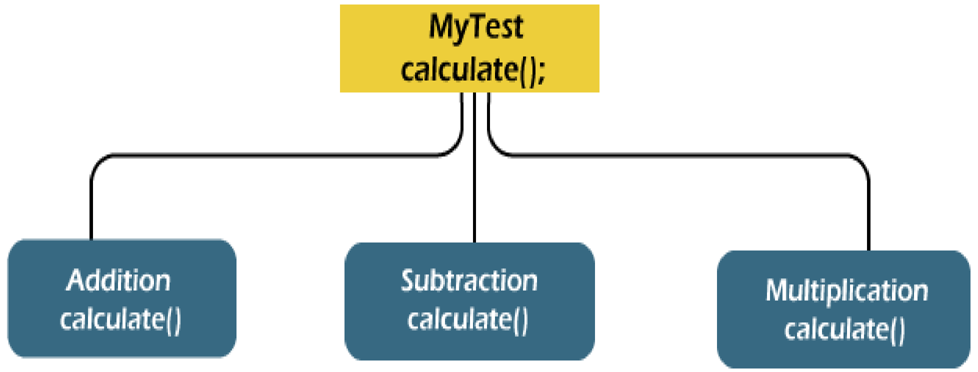
Pillars of OOPs





Abstraction

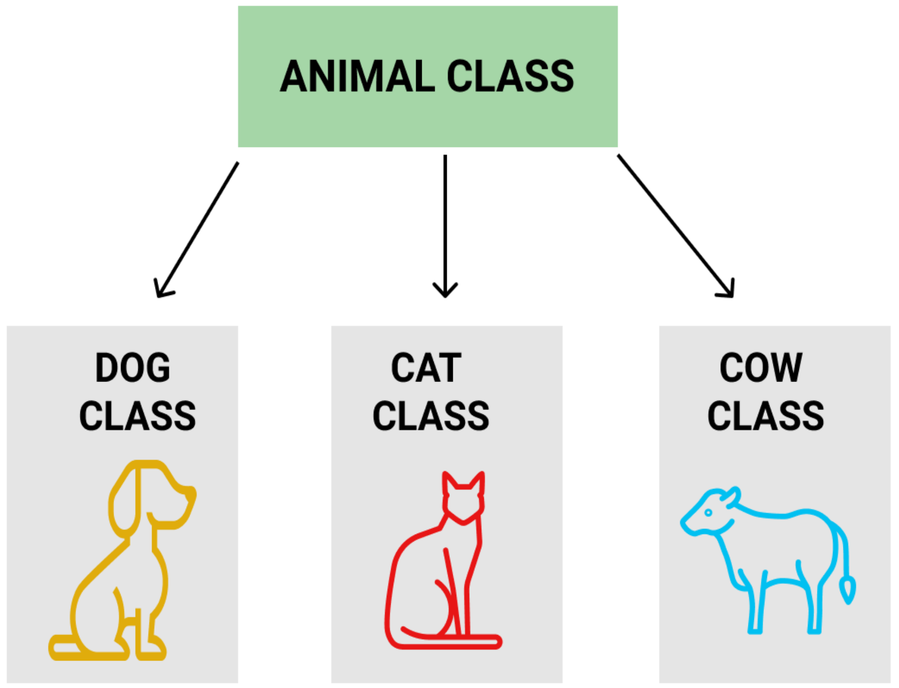
Hide the implementation from the user.

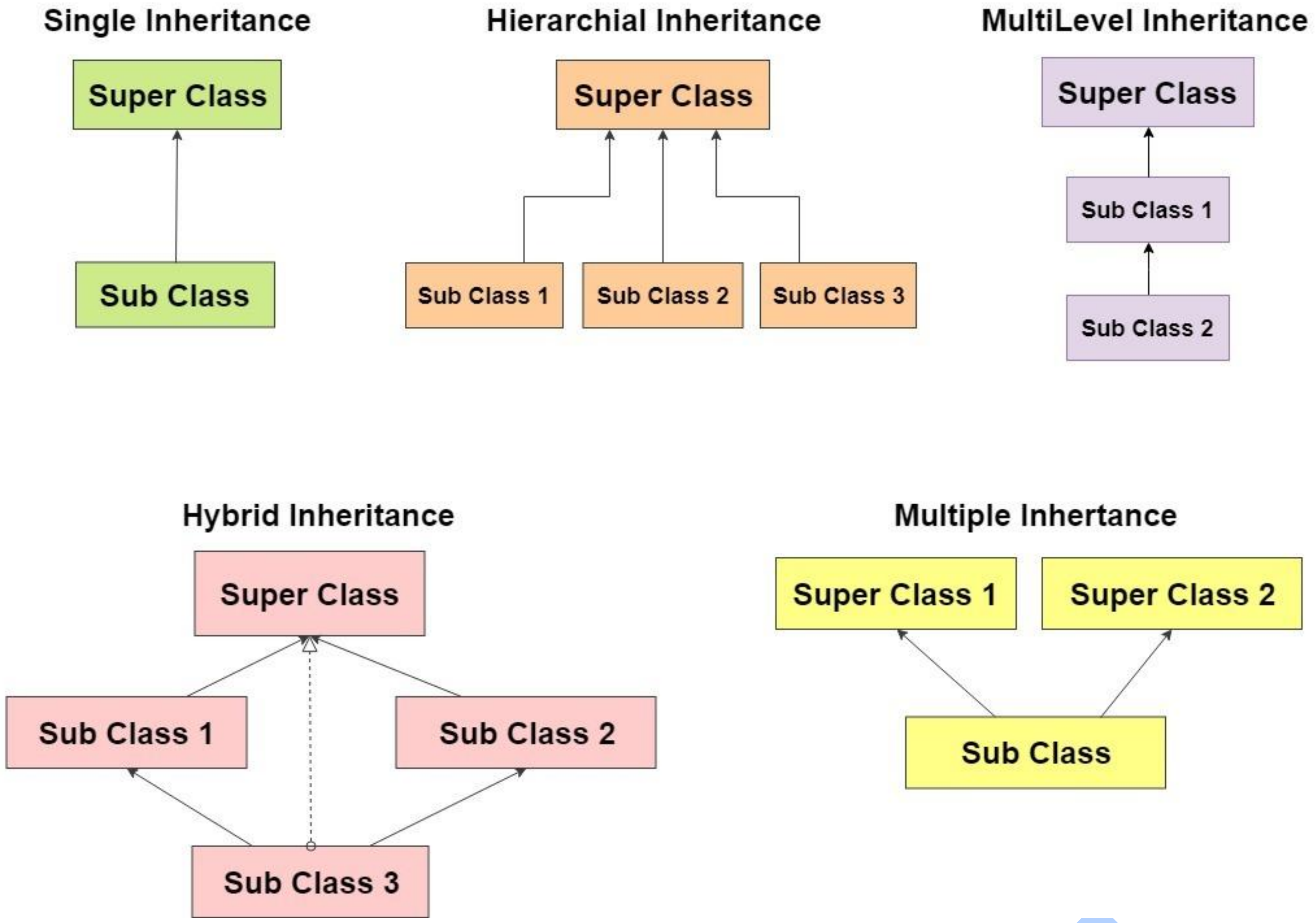


Inheritance

Inherit or acquire the properties of an existing class (parent class) into a newly created class (child class).

Provides code reusability.



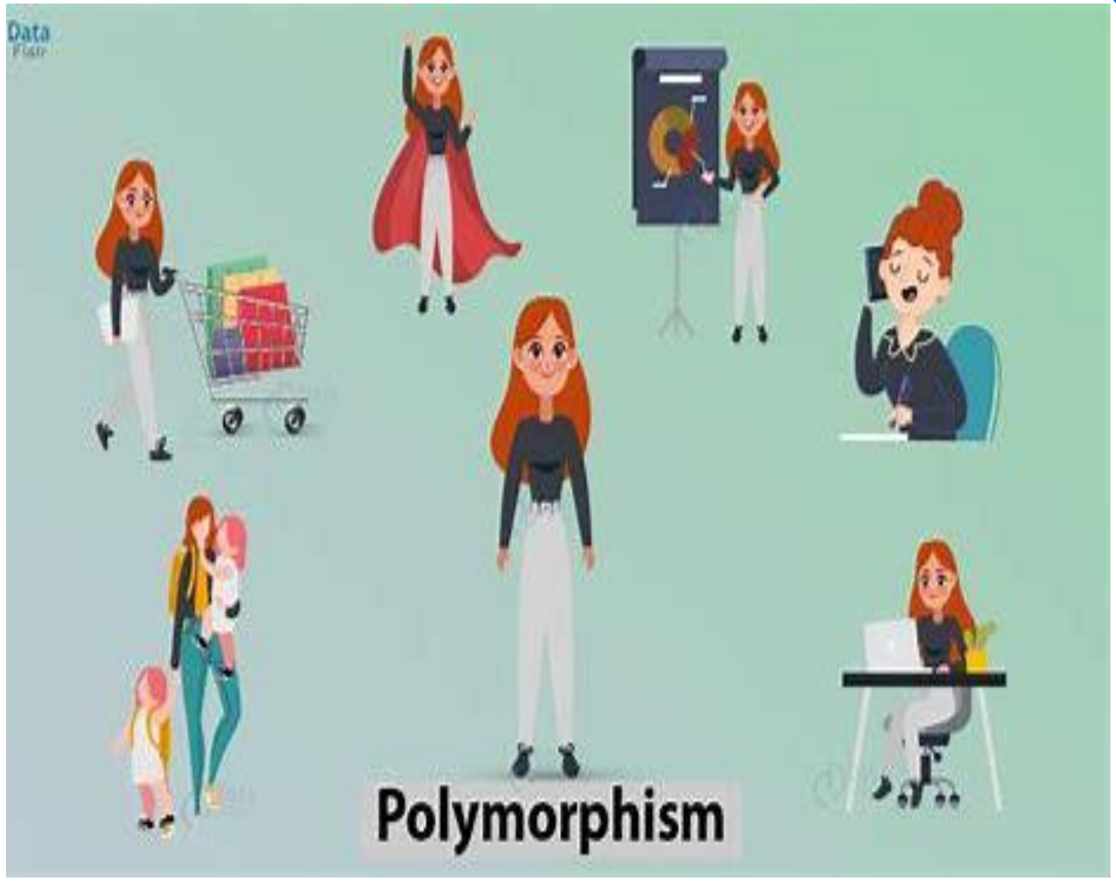

Single Inheritance
Super Class
Sub Class
Hierarchial Inheritance
Super Class
Sub Class 1
Sub Class 2
Sub Class 3
MultiLevel Inheritance
Super Class
Sub Class 1
Sub Class 2
Hybrid Inheritance
Super Class
Sub Class 1
Sub Class 2
Sub Class 3
Multiple Inhertance
Super Class 1
Super Class 2
Sub Class

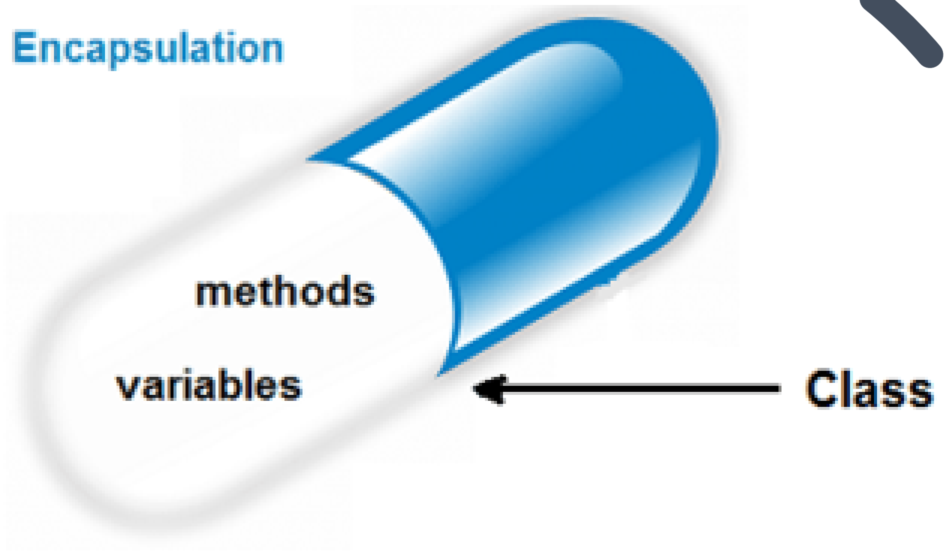
Polymorphism

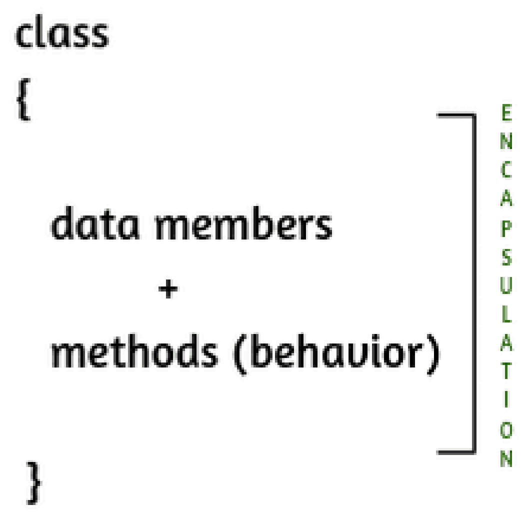
Poly means many and morphs means forms.

One action -> multiple implementations

Allows us to create methods with the same name but different method signatures.







Encapsulation

Bind data and functions of a class into an entity.

Protects data and functions from outside interference and misuse.

A class variables are hidden from other classes and used by the methods of that class itself.

Benefits of OOPS

- Modular, scalable, extensible, reusable, and maintainable.
- Object can be used across the program.
- Code can be reused.
- Provides security through encapsulation and data hiding features.
- Beneficial to collaborative development in which a large project is divided into groups.
- Debugging is easy.

Programming Paradigms

- Paradigm: method to solve a program
- Programming paradigm is a method to solve a problem using tools and techniques that are available to us following some approach.
- Types

  Imperative programming paradigm

  Declarative Programming paradigm

Imperative programming paradigm

```
    int sum = 0;
    sum += 1;
    sum += 2;
    sum += 3;
    sum += 4;
    sum += 5;
    sum += 6;
    sum += 7;
    sum += 8;
    sum += 9;
    sum += 10;

    printf("The sum is: 
%d\n", sum); //prints->
The sum is 55

    return 0;
}
```

Types of Imperative Programming paradigm

Proceedural

Object Oriented

Parallel Processing

Proceedural Programming Paradigm

Proceedures are not functions

They donot return any value

Primary purpose is to complete a given task

Example: for loop

Object Oriented programming paradigm

Reusability through inheritance

Flexibility through polymorphism

High security with the use of data hiding (Encapsulation) and Abstraction mechanisms.

Faster development with high quality

Parallel Processing Approach

Allows many processors to run a program in less time by dividing them up.

Structured Programming Paradigm

Make programs easier to understand through the use of control structures, subroutines, and blocks.

Sequence: instructions must follow a strict order from one instruction to the next. This forbids the use of goto's and jumps that can arbitrarily set the execution point.

Selection: certain instructions are only executed if a condition is met, typically seen with if-else statements.

Iteration: executing a set of instructions repetitively until a condition is met. This usually takes the form of for and while loops.

Declarative programming paradigm

Programmer defines what needs to be accomplished by the program without defining how it needs to be implemented.

Ex: Relational Databases

Aspect Oriented programming paradigm

modularize cross-cutting concerns

Cross-cutting concerns: affect multiple parts of a software application.

In OOP, it is typically encapsulated within classes, leading to a scattered and tangled code when dealing with cross-cutting concerns.

address this issue by separating these concerns from the main business logic.

Key Concepts:
Aspect: module that encapsulates a cross-cutting concern
Join point: where aspects can be applied.(exception handling,method invocation, object instantiation.)
Advice: action or code to be executed at join point.
Pointcut: set of join points.defines the criteria for where an aspect should be applied in the program.
Weaving: Integrating aspects into main program
Introduction: enables aspects to add new functionality

Logical Programming paradigm

Relationships

1) Inheritance(is-a)
2) Association(has-a)

    Aggregation

    Composition

Inheritance

```
class Vehicle{
}
Class Car extends Vehicle{
}
```

Car **is a** Vehicle

Association

```
class Engine{
}
class Car{
   Engine e=new Engine();
}
```

Car **has a** Engine

Aggregation

- Weak bonding
- Car has a music player

Composition

- Strong Bonding
- Car has a Engine

Top Down Approach	Bottom Up Approach
General to specific	Specific to general
Prioritized high level planning and decission making	Prioritizes the execution of individual tasks and the development of detailed knowledge.
A problem is divided into parts and the parts are then solved by dividing into subparts	Individual parts of a problem are specified in detail and and these parts are linked to form a solution for larger problem
Structured programming	Object oriented programming
Have redundancy as each part is solved separately	Redundancy is minimized by encapsulationa and abstraction
Debugging	Testing
Decomposition	composition
Identifing the overview of problem is difficultt	Sometimes cant build a program from the piece we started

Fully Encapsulated Class

```
Class Employee{
    private int id;
    private String name;
    public getname(){  return name; }  //getter
    public setname(String name){ this.name=name; }  //setter
    public getid(){ return id;}  //getter
    public setid(int id){ this.id=id;}  //setter
}
```

Constructors

Special method to initialize objects.

Called when object of class is created and memory is allocated to the object.

When object is created with new() method, atleast one constructor is called.

Have the same name as class

Donot need to have return type like the methods.

They are called only once at object creation.

Types of constructors - Default

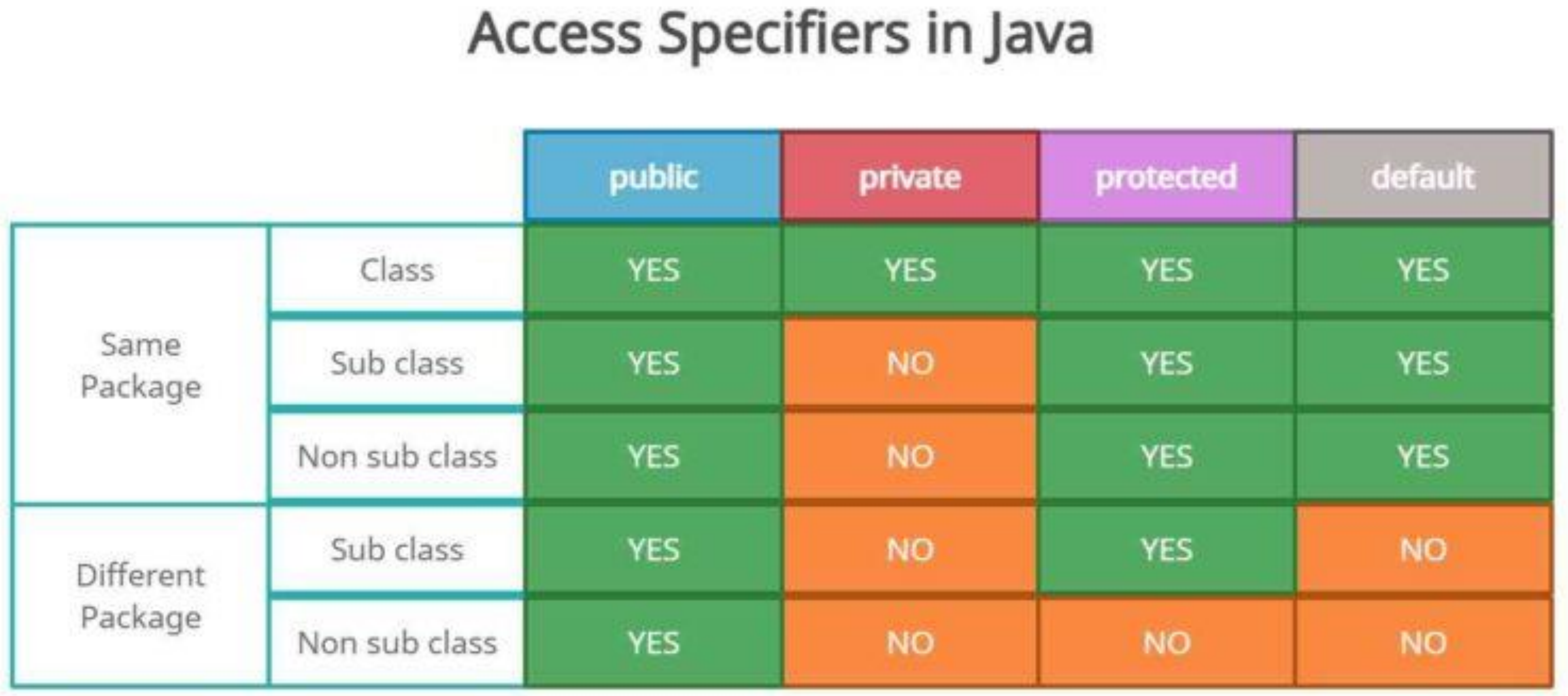
```
import java.io.*;
class Default {
    Default() { System.out.println("Default constructor"); }
    public static void main(String[] args)
    {
        Defaulthello = new Default();
    }
}
```

Types of constructors : Parameterized

```
import java.io.*;
class Employee{
    String name;
    int id;
    Employee(String name, int id){
        this.name = name;
        this.id = id; }}
class Office{
    public static void main(String[] args){
        Employee e1 = new Employee ("avinash", 68);
        System.out.println("GeekName :" + e1.name
                        + " and GeekId :" + e1.id);}}
```

Types of Constructors – copy constructor

```
public class Main {
    private int data;
    public Main() {
       System.out.println("Default constructor called.");}
    public Main(int val) {
        this.data = val;
        System.out.println("Parameterized constructor called with value " + val + ".");}
    public Main(Main other) {
        this.data = other.data;
        System.out.println("Copy constructor called to copy from another object.");}
    public void displayData() {
        System.out.println("Data: " + data);}
    public static void main(String[] args) {
        Main obj1 = new Main();
        obj1.displayData();
        Main obj2 = new Main(obj1);
        obj2.displayData();}}
```



Access Specifiers in Java

		public	private	protected	default
Same Package	Class	YES	YES	YES	YES
	Sub class	YES	NO	YES	YES
	Non sub class	YES	NO	YES	YES
Different Package	Sub class	YES	NO	YES	NO
	Non sub class	YES	NO	NO	NO

Static keyword

The static keyword in Java is used to share the same variable or method of a given class.

Characterstics

- Shared memory allocation
- Accessible without object instantiation
- Associated with class, not objects
- Cannot access non-static members
- Static methods can be overloaded, but not overridden

Static class

a static nested class is a class that is declared as a static member of another class.

does not have access to the instance variables and methods of the outer class.

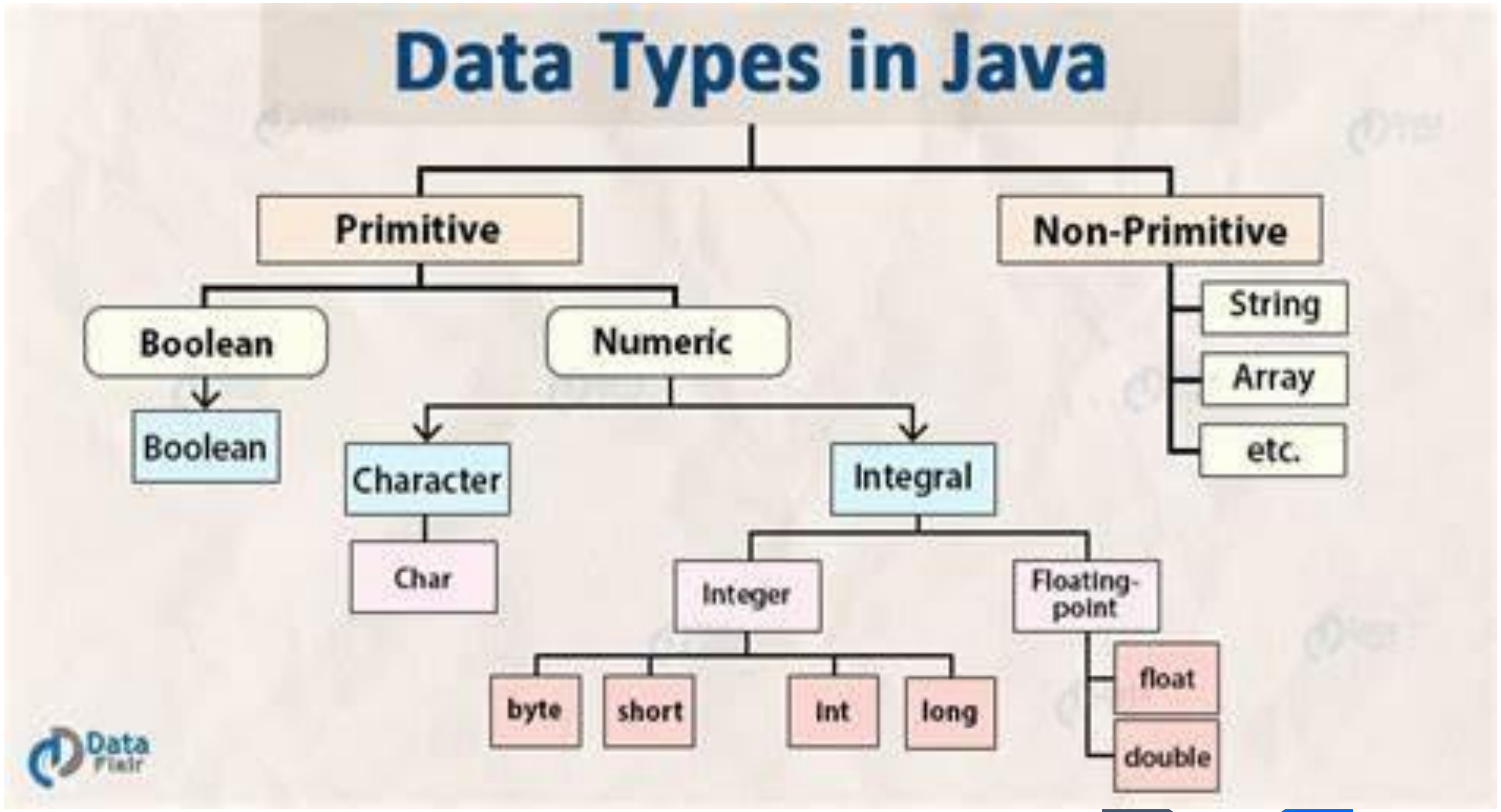
```
class OuterClass {
    private static String msg = "Hello";
    private String msg1 = "Hai";
    public static class NestedStaticClass {
        public void printMessage(){
            System.out.println("Message from nested static class: " + msg);
            System.out.println("Message from nested static class with non
static variable: " + msg1);}}
    public class InnerClass {

        public void display()
        {    System.out.println("Message from non-static nested class with
static msg: "+ msg);
            System.out.println(    "Message from non-static nested class
with non static msg: "+ msg1);}
    }}
class Main {
    public static void main(String args[]){
        OuterClass.NestedStaticClass printer= new
OuterClass.NestedStaticClass();
        printer.printMessage();
        OuterClass outer = new OuterClass();
        OuterClass.InnerClass inner    = outer.new InnerClass();
        inner.display();
        OuterClass.InnerClass innerObject    = new OuterClass().new
InnerClass();
        innerObject.display();}
}
```

Static method

A method that belongs to a class rather than to any particular instance of that class.

This means that you can call a static method on the class itself, without creating an instance of the class.

```
class Myclass{
    public static void display(){
    System.out.println("Hello");
}
}
class Main{
Public static void main(String[] args){
Myclass.display();
}
}
```


Data Types in Java
Primitive
Non-Primitive
Boolean
Numeric
Boolean
Character
Integral
Char
Integer
Floating-point
byte
short
int
long
float
double
String
Array
etc.
Data Flair

Primitive data type	Non-primitive data type
Defined by language itself	Defined by user
When declared stored in stack	When declared the object is stored in heap, but reference variables are stored in stack
When a copy is created it creates a complete new allocation for the copy	When a copy is created a reference variable is created but it points to same object in heap
Changes made to copy don’t change the original	Changes made to copy trace back to the original
Cant have null value	Can have null value
Size is fixed, depends on the data structure	Size is not fixed

Partial keyword

Definition of class, interface, or structure can be split across multiple source files.

Each part of the partial class, interface, or structure contributes to the same type, and the compiler combines them into a single type during compilation.

Partial Class

```
// File1.cs
partial class MyClass {
    public void Method1()
{ /* Implementation */ }
}
// File2.cs
partial class MyClass {
    public void Method2()
{ /* Implementation */ }
}
MyClass myObject = new
MyClass();
myObject.Method1();
myObject.Method2();
```

Partial method

```
// File1.cs
partial class MyClass {
    partial void
MyPartialMethod();
}
// File2.cs
partial class MyClass {
    partial void
MyPartialMethod() {
        // Implementation
    }
}
```

Partial Interface

```
// File1.cs
partial interface
IMyInterface {
    void Method1();
}
// File2.cs
partial interface
IMyInterface {
    void Method2();
}
```

Destructor

Responsible for cleaning up resources or performing finalization tasks when an object is no longer needed.

Opposite of constructors(initialising objects)

Cleans up memory

Automatically invoked

Finalisation process

Using Idisposable interface we use destructors in c#

```
public class MyClass : IDisposable {
    // Constructor
    public MyClass() {
        Console.WriteLine("Constructor
called");
    }

    // Method to simulate resource
usage
    public void UseResource() {
        Console.WriteLine("Resource
used");
    }

    // Implementation of IDisposable
interface
    public void Dispose() {
        // Clean up resources
        Console.WriteLine("Dispose
method called");
    }
}
```

Thank you

Palla Nikhita

nikhita_palla@epam.com