Classes & Objects:

```
class Car {
        private int number;
        private String model;
        private String color;
        private double price;
        private int launchedYear;

        // Paramterized Constructor
        // Getters and Setters
}

Car Hyundai = new Car(1, "XYZ", "Red", 1000000, 2024);
```

Class is a blueprint to create objects.

Object is the real world entity constructed using the class.

Car is a template or blueprint to create the object.

Hyundai is the real world entity which was created by using the Car class.

Polymorphism:

Poly means many and morphism means forms.

Polymorphism refers to the ability of the objects to respond to the same method in different ways.

Compile Time Polymorphism:

```
class Calculator {
        public int add (int a, int b) {
```

```
        return a + b;
    }
    public int add (int a, int b, int c) {
        return a + b + c;
    }
}

Calculator calculator1 = new Calculator();
calculator1.add(5, 6);
calculator1.add(9, 12, 3);
```

Run Time Polymorphism:

```
class Person {
    private String name;
    private int age;
    private int id;

    // Parameterized Constructor
    // Getters and Setters

    @Override
    public String toString() {
        return this.name+" "+this.age+" "+this.id;
    }
}

Person maha = new Person("Maha", 28, 1);
maha.toString(); // toString method implemented in Person class will be executed.
```

Inheritance:

Inheritance refers to extending the properties and behaviours of the parent class.

Types:

1. Single
2. Multi Level
3. Hybrid

Single Inheritance:

A subclass can inherit from only one superclass.

```
class Parent {
    private int prop1;
    private int prop2;

    // Getters and Setters

    // Parent Functions
}

class Child extends Parent { // Child extended the properties of it parent class.

}
```

Child child = new Child(); // child object has access to all the properties and functions that are defined in it parent class.

child.getProp1(); // Inherited from Parent

child.getProp2();

Multi Level:

A subclass which inherits properties from the parent class, which in turn inherits from another class, forming a chain.

```
class GrandParent {
	private String familyName;

	// Getters and Setters
}

class Parent extends GrandParent {
	private String parentName;

	public Parent (String familyName, String parentName) {
		this.parentName = parentName;
		super(familyName);
	}

	// Getters and Setters
}

class Child extends Parent {
	private String childName;

	public Child (String familyName, String parentName, String childName) {
		this.childName = childName;
		super(familyName, parentName);
	}

	// Getters and Setters
```

```
}

Child child = new Child ("Smith", "John", "Alex");
child.getFamilyName();
child.getParentName();
child.getChildName();
```

Hierarchical:

One parent class can have multiple child classes that inherit from it.

```
class Parent {
	private int prop1;
	private int prop2;
	private int prop3;

	// Getters and Setters
}

class Child1 extends Parent {

}

class Child2 extends Parent {

}

class Child3 extends Parent {

}
```

```
class Child4 extends Parent {

}
```

Abstraction:

Hiding the internal implementation details and showing only the functionalities to the user.

Abstraction can be achieved in two ways:

1. Interface
2. Abstract Class

Interface (can have only abstract methods):

```
interface IATMService {
	double getBalance();
	void deposit(Sting accountNumber, double creditAmount);
	void withdraw(String accountNumber, double debitAmount);
}

class ATMServiceImpl implements IATMService {
	@Override
	public double getBalance() {
		// Get Balance logic.
	}
	@Overirde
	public void deposit(String accountNumber, double creditAmount) {
		// Deposit logic.
	}
	@Override
	public void withdraw(String accountNumber, double debitAmount) {
		// Withdraw logic.
```

```
	}
}
```

Abstract Class: (can have both abstract and non-abstract methods in it)

```
class abstract ATMService {
	private String machineName;
	public String getMachineName() {
		return this.machineName;
	}
	double getBalance();
	void deposit(Sting accountNumber, double creditAmount);
	void withdraw(String accountNumber, double debitAmount);
}

class ATMServiceImpl extends ATMService {
	@Override
	public double getBalance() {
		// Get Balance logic.
	}
	@Overirde
	public void deposit(String accountNumber, double creditAmount) {
		// Deposit logic.
	}
	@Override
	public void withdraw(String accountNumber, double debitAmount) {
		// Withdraw logic.
	}
}
```

Encapsulation:

Binding data and methods into a single unit is called as Encapsulation.

```
class Person {
        private int id;
        private String name;
        private int age;

        // Parameterized Constructor

        // Getters and Setters
}
```

Encapsulation - Helps to achieve the following: Restricting the direct access to class member by using the objects and allowing only to access them through methods.

```
Person maha = new Person(1, "Maha", 28);
maha.name = "Mahalakshmi"; // compile time error
maha.age = 27; // compile time error
maha.setAge(27); // Age will be successfully gets updated.
```