SNS COLLEGE OF ENGINEERING

Kurumbapalayam (Po), Coimbatore – 641 107

An Autonomous Institution

Accredited by NAAC – UGC with 'A' Grade

Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

23CSB101

OBJECT ORIENTED PROGRAMMING

OOPS CONCEPTS

By

M.Kanchana

Assistant Professor/CSE

OBJECT ORIENTED PROGRAMMING

Object means a real-world entity such as a pen, chair, table, computer, watch, etc.

Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects.

The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

OBJECT ORIENTED PROGRAMMING

It simplifies software development and maintenance by providing some concepts:

OOPs Concepts:

- Objects
- Class
- Data Abstraction
- Encapsulation
- Inheritance
- Polymorphism
- Dynamic Binding
- Message Passing

One Clever Dude Enjoys Ice cream, Playing Daily Music.

OBJECT

- Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical.
- An Object can be defined as an instance of a class. It contains an address and takes up some space in memory.

For example “Dog” is a real-life Object, which has some characteristics like color, Breed, Bark, Sleep, and Eats.

CLASS

- Collection of objects is called class.
- It is a logical entity.
- A class can also be defined as a blueprint from which you can create an individual object.
- Class does not consume any space.

One class

Many objects

CLASS_SYNTAX

```
class <class_name>
{
field;
 method;
}
```

class_name object_name = **new** class_name();

S.No.	Object	Class
1)	Object is an **instance** of a class.	Class is a **blueprint or template** from whichobjects are created.
2)	Object is a **real world entity** such aspen, laptop, mobile, bed, keyboard, mouse, chair etc.	Class is a **group of similar objects**.
3)	Object is a **physical** entity.	Class is a **logical** entity.
4)	Object is created through **new keyword** mainly e.g. Student s1=new Student();	Class is declared using **class keyword** e.g.class Student{}
5)	Object is created **many times** as per requirement.	Class is declared **once**.
6)	Object **allocates memory when it is created**.	Class **doesn't allocated memory when it is created**.
7)	There are **many ways to create object** in java such as new keyword, newInstance() method, clone() method, factory method and deserialization.	There is only **one way to define class** in java using class keyword.

Program

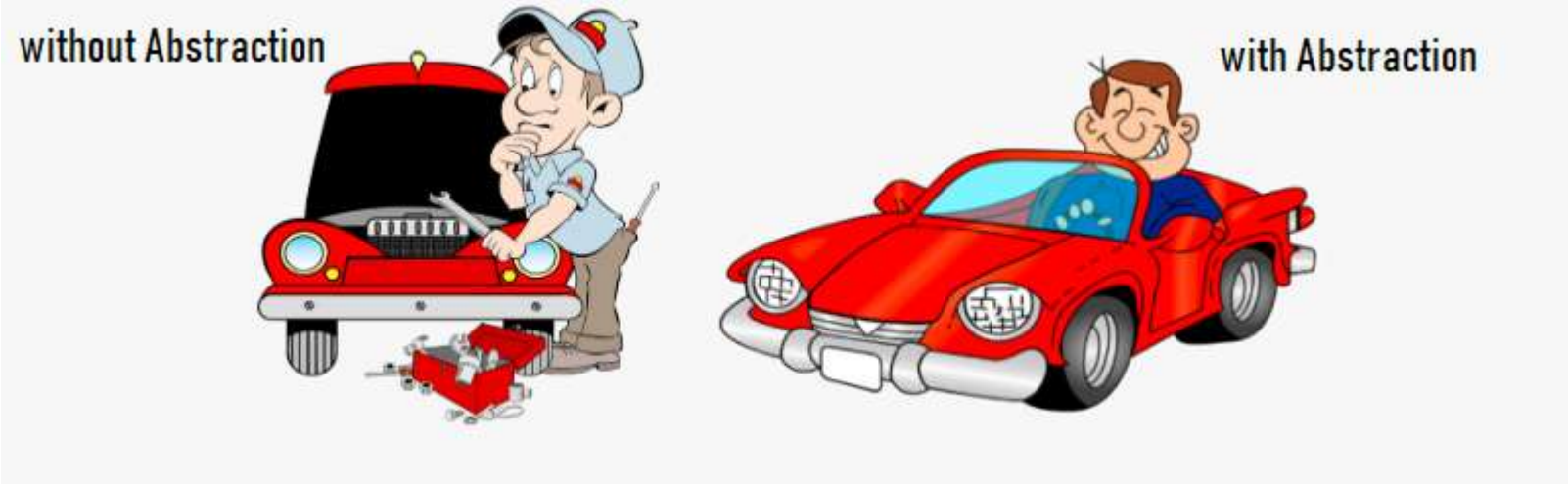
```
class Dog {
String name; //field
void displayName() //method
 {
System.out.println("The dog's name is: " + name);
 }
public static void main(String[] args)
{
Dog myDog = new Dog(); // object creation
myDog.name = "Buddy";
myDog.displayName();
}
}
```

Output

The dog's name is: Buddy

Data abstraction

- Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.



Data abstraction

- This is achieved using **abstract classes and methods.**
- An abstract class cannot be instantiated directly and may contain abstract methods—methods without a body that must be implemented by subclasses.
- This mechanism enables you to define a template for other classes to follow.

```
abstract class Animal {
    abstract void displayName();
}
```

Encapsulation

- Binding (or wrapping) code and data together into a single unit are known as encapsulation. For example, a capsule, it is wrapped with different medicines.

Encapsulation = Data Hiding + Abstraction

Encapsulation

Suppose you have an account in the bank. If your balance variable is declared as a public variable in the bank software, your account balance will be known as public, In this case, anyone can know your account balance.

So, would you like it?

Encapsulation

We declare balance variable as private for making your account safe, so that anyone cannot see your account balance.

The person who has to see his account balance, will have to access only private members through methods defined inside that class and this method will ask your account holder name or user Id, and password for authentication.

Data hiding

It prevents to access data members (variables) directly from outside the class so that we can achieve security on data.

Encapsulation = Data Hiding + Abstraction

```
public class Account
{
 private double balance;
 public double getbalance()
 {
   return balance;
 } }
```

Encapsulation

```
class Dog
{
private String name;
public void setName(String n)
{
 name = n;
}
void displayName()
 {
System.out.println("The dog's name is: " + name);
 }
public static void main(String[] args)
{
Dog myDog = new Dog();
myDog.setName(“SubraMani”);
myDog.displayName();
}
}
```

Output:

The dog's name is: SubraMani

Encapsulation

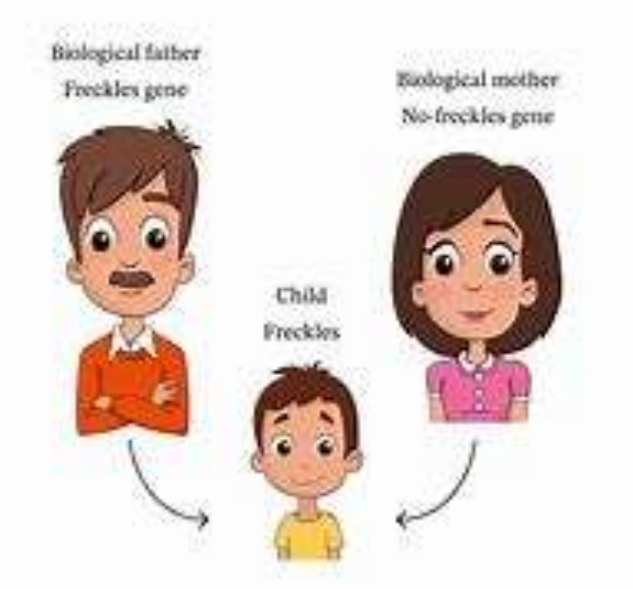
- ✓ Private field (name) → Cannot be accessed directly.
- ✓ Public method (setName()) → Controls how data is modified.
- ✓ Public method (displayName()) → Controls how data is accessed.

public: The member is accessible from any other class, anywhere.

private: The member is accessible only within the same class.

Inheritance

- Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object
- Inheritance allows the user to reuse the code whenever possible and reduce its redundancy.

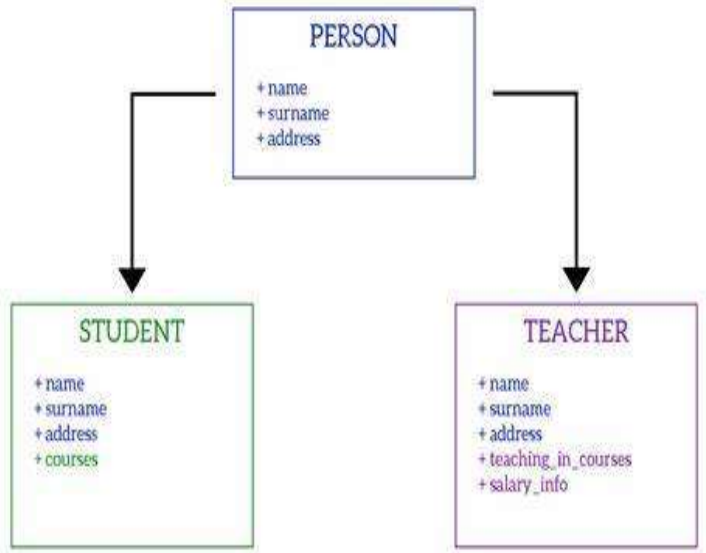


Inheritance

- **subclass** (child) - the class that inherits from another class
- **superclass** (parent) - the class being inherited from

To inherit from a class, use the extends keyword.

```
class Subclass extends Superclass
{
  //methods and fields
}
```



Inheritance

```
class Animal {
   String name;
   void DisplayName() {
      System.out.println("The animal's
name is: " + name);
   }
}

class Dog extends Animal {
    void setName(String name) {
      this.name = name;
   }
}

class Cat extends Animal {
     void setName(String name) {
      this.name = name;
}
}
```

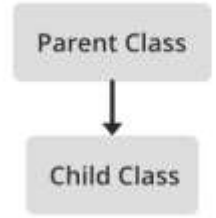
```
public class Main {
   public static void main(String[]
args) {
Dog myDog = new Dog();
      myDog.setName("Wolffff");
      myDog.DisplayName();


        Cat myCat = new Cat();
      myCat.setName("Tigerrr");
      myCat.DisplayName();
   }
}
```

Inheritance

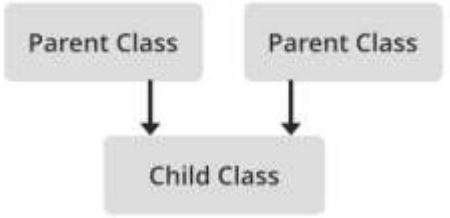
Output

The animal's name is: Wolffff
The animal's name is: Tigerrr

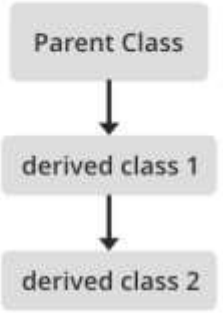
Inheritance Types



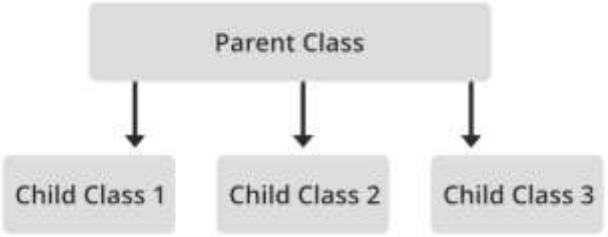
Single inheritance



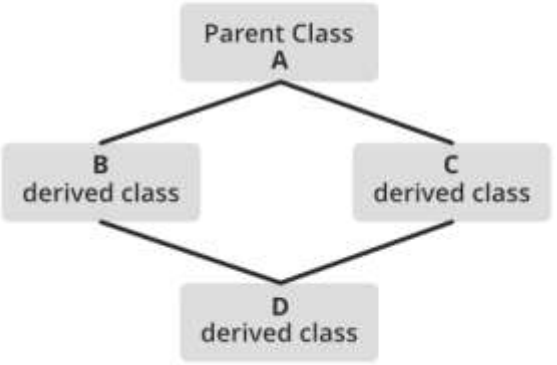
Multiple inheritance



Multilevel inheritance



Hierarchical inheritance



Hybrid Class

Polymorphism

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

For Example:- Suppose if you are in a classroom that time you behave like a student, when you are in the market at that time you behave like a customer,when you at your home at that time you behave like a son or daughter, Here one person present in different-different behaviours.

Types of Polymorphism

1. Compile time polymorphism / Method Overloading
2. Runtime polymorphism / Method Overriding

1.Compile-time Polymorphism (Static Polymorphism):
Compile-time polymorphism occurs when the method to be executed is determined at compile time (before the program runs). The most common example is **Method Overloading**.

- Java **chooses** the correct method based on the method signature (the number and types of parameters)

Types of Polymorphism

```
class Dog {
String name;
void displayName()
{
System.out.println("The dog's name is: " + name);
 }
void displayName(String owner )
 {
System.out.println("The dog's name is: " + name " and its owner is: " + owner);
 }
public static void main(String[] args)
{
Dog myDog = new Dog(); // object creation
myDog.name = "Buddy";
myDog.displayName();
myDog.displayName( “Babu”);
}
}
```

Output:

The dog's name is: Buddy

The dog's name is: Buddy

and its owner is: John

Types of Polymorphism

2. Run-time Polymorphism (Dynamic Polymorphism):
Run-time polymorphism occurs when the method to be executed is determined at runtime (when the program is running). The most common example is **Method Overriding**.
The method **called depends on the actual object type** (not the reference type).

Runtime polymorphism is a concept that uses dynamic binding.

Run Polymorphism

```
class Animal {
    String name;
    void relation() {  // Overridable method
        System.out.println("Animals have different
relations.");
    }
    void setName(String name) {  // Overridable
method
        System.out.println("The animal's name is: "
+ name);
    }
}
class Dog extends Animal {
    @Override
    void relation() {
        System.out.println("Wolves and dogs are
related.");
    }
     @Override
    void setName(String name) {
        System.out.println("The animal's name is
from Dog class: " + name);
    }
}
```

```
class Cat extends Animal {
    @Override
    void relation() {
        System.out.println("Tigers and cats are related.");
    }

    @Override
    void setName(String name) {
        System.out.println("The animal's name is from Cat
class: " + name);
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myAnimal;  // Parent class reference
        myAnimal = new Dog();  // Dog object assigned
        myAnimal.setName("Wolffff");
        myAnimal.relation();
        myAnimal = new Cat();  // Cat object assigned
        myAnimal.setName("Tigerrr");
        myAnimal.relation();
    }
}
```

Run Polymorphism

Output

The animal's name is from dog class: Wolffff
wolves and dogs are related

The animal's name is from Cat class: Tigerrr
Tiger and Cats are related

Dynamic Binding

- Dynamic Binding (also called **Late Binding**) refers to the process where the method to be called is determined **at runtime** rather than compile time.
- It happens when a **parent class reference** is used to refer to a **child class object** and an **overridden method** is invoked.

Dynamic binding enables runtime polymorphism.

Dynamic Binding

```
class Animal {
   String name;
   void DisplayName(String name) {
      System.out.println("The animal's
name is from animal class: " + name);
   }
}

class Dog extends Animal {
    void setName(String name) {
      System.out.println("The animal's
name is from dog class: " + name);
   }
}

class Cat extends Animal {
     void setName(String name) {
    System.out.println("The animal's name
is from Cat class: " + name);
}
}
```

```
public class Main {
   public static void main(String[]
args) {
Animal myDog = new Dog();
      Dog mydog= new Dog();
mydog.setName("Wolffff");
      myDog.DisplayName("Dog");
           Animal myCat = new Cat();
 Cat mycat= new Cat();
      mycat.setName("Tigerrr");
      myCat.DisplayName("Cat");
   }
}
```

Dynamic Binding

Output

The animal's name is from **dog class**: Wolffff

The animal's name is from **animal class**: Dog

The animal's name is from **Cat class**: Tigerrr

The animal's name is from **animal class**: Cat

Message Passing

Message passing is a fundamental concept in OOP where objects communicate with each other by sending and receiving messages (i.e., method calls). It allows encapsulation and abstraction, making it easier to design modular and maintainable software.

Program

```
class Dog {
String name; //field
void displayName() //method
 {
System.out.println("The dog's name is: " + name);
 }
public static void main(String[] args)
{
Dog myDog = new Dog(); // object creation
myDog.name = "Buddy"; // Message passing (Method call)
myDog.displayName();// Message passing (Method call)
}
}
```

Output

The dog's name is: Buddy

PUZZLE

- A **father** passes down certain characteristics to his **child**. The child can use these traits but may also develop their own.

- Imagine a **button** on a website. When you click it, sometimes it **submits a form**, sometimes it **opens a new page**, and sometimes it **plays a sound**.

- You have a **personal diary** with a **lock**. Only you can **read or write** in it. However, others can ask you **certain questions**, and you decide whether to share the answers.

- An architect creates a **blueprint** for a house. The blueprint **does not** show **actual bricks, paint, or furniture**—only the **structure and design**. Different builders can use this blueprint to construct **houses with different materials**.

THANK YOU