**Java OOPS**

**Concepts of OOPS:**

1. Class
2. Object
3. Method and Method Passing
4. **Pillars of OOPs**
   1. **Abstraction**
   2. **Encapsulation**
   3. **Inheritance**
   4. **Polymorphism**
      1. **Compile-time polymorphism**
      2. **Runtime polymorphism**

**Class**

A class is a user-defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. Using classes, you can create multiple objects with the same behavior instead of writing their code multiple times. This includes classes for objects occurring more than once in your code. In general, class declarations can include these components in order:

1. Modifiers: A class can be public or have default access (Refer to this for details).
2. Class name: The class name should begin with the initial letter capitalized by convention.
3. Superclass (if any): The name of the class’s parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
4. Interfaces (if any): A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
5. Body: The class body is surrounded by braces, { }.

**Object**

An object is a basic unit of Object-Oriented Programming that represents real-life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. The objects are what perform your code, they are the part of your code visible to the viewer/user. An object mainly consists of:

1. State: It is represented by the attributes of an object. It also reflects the properties of an object.
2. Behavior: It is represented by the methods of an object. It also reflects the response of an object to other objects.
3. Identity: It is a unique name given to an object that enables it to interact with other objects.
4. Method: A method is a collection of statements that perform some specific task and return the result to the caller. A method can perform some specific task without returning anything. Methods allow us to reuse the code without retyping it, which is why they are considered time savers. In Java, every method must be part of some class, which is different from languages like C, C++, and Python.

**Pillars of OOPs**

**Pillar 1: Abstraction**

Data Abstraction is the property by virtue of which only the essential details are displayed to the user. The trivial or non-essential units are not displayed to the user. Ex: A car is viewed as a car rather than its individual components.  
Data Abstraction may also be defined as the process of identifying only the required characteristics of an object, ignoring the irrelevant details. The properties and behaviors of an object differentiate it from other objects of similar type and also help in classifying/grouping the object.  
**Consider a real-life example** **of a person driving a car.** The man only knows that pressing the accelerators will increase the car speed or applying brakes will stop the car, but he does not know how on pressing the accelerator, the speed is actually increasing. He does not know about the inner mechanism of the car or the implementation of the accelerators, brakes etc. in the car. This is what abstraction is.   
In Java, abstraction is achieved by interfaces and abstract classes. We can achieve 100% abstraction using interfaces.

**Pillar 2: Encapsulation**

It is defined as the wrapping up of data under a single unit. It is the mechanism that binds together the code and the data it manipulates. Another way to think about encapsulation is that it is a protective shield that prevents the data from being accessed by the code outside this shield.

* Technically, in encapsulation, the variables or the data in a class is hidden from any other class and can be accessed only through any member function of the class in which they are declared.
* In encapsulation, the data in a class is hidden from other classes, which is similar to what data-hiding does. So, the terms “encapsulation” and “data-hiding” are used interchangeably.
* Encapsulation can be achieved by declaring all the variables in a class as private and writing public methods in the class to set and get the values of the variables.
* In the encapsulation technique, we declare fields as private in the class to prevent other classes from accessing them directly. The required encapsulated data can be accessed by using public Java getter and setter method.
* If the field is declared private in the class then it cannot be accessed by anyone from outside the class and hides field within the class. Therefore, it is also called **data hiding**.
* **Real-life example :If combinations of medicine are variables and methods then the capsule will act as a class and the whole process is called Encapsulation** as shown in the below figure.
* Diagram

  Description automatically generated

**Pillar 3: Inheritance**

Inheritance is an important pillar of OOP (Object Oriented Programming). It is the mechanism in Java by which one class is allowed to inherit the features (fields and methods) of another class.

Let us discuss some frequently used important terminologies:

* Superclass: The class whose features are inherited is known as superclass (also known as base or parent class).
* Subclass: The class that inherits the other class is known as subclass (also known as derived or extended or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
* Reusability: Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.
* **Real-life example – Parent-child**

**Diagram

Description automatically generated**

**Pillar 4: Polymorphism**

It refers to the ability of object-oriented programming languages to differentiate between entities with the same name efficiently. This is done by Java with the help of the signature and declaration of these entities.

The best **real-life example of polymorphism is human behavior.** One person can have different behavior.

**Introducing Classes**

1. **General form of a class**

class Classname {

type instance-variable1;

type instance-variable2;

// ...

type instance-variableN;

type methodname1(parameter-list) { // body of method

}

type methodname2(parameter-list) {

// body of method }

// ...

type methodnameN(parameter-list) {

// body of method }

}

1. **Some key points for classes**
   1. The data, or variables, defined within a class are called instance variables.
   2. The code is contained within methods.
   3. Collectively, the methods and variables defined within a class are called members of the class.
   4. In most classes, the instance variables are acted upon and accessed by the methods defined for that class.
   5. Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another. We will come back to this point shortly, but it is an important concept to learn early.
   6. All methods have the same general form as main( ), which we have been using thus far. However, most methods will not be specified as static or public. Notice that the general form of a class does not specify a main( ) method. Java classes do not need to have a main( ) method. You only specify one if that class is the starting point for your program.
2. **A very simple class**

class Box {

double width;

double height;

double depth;

}

1. **Creating an object**
   1. As stated, a class defines a new type of data. In this case, the new data type is called Box. You will use this name to declare objects of type Box. It is important to remember that a class declaration only creates a template; it does not create an actual object. Thus, the preceding code does not cause any objects of type Box to come into existence.
   2. To actually create a Box object, you will use a statement like the following:
   3. **Box mybox = new Box();** // create a Box object called mybox
   4. After this statement executes, mybox will be an instance of Box. Thus, it will have “physical” reality.
2. **Dot Operator**
   1. As mentioned earlier, each time you create an instance of a class, you are creating an object that contains its own copy of each instance variable defined by the class.
   2. Thus, every Box object will contain its own copies of the instance variables width, height, and depth. To access these variables, you will use the dot (.) operator.
   3. The **dot operato**r links the name of the object with the name of an instance variable. For example, to assign the width variable of mybox the value 100, you would use the following statement:
   4. **mybox.width = 100;**
   5. This statement tells the compiler to assign the copy of width that is contained within the mybox object the value of 100. In general, you use the dot operator to access both the instance variables and the methods within an object.
3. **A program that uses Box class**

/\* A program that uses the Box class.

Call this file BoxDemo.java

\*/

class Box {

double width;

double height;

double depth;

}

// This class declares an object of type Box.

class BoxDemo {

public static void main(String args[]) {

Box mybox = new Box();

double vol;

// assign values to mybox's instance variables

mybox.width = 10;

mybox.height = 20;

mybox.depth = 15;

// compute volume of box

vol = mybox.width \* mybox.height \* mybox.depth;

System.out.println("Volume is " + vol);

}

}

1. **Compiling and Running the program**
   1. You should call the file that contains this program BoxDemo.java, because the main( ) method is in the class called BoxDemo, not the class called Box. When you compile this program, you will find that two .class files have been created, one for Box and one for BoxDemo.
   2. The Java compiler automatically puts each class into its own .class file. It is not necessary for both the Box and the BoxDemo class to actually be in the same source file. You could put each class in its own file, called Box.java and BoxDemo.java, respectively.
   3. To run this program, you must execute BoxDemo.class. When you do, you will see the following output:

Volume is 3000.0

1. **Creating multiple objects**
   1. each object has its own copies of the instance variables. This means that if you have two Box objects, each has its own copy of depth, width, and height. It is important to understand that changes to the instance variables of one object have no effect on the instance variables of another.
   2. Program:

class Box {

double width;

double height;

double depth;

}

class BoxDemo2 {

public static void main(String args[]) {

Box mybox1 = new Box();

Box mybox2 = new Box();

double vol;

// assign values to mybox1's instance variables

mybox1.width = 10;

mybox1.height = 20;

mybox1.depth = 15;

/\* assign different values to mybox2's

instance variables \*/

mybox2.width = 3;

mybox2.height = 6;

mybox2.depth = 9;

// compute volume of first box

vol = mybox1.width \* mybox1.height \* mybox1.depth;

System.out.println("Volume is " + vol);

// compute volume of second box

vol = mybox2.width \* mybox2.height \* mybox2.depth;

System.out.println("Volume is " + vol);

} }

* 1. The output produced by this program is shown here:   
     Volume is 3000.0

Volume is 162.0

* 1. As you can see, mybox1’s data is completely separate from the data contained in mybox2.

**Declaring Objects**

1. **Using the new keyword**
   1. When we create a class, you are creating a new data type
   2. obtaining objects of a class is a two-step process.
   3. First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can refer to an object.
   4. Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the new operator.
   5. **The new operator dynamically allocates** (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by new. This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated.

Box mybox = new Box();

Box mybox; // declare reference to object

mybox = new Box(); // allocate a Box object

1. **Closer look at the new**
   1. As just explained, the new operator dynamically allocates memory for an object. It has this general form:

class-var = new classname( );

* 1. Diagram

     Description automatically generated with medium confidence
  2. The class name followed by parentheses specifies the constructor for the class. A constructor defines what occurs when an object of a class is created.
  3. **Constructors** are an important part of all classes and have many significant attributes. Most real-world classes explicitly define their own constructors within their class definition. However, if no explicit constructor is specified, then Java will automatically supply a default constructor. This is the case with Box.
  4. For now, we will use the default constructor. Soon, you will see how to define your own constructors.

1. **Why are primitives different?**
   1. At this point, you might be wondering why you do not need to use new for such things as integers or characters.
   2. The answer is that Java’s primitive types are not implemented as objects. Rather, they are implemented as “normal” variables.
   3. This is done in the interest of efficiency. As you will see, objects have many features and attributes that require Java to treat them differently than it treats the primitive types. By not applying the same overhead to the primitive types that applies to objects, Java can implement the primitive types more efficiently.
2. **Distinction b/w class and object – Review**
   1. A class creates a new data type that can be used to create objects. That is, a class creates a logical framework that defines the relationship between its members. When you declare an object of a class, you are creating an instance of that class. Thus, a class is a logical construct.
   2. An object has physical reality. (That is, an object occupies space in memory.) It is important to keep this distinction clearly in mind.
3. **Assigning Object Reference Variables**
   1. Object reference variables act differently than you might expect when an assignment takes place. For example, what do you think the following fragment does?

Box b1 = new Box();

Box b2 = b1;

* 1. You might think that b2 is being assigned a reference to a copy of the object referred to by b1. That is, you might think that b1 and b2 refer to separate and distinct objects. However, this would be wrong.
  2. Instead, after this fragment executes, b1 and b2 will both refer to the same object. The assignment of b1 to b2 did not allocate any memory or copy any part of the original object.
  3. It simply makes b2 refer to the same object as does b1. Thus, any changes made to the object through b2 will affect the object to which b1 is referring, since they are the same object.
  4. FigureDiagram

     Description automatically generated
  5. Although b1 and b2 both refer to the same object, they are not linked in any other way. For example, a subsequent assignment to b1 will simply unhook b1 from the original object without affecting the object or affecting b2. For example:

Box b1 = new Box();

Box b2 = b1;

// ...

b1 = null;

* 1. Here, b1 has been set to null, but b2 still points to the original object.