Module: Core Java

Session 1: Getting Started with OO

* What is OO thinking?
* OOP: A leap from switches to objects
* An obvious, yet brilliant solution
* Object-Oriented Programming
* Benefits of OOPs
* Example: Encapsulation, Inheritance, and Polymorphism

**Objective**

At the end of this session, you will be able to:

* Advance from switches to objects
* Understand and use Object-Oriented Programming concepts
* Have a better idea about Abstraction, Encapsulation, Inheritance and Polymorphism
* Know the benefits of OOPS
* Learn from examples of Encapsulation, Inheritance and Polymorphism

**What is OO thinking?**

Programming languages, like spoken languages, evolve from time to time. They are constantly refined and developed to meet the ever-changing needs of users. Like other modern programming languages such as C++, Java uses a combination of techniques developed over the years. Therefore, we will start exploring Object Oriented Programming (OOP) by briefly looking at the history of programming languages and their consequent developments.

An idea of where object-oriented ideas originated from will help you to better understand why they are an important part of modern programming languages. Once you understand why OOP was developed, you will know exactly what makes a programming language object-oriented.

## OOP: A leap from Switches to Objects

Back in the earlier days of computing and data encryption, technicians used to create computer programs by turning on and turning off, hundreds of switches. Each switch represented a bit. In those days, programming even simple programs needed a lot of patience and precision. As the need for more sophisticated programs grew, so did the need for better ways to write these programs. The need to make computer programming quicker and simpler acted as the primary stimulus for the invention of assembly language and high-level languages like FORTRAN. With the help of these high-level languages, programmers could frame and initiate comprehensive programs that were not only precise and exact but also task-efficient. Using a single command such as PRINT, they could print a thousand documents.

Programmers then needed a new way of using high-level languages that would enable them to partition their programs into logical sections. Eventually, the structured-programming concept was developed. Any structured program has a top-down approach. In structured programming, the programmer primarily focuses on the general functions of the program rather than the details of how the functions are implemented. With a top-down approach, programmers can easily handle large projects without producing any twisted code.

For an analogy, let us consider an everyday task such as cleaning a house. If you wanted to write out the steps needed to complete this task, you would have written something like this:

* Go to the living room
* Dust the coffee table
* Dust the study table
* Vacuum the rug
* Go to the Kitchen
* Wash the dishes
* Wipe the counters
* Clean the stove
* Wipe off the refrigerator
* Sweep the floor
* Go to the bedroom
* Clean the wardrobe
* Make the bed

The preceding list of steps is similar, in theory, to how you would create a program without using a top-down approach. Using the top-down programming approach the same program will look like:

* **TOP LEVEL**
* Clean the Living Room
* Clean the Kitchen
* Clean the Bedroom
* **SECOND LEVEL**
* Clean the Living Room

START  
  Go to the living room  
  Vacuum the rug

. . //All the activities in the living room

END

* Clean the Kitchen

START  
 Go to the Kitchen  
     Wash the dishes

. . //All the activities in the kitchen room

END

* Clean the Bedroom

START  
     Go to the bedroom  
 Make the bed

. . //All the activities in the bedroom

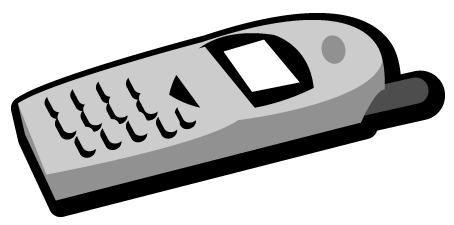
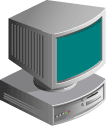
END

Now, if you are only interested in seeing what the "program" does or how the program is framed, you can glance at the top level and see that these are instructions for cleaning the living room, kitchen, and bedroom. If, however, you want to know exactly how to clean the living room, you can go down one level in the top-down structure and find the detailed instructions for cleaning the living room. In a nutshell, a top-down approach tends to make programs longer, but it also adds clarity to the program, because you can hide the details until you really need them.

Today, the need for efficient programming methods is more important than ever. The size of an average computer program has grown dramatically and now consists of hundreds of thousands of lines coded in different forms. With these huge programs, reusability is critical. Again, a better way of programming is needed and that better way is object-oriented programming.

## An obvious, yet brilliant solution

The world consists of many objects like car, bottle, chair, cell phone, computer, pen etc. Even you are also an Object!



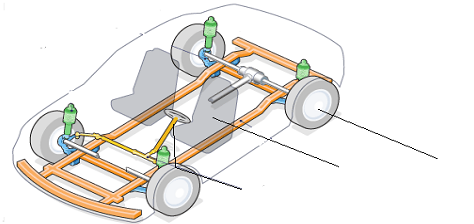
I am also an object.

Fig. 1: Objects in the world

Every object, be it a pen, or a chair or ‘You’, have attributes and behavior as shown in Fig.1. All objects have knowledge and each piece of the knowledge is called an attribute. For example, ink in a pen is an attribute of pen. The functionality of an object is called behavior. For example, writing is a behavior of pen.

Most of the objects manipulate other objects or data. For example, a car is an object that manipulates its speed and direction to transport people to a different location. This car object encapsulates all the functions and data that it needs to get its job done. It has a switch to turn it on, a wheel to control its direction of motion, and brakes to slow it down. These functions directly manipulate the car's data, including direction, position, and speed.

When you travel in a car, however, you don't have to know the details of how these operations work. To turn a car, simply revolve the steering wheel. You don't have to know how the instrument turns the car. You simply know that it works.



Wheel

Seat

Steering

R

#### Fig. 2: Aggregation diagram for Objects

All these functions and data work together to define the object called as car. Moreover, all these functions work very similarly from one car to the next. Objects can be composed of other objects.

## Objects can be part of other objects. This relationship between objects is known as aggregationas shown

## in Fig. 2.

## Object-Oriented Programming

From the primordial point of view, a program has been viewed as a logical procedure that takes input data, processes it, and produces output data. The programming challenge was seen as how to write the logic, not how to define the data. Object-oriented programming takes the view that what we really care about are the objects we want to manipulate, rather than the logic required to manipulate them.

Object-oriented programming enables you to think of program elements as objects. In the case of a window object, you do not need to know the details of how it works, nor do you need to know about the window's private data fields. You need to know only how to call the various functions (called methods in Java) that make the window operate. Consider the object car discussed in the previous section. To drive a car, you do not have to know the details of how a car works. You need to know only how to drive it. What is going on under the hood matters very little and in most cases are ignored.

Object-oriented Programming (OOP) is a Programming language model, organized around "**objects**" rather than "**actions**", and “**data**” rather than “**logic**”.

Object Oriented Programming solves three main software engineering goals:

* Re-usability
* Flexibility
* Extensibility

For more details, look at Fig. 3.

**Object-Oriented Software Structures**

**Software Engineering**

**Goals**

Object

Classes

Inheritance

(Polymorphism)

Templates

Design

Patterns

Reusability

Extensibility

Flexibility

Fig. 3: Object Oriented Programming

Examples of objects range from human beings (described by name, address, and so forth) to buildings and floors (whose properties can be described and managed) down to the little widgets on your computer desktop (such as buttons and scroll bars).

## Object-Oriented Programming Concepts

To learn about OOP, you need to understand the main concepts that form the backbone of OOP.

These concepts, which are covered in the following sections, are:

* Class
* Object
* Abstraction
* Encapsulation
* Inheritance
* Polymorphism

### Class

In real world, you will often find many individual objects, all of the same kind. There may be thousands of other cars in existence, all of the same make and model. Each car was built from the same set of blueprints and therefore contains the same components. In object-oriented terms, we say that car is an instance of the class of objects known as Car. A class is the blueprint from which individual objects are created.

A class is a template for multiple objects with similar features. Classes embody all the features of a particular set of objects. When you write a program in an object-oriented language, you do not define actual objects. You define classes of objects.

### Object

Objects are key to understanding object-oriented technology. Look around right now and you will find many examples of real-world objects: your dog, your desk, your television set, your bicycle.

Real-world objects share two characteristics:

* State
* Behavior

Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). Bicycles also have state (gear, pedal, current speed) and behavior (changing gear, changing pedal cadence, applying brakes). Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of Object-oriented programming.

Software objects are conceptually similar to real-world objects: they too consist of state and related behavior. An object stores its state in fields (variables in some Programming languages) and exposes its behavior through methods (functions in some Programming languages). Methods operate on an object's internal state and serve as the primary mechanism for object-to-object communication.

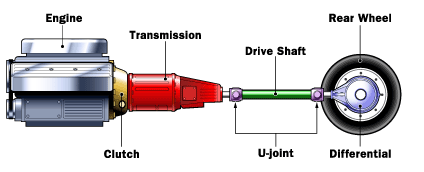
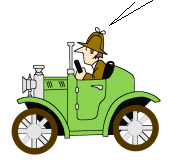
## Abstraction

Abstraction is the process of reducing the information content of a concept, typically in order to retain only the information which is relevant for a particular purpose. Abstraction manages the complexity of the program. For example, abstracting a leather soccer ball to a ball retains only the information on general ball attributes and behaviour.

Abstraction has four major attributes:

* Coherence
* Conciseness
* Identity
* Completeness

These entire features make ‘Abstraction’ an essential element for Object-Oriented Programming. In a sense, when someone works on a computer, it is not necessary that he should know the working of each and every part of the computer. Even without the hardware knowledge, he can e-mail, type or do other jobs on the computer.



**Do you think about how the different parts of your car engine work while driving?**

Fig. 4: Abstraction diagram

Similarly, people do not think of a computer as a unit made up of hundreds of cards and chips, but as a well-defined object with its own unique behavior. When you interact with objects in the world, often you are only concerned with a subset of their properties. Without this ability to abstract or filter out the extra properties of objects, you would find it hard to process all the code and concentrate on the task at hand. As a result of abstraction, when two different people interact with the same object, they often deal with a different subset of attributes. Fig. 4 gives you a brief idea about abstraction. When you are driving a car, for example, you need to know the speed of the car and the direction it is going. Because the car is automatic, you do not need to know the RPMs of the engine. Hence, you filter this information out. On the other hand, this information would be critical to a racecar driver, who would not filter it out.

While constructing objects in OOP applications, it is important to incorporate the concept of abstraction. If you were building a shipping application, you would construct a product object with attributes, such as size and weight. The color of the item would be extraneous information and filtered out. On the other hand, when constructing an order-entry application, the color could be important and would be included as an attribute of the product object.

Consider the simple “student” object. A real “student” has an identity, a family genealogy, a medical history, a genetic profile, a credit record, a set of talents, and many more. These are the student's attributes. Similarly, there is a rich set of actions or behaviors of which the student is capable (singSong, doDance, getSick, increaseCreditLimit, payBills, etc.). To know the genetic profile of the student you can abstract it out and avoid other extraneous information.

**Encapsulation**

Encapsulation is a process by which you can hide data fields and methods inside the object. It is needless to say that data fields and methods are the two main elements of an object in the Java programming language. Encapsulation draws the demarcation line between conventional structured Programming and object-oriented Programming. In strict object-oriented design, an object's data is always private to the object. Other parts of a program should never have direct access to that data. With encapsulation, you can control access to data, forcing programs to retrieve or modify data only through the object's interface.

How does this hiding of data differ from a structured-programming approach? After all, you can always hide data inside functions, just by making that data local to the function. A problem arises, however, when you want to make the data of one function available to other functions. The way to do this in a structured program is to make the data global to the program, which gives any function access to it. It seems that you could use another level of scope, one that would make your data global to the functions that need it, but still prevent other functions from gaining access. This is exactly what Encapsulation does. In an object, the encapsulated data members are global to the object's methods, yet they are local to the object. They are not global variables.

A well-established use of encapsulation in software design and development is the separation of an interface from an implementation. Interface is viewed as the discernible external aspect of the software that must be understood to manipulate the software. The implementation is viewed as the hidden intrinsic aspect of the software that is only relevant to the implementers.

### Inheritance

Inheritance enables you to create a class that is similar to a previously defined class. Despite being the replica, the new class possesses indigenous properties.



**Vehicle**

**Car**

**Bus**

**Truck**

Vehicle

Fig. 5: Inheritance diagram

The Fig. 5 can help you understand inheritance in an easy way. See how Car, Bus and Truck inherit some characteristics features from super class ‘Vehicle’. In other words it is inheritance by which they derive some property from their parent object.

Consider a car-simulation program. Suppose that you have a class for a regular car, but now you want to create a car that has a high-speed passing gear. In a traditional program, you might have to modify the existing code extensively and might introduce bugs into code that worked fine before you can make the necessary changes. To avoid these hassles, you use the object-oriented approach where you create a new class by inheritance. This new class inherits all the data and methods from the tested base class. (You can control the level of inheritance with the public, private, and protected keywords, which you will learn in the chapter on "Introducing Classes and Objects") Now, you only need to worry about testing the new code you added to the derived class.

### Polymorphism

The last major feature of object-oriented Programming is polymorphism. By using polymorphism, you can create new objects that perform the same functions as the base object but which perform one or more of these functions in a different way. For example, in Fig. 6, you may have a shape class that derived into three subclasses having three different functionality with a same method as given here draw()*.*

* One draws a circle on the screen.
* One draws triangle on the screen.
* Another draws square on the screen.

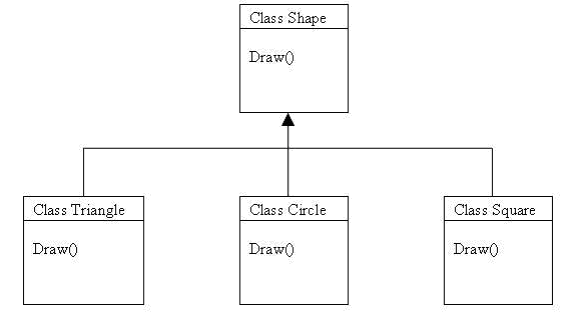


Fig. 6: Polymorphism diagram

All methods have the same name (such as draw()) but accomplish the drawing in a different way.

## Benefits of OOPS

Bundling code into individual software objects provides a number of benefits, including:

* **Modularity:** The source code for an object can be written and maintained independent of the source code for other objects. Once created, an object can be easily passed around inside the system.
* **Information hiding**: By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.
* **Code reusability**: If an object already exists (perhaps written by another software

developer), you can use that in your program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust, to run in your own code.

* **Pluggability and debugging ease**: If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is same as to fixing mechanical problems in the real world. If a bolt breaks, you replace it, not the entire machine.

### Example: Encapsulation, Inheritance, and Polymorphism

Now that you are comfortable with OOP concepts, let us look at these concepts in a general way. For example, you will extend the car metaphor that you read earlier in this chapter.

Let us take the car again as an object having several characteristics (direction, color, and speed) and several means (steering wheel, set color, and brakes) to act on those characteristics. In terms of constructing a class for a car object, you can think of direction, color, and speed as the class's data fields and the steering wheel, setting color, and brakes as representing the class's methods.

The first step in creating an object is to define its class. For now, you will use code to create a Car class. Example 1 is the code to create a Car Class.

class Car

{

private String direction; //left, right etc.

private String color; // red, green etc.

private double speed; //40, 60.66 etc.

public void steer(){}

public void setColor(){}

public void pressBrake(){}

}

**Example 1:** The code for a Car Class

In this base Car class, a car is defined by its direction (which way its pointed), color (color of the car), and speed. These three data fields can be manipulated by the three methods steer(), setColor(), and pressBrake(). The steer() method changes the car's direction, whereas the pressBrake() change the car's speed.

The data fields and methods are all encapsulated inside the class. Moreover, the data fields are private to the class, meaning that they cannot be directly accessed from outside the class. Only three methods in the class can access the data fields. In short, Example 1 not only shows what a class might look like, it also shows how encapsulation works.

Now, suppose you want to create a new car that has a special passing gear. To do this, you can use OOP inheritance to derive a new class from the Car base class. Example 2 is the code for this new class.

class PassingCar extends Car

{

public void pass(){}

}

**Example 2:** Deriving a New Class Using Inheritance.

You may be surprised to see how small this new class is. It's small because it implicitly inherits all the methods from the Car base class. In other words, not only does the PassingCar class have a method called pass(), but it also has the the steer(), setColor(), and pressBrake() methods. The PassingCar class can use all these methods exactly as if they were explicitly defined in Example 2. This is an example of inheritance.

The last OOP concept that you will apply to the car classes is polymorphism. Suppose that you now decide that you want a new kind of car that has all the characteristics of a PassingCar, except that its passing gear is twice as fast as PassingCar's. You can solve this problem as shown in the following example.

class FastCar inherits from PassingCar

{

public void pass(){}

}

**Example 3:** Using Polymorphism to Create a Faster Car.

The FastCar class looks exactly like the original PassingCar class. However, rather than just inheriting the pass() method, it defines its own version. This new version makes the car move twice as fast as PassingCar's pass() method does (the code that actually implements each method is not shown). In this way, the FastCar class implements the same functionality as the PassingCar class, but it implements that functionality a little differently.

### Summary

In this chapter, you have learnt about the various concepts, terms and condition of object-oriented programming. By the end of this chapter you will also become familiar with OOP programming concepts.

* You know that an object is an entity with a defined behavior and characteristic defined by the object’s class.
* You can think of a class as a pattern or template from which to build objects.
* When you create an instance of a class, you create an object of that class type. The new object has all the functionality, data, and characteristics defined by the class.
* A fundamental and powerful feature of classes is the capability to build a new class from another class. This process is known as inheritance.
* Abstraction is the process of reducing the information content of a concept, typically in order to retain only information, which is relevant for a particular purpose.
* Encapsulation enables you to hide, inside the object, both the data fields and the methods that act on that data.
* By using polymorphism, you can create new objects that perform the same functions as the base object but which perform one or more of these functions in a different way.
* Finally, you should have a better understanding of what abstraction, inheritance, encapsulation and polymorphism are, and how they relate to a class.