3.1 An Introduction to 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. It simplifies software development and maintenance by providing some concepts:

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

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. An object contains an address and takes up some space in memory. Objects can communicate without knowing the details of each other's data or code. The only necessary thing is the type of message accepted and the type of response returned by the objects.

Example: A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.

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 doesn't consume any space.

Inheritance

When one object acquires all the properties and behaviors of a parent object, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

"Now Speak!"
"Quack"
"Woof"
"Meow"

Polymorphism

If *one task is performed in different ways*, it is known as polymorphism. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.

In Java, we use method overloading and method overriding to achieve polymorphism.

Another example can be to speak something; for example, a cat speaks meow, dog barks woof, etc.

Abstraction

Hiding internal details and showing functionality is known as abstraction. For example phone call, we don't know the internal processing.

In Java, we use abstract class and interface to achieve abstraction.

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.

A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

3.2 Using Predefined Classes

The Concept of Java Classes and Objects

Classes and objects are the two most essential Java concepts that every programmer must learn. Classes and
objects are closely related and work together. An object has behaviors and states, and is an instance of class. For
instance, a cat is an object—it's color and size are states, and its meowing and clawing furniture are behaviors. A
class models the object, a blueprint or template that describes the state or behavior supported by objects of that
type.

What Is a Class?

We can define a class as a container that stores the data members and methods together. These data members and
methods are common to all the objects present in a particular package.

Every class we use in Java consists of the following components, as described below:

Access Modifier

Object-oriented programming languages like Java provide the programmers with four types of access modifiers.

1. Public
2. Private
3. Protected
4. Default

These access modifiers specify the accessibility and users permissions of the methods and members of the class.

Class Name

This describes the name given to the class that the programmer decides on, according to the predefined naming conventions.

Body of the Class

The body of the class mainly includes executable statements.

Apart from these, a class may include keywords like "super" if you use a superclass, "implements" if you are inheriting members, methods, and instances from the same class, and
"interface" if you are inheriting any members, methods, and instances from a different class.

Type of Classes

In Java, we classify classes into two types:

- Built-in Classes
- User-defined Classes

Built-in Classes

Built-in classes are just the predefined classes that come along with the Java Development Kit (JDK). We also call
these built-in classes libraries. Some examples of built-in classes include:
java.lang.System
java.util.Date
java.util.ArrayList
java.lang.Thread

Lots of classes and methods are already predefined by the time you start writing your own code:

Some already written by other programmers in your team
Many predefined packages, classes, and methods come from the Java Library.
Library: collection of packages
Package: contains several classes
class: contains several methods
Method: a set of instructions
Using Predefined Classes and Methods
To use a method, you must know:

- Name of the class containing the method (like **Math**)
- Name of the package containing the class (like **java.lang**)
- Name of the method (like **pow**) and list of parameters

```
import java.lang.*; // imports package
Math.pow( 2, 3 ); // calls power method in class Math
Math.pow( x, y ); // another call
```

java.lang.String
String class will be the undisputed champion on any day by popularity and none will deny that. This is a final class and used to create / operate immutable string literals. It was available from JDK 1.0
java.lang.System
Usage of System depends on the type of project you work on. You may not be using it in your project but still it is one of the popular java classes around. This is a utility class and cannot be instantiated. Main uses of this class are access to standard input, output, environment variables, etc. Available since JDK 1.0
java.lang.Exception
Throwable is the super class of all Errors and Exceptions. All abnormal conditions that can be handled comes under Exception. NullPointerException is the most popular among all the exceptions. Exception is at top of hierarchy of all such exceptions. Available since JDK 1.0
java.util.ArrayList
An implementation of array data structure. This class implements List interface and is the most popular member or java collections framework. Difference between ArrayList and Vector is one popular topic among the beginners and frequently asked question in java interviews. It was introduced in JDK 1.2
java.util.HashMap
An implementation of a key-value pair data structure. This class implements Map interface. As similar to ArrayList vs Vector, we have HashMap vs Hashtable popular comparisons. This happens to be a popular collection class that acts as a container for

property-value pairs and works as a transport agent between multiple layers of an application. It was introduced in JDK 1.2

java.lang.Object

Great grandfather of all java classes. Every java class is a subclass of Object. It will be used often when we work on a platform/framework. It contains the important methods like equals, hashcode, clone, toString, etc. It is available from day one of java (JDK 1.0)

java.lang.Thread

A thread is a single sequence of execution, where multiple thread can co-exist and share resources. We can extend this Thread class and create our own threads. Using Runnable is also another option. Usage of this class depends on the domain of your application. It is not absolutely necessary to build a usual application. It was available from JDK
1.0

java.lang.Class

Class is a direct subclass of Object. There is no constructor in this class and their objects are loaded in JVM by classloaders. Most of us may not have used it directly but I think its an essential class. It is an important class in doing reflection. It is available from JDK 1.0

java.util.Date

This is used to work with date. Sometimes we feel that this class should have added more utility methods and we end up creating those. Every enterprise application we create has a date utility. Introduced in JDK 1.0 and later made huge changes in JDK1.1 by deprecating a whole lot of methods.

java.util.Iterator

This is an interface. It is very popular and came as a replacement for Enumeration. It is a simple to use convenience utility and works in sync with Iterable. It was introduced in JDK 1.2

Using predefined class name as Class or Variable name in Java

In Java, Using predefined class name as Class or Variable name is allowed. However, According to Java Specification Language(§3.9) the basic rule for naming in Java is that you cannot use a keyword as name of a class, name of a variable nor the name of a folder used for package.

Using any predefined class in Java won’t cause such compiler error as Java predefined classes are not keywords.

Following are some invalid variable declarations in Java:

boolean break = false; // not allowed as break is keyword
int boolean = 8; // not allowed as boolean is keyword

```
boolean goto = false; // not allowed as goto is keyword
String final = "hi"; // not allowed as final is keyword
```

Using predefined class name as User defined class name

Question : Can we have our class name as one of the predefined class name in our program?

Answer : Yes we can have it. Below is example of using **Number** as user defined class

```
// Number is predefined class name in java.lang package
// Note : java.lang package is included in every java
program by default
public class Number
{
        public static void main (String[] args)
        {
                System.out.println("It works");
        }
}
```

Output:

```
It works
```

Using String as User Defined Class:

```
// String is predefined class name in java.lang package
// Note : java.lang package is included in every java
program by default
public class String
{
        public static void main (String[] args)
        {
                System.out.println("I got confused");
        }
}
```

3.3 Defining Your Own Class

User-defined classes are rather self-explanatory. The name says it all. They are classes that the user defines and
manipulates in the real-time programming environment. User-defined classes are broken down into three types:

Concrete Class

Concrete class is just another standard class that the user defines and stores the methods and data members in.
Syntax:
```
class con{
	//class body;
}
```

Abstract Class

Abstract classes are similar to concrete classes, except that you need to define them using the "abstract" keyword.
If you wish to instantiate an abstract class, then it should include an abstract method in it. Otherwise, it can only be
inherited.
Syntax:
```
abstract class AbstClas{
	//method();
	abstract void demo();
}
```

Interfaces

Interfaces are similar to classes. The difference is that while class describes an object's attitudes and behaviors,
interfaces contain the behaviors a class implements. These interfaces will only include the signatures of the
method but not the actual method.
Syntax:
```
public interface demo{
	public void signature1();
	public void signature2();
	}
	public class demo2 implements demo{
		public void signature1(){
			//implementation;
		}
		public void signature2(){
			//implementation;
	}
}
```
Rules for Creating Classes

The following rules are mandatory when you're working with Java classes:
1. The keyword "class" must be used to declare a class
2. Every class name should start with an upper case character, and if you intend to include multiple words in
a class name, make sure you use the camel case
3. A Java project can contain any number of default classes but should not hold more than one public class
4. You should not use special characters when naming classes
5. You can implement multiple interfaces by writing their names in front of the class, separated by commas
6. You should expect a Java class to inherit only one parent class

Key Differences Between Java Classes and Objects

The key differences between a class and an object are:
Class:
A class is a blueprint for creating objects
A class is a logical entity
The keyword used is "class"
A class is designed or declared only once
The computer does not allocate memory when you declare a class
Objects:
An object is a copy of a class
An object is a physical entity
The keyword used is "new"
You can create any number of objects using one single class
The computer allocates memory when you declare a class

Create a Class

To create a class, use the keyword class:
Main.java
Create a class named "Main" with a variable x:

```
public class Main {
int x = 5;
}
```

3.4 Static Fields and Methods

Java static variable

If you declare any variable as static, it is known as a static variable.

- The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
- The static variable gets memory only once in the class area at the time of class loading.

Advantages of static variable
It makes your program **memory efficient** (i.e., it saves memory).

Example of static variable

```
//Java Program to illustrate the use of static variable which

//is shared with all objects.
class Counter2{
static int count=0;//will get memory only once and retain its value
 Counter2(){
count++;//incrementing the value of static variable
System.out.println(count);
}

public static void main(String args[]){
//creating objects
Counter2 c1=new Counter2();
Counter2 c2=new Counter2();
Counter2 c3=new Counter2();
}
}
```

Java static method
If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

Example of static method

```
    //Java Program to demonstrate the use of a static method.
    class Student{
        int rollno;
        String name;
        static String college = "ITS";
        //static method to change the value of static variable
        static void change(){
        college = "BBDIT";
        }
        //constructor to initialize the variable
        Student(int r, String n){
        rollno = r;
        name = n;
        }
        //method to display values
        void display(){System.out.println(rollno+" "+name+" "+college);}
    }
    //Test class to create and display the values of object
    public class TestStaticMethod{
       public static void main(String args[]){
       Student.change();//calling change method
       //creating objects
       Student s1 = new Student(111,"Karan");
       Student s2 = new Student(222,"Aryan");
       Student s3 = new Student(333,"Sonoo");
       //calling display method
       s1.display();
       s2.display();
       s3.display();
       }
    }
```

```
Output:111 Karan BBDIT
       222 Aryan BBDIT
       333 Sonoo BBDIT
```

Another example of a static method that performs a normal calculation

//Java Program to get the cube of a given number using the static method

```
    class Calculate{
     static int cube(int x){
     return x*x*x;
     }
     public static void main(String args[]){
     int result=Calculate.cube(5);
     System.out.println(result);
     }
    }
```

Output:125

Restrictions for the static method

There are two main restrictions for the static method. They are:

1. The static method can not use non static data member or call non-static method directly.
2. this and super cannot be used in static context.

3.5 Method Parameters

A method is a block of code which only runs when it is called.

You can pass data, known as parameters, into a method.

Methods are used to perform certain actions, and they are also known as functions.

Why use methods? To reuse code: define the code once, and use it many times.

Create a Method

A method must be declared within a class. It is defined with the name of the method, followed by parentheses ().

Java provides some pre-defined methods, such as System.out.println(), but you can also create your own methods
to perform certain actions:

Example

Create a method inside Main:

```
public class Main {
    static void myMethod() {
        // code to be executed
    }
}
```

Example Explained

myMethod() is the name of the method

static means that the method belongs to the Main class and not an object of the Main class.

void means that this method does not have a return value. You will learn more about return values later in this chapter

Call a Method

To call a method in Java, write the method's name followed by two parentheses () and a semicolon;

In the following example, myMethod() is used to print a text (the action), when it is called:

Example

Inside main, call the myMethod() method:

```
public class Main {
        static void myMethod() {
                System.out.println("I just got executed!");
        }
        public static void main(String[] args) {
                myMethod();
        }
}
// Outputs "I just got executed!"
```

A method can also be called multiple times:

Example

```
public class Main {
static void myMethod() {
        System.out.println("I just got executed!");
}
public static void main(String[] args) {
        myMethod();
        myMethod();
        myMethod();
}
}
// I just got executed!
// I just got executed!
// I just got executed!
```

Java Method Parameters

Parameters and Arguments

Information can be passed to methods as parameter. Parameters act as variables inside the method.
Parameters are specified after the method name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.
The following example has a method that takes a String called fname as parameter. When the method is called, we pass along a first name, which is used inside the method to print the full name:

Example

```
public class Main {
static void myMethod(String fname) {
System.out.println(fname + " Refsnes");
}
public static void main(String[] args) {
        myMethod("Liam");
        myMethod("Jenny");
        myMethod("Anja");
}
}
// Liam Refsnes
// Jenny Refsnes
// Anja Refsnes
```

Multiple Parameters

You can have as many parameters as you like:

Example

```
public class Main {
static void myMethod(String fname, int age) {
System.out.println(fname + " is " + age);
}
public static void main(String[] args) {
myMethod("Liam", 5);
myMethod("Jenny", 8);
myMethod("Anja", 31);
}
}
// Liam is 5
// Jenny is 8
// Anja is 31
```

Return Values

The void keyword, used in the examples above, indicates that the method should not return a value. If you want the method to return a value, you can use a primitive data type (such as int, char, etc.) instead of void, and use the return keyword inside the method:

Example

```
public class Main {
        static int myMethod(int x) {
        return 5 + x;
}
public static void main(String[] args) {
        System.out.println(myMethod(3));
}
}
// Outputs 8 (5 + 3)
```

This example returns the sum of a method's two parameters:

Example

```
public class Main {
static int myMethod(int x, int y) {
return x + y;
}
public static void main(String[] args) {
System.out.println(myMethod(5, 3));
}
}
// Outputs 8 (5 + 3)
```

You can also store the result in a variable (recommended, as it is easier to read and maintain):

Example

```
public class Main {
static int myMethod(int x, int y) {
return x + y;
}
public static void main(String[] args) {
int z = myMethod(5, 3);
System.out.println(z);
}
}
// Outputs 8 (5 + 3)
```

A Method with If...Else

It is common to use if...else statements inside methods:

Example

```
public class Main {
        // Create a checkAge() method with an integer variable called age
        static void checkAge(int age) {
        // If age is less than 18, print "access denied"
        if (age < 18) {
        System.out.println("Access denied - You are not old enough!");
        // If age is greater than, or equal to, 18, print "access granted"
        } else {
        System.out.println("Access granted - You are old enough!");
        }
        }
        public static void main(String[] args) {
        checkAge(20); // Call the checkAge method and pass along an age of 20
        }
}
// Outputs "Access granted - You are old enough!"
```

3.6 Object Construction

Create an Object

In Java, an object is created from a class. We have already created the class named MyClass, so now we can use this
to create objects.
To create an object of MyClass, specify the class name, followed by the object name, and use the keyword new:

Example

Create an object called "myObj" and print the value of x:

```
public class Main {
int x = 5;
public static void main(String[] args) {
Main myObj = new Main();
System.out.println(myObj.x);
}
}
```

Multiple Objects

You can create multiple objects of one class:

Example

Create two objects of Main:

```
public class Main {
int x = 5;
public static void main(String[] args) {
Main myObj1 = new Main(); // Object 1
Main myObj2 = new Main(); // Object 2
System.out.println(myObj1.x);
System.out.println(myObj2.x);
}
}
```

Using Multiple Classes

You can also create an object of a class and access it in another class. This is often used for better organization of
classes (one class has all the attributes and methods, while the other class holds the main() method (code to be
executed)).

Remember that the name of the java file should match the class name. In this example, we have created two files in
the same directory/folder:

Main.java
Second.java

Main.java

```
public class Main {
int x = 5;
}
```

Second.java

```
class Second {
public static void main(String[] args) {
```

```
Main myObj = new Main();
System.out.println(myObj.x);
}
}
```

When both files have been compiled:

```
C:\Users\Your Name>javac Main.java
C:\Users\Your Name>javac Second.java
```

Run the Second.java file:

```
C:\Users\Your Name>java Second
```

And the output will be:

```
5
```

Java Class Attributes

In the previous chapter, we used the term "variable" for x in the example (as shown below). It is actually
an **attribute** of the class. Or you could say that class attributes are variables within a class:

Example

Create a class called "Main" with two attributes: x and y:

```
public class Main {
int x = 5;
int y = 3;
}
```

Accessing Attributes

You can access attributes by creating an object of the class, and by using the dot syntax (.):
The following example will create an object of the Main class, with the name myObj. We use the x attribute on the
object to print its value:

Example

Create an object called "myObj" and print the value of x:

```
public class Main {
int x = 5;
public static void main(String[] args) {
Main myObj = new Main();
System.out.println(myObj.x);
}
}
```

Modify Attributes

You can also modify attribute values:

Example

Set the value of x to 40:

```
public class Main {
int x;
public static void main(String[] args) {
Main myObj = new Main();
myObj.x = 40;
System.out.println(myObj.x);
}
}
```

Or override existing values:

Example

Change the value of x to 25:

```
public class Main {
int x = 10;
public static void main(String[] args) {
Main myObj = new Main();
myObj.x = 25; // x is now 25
System.out.println(myObj.x);
}
}
```

If you don't want the ability to override existing values, declare the attribute as final:

Example

```
public class Main {
final int x = 10;
public static void main(String[] args) {
Main myObj = new Main();
myObj.x = 25; // will generate an error: cannot assign a value to a final variable
System.out.println(myObj.x);
}
}
```

Multiple Objects

If you create multiple objects of one class, you can change the attribute values in one object, without affecting the
attribute values in the other:

Example

Change the value of x to 25 in myObj2, and leave x in myObj1 unchanged:

```
public class Main {
int x = 5;
public static void main(String[] args) {
Main myObj1 = new Main(); // Object 1
Main myObj2 = new Main(); // Object 2
myObj2.x = 25;
System.out.println(myObj1.x); // Outputs 5
System.out.println(myObj2.x); // Outputs 25
}
}
```

Multiple Attributes

You can specify as many attributes as you want:

Example

```
public class Main {
String fname = "John";
String lname = "Doe";
int age = 24;
public static void main(String[] args) {
Main myObj = new Main();
System.out.println("Name: " + myObj.fname + " " + myObj.lname);
System.out.println("Age: " + myObj.age);
}
}
```

3.7 Packages

A package in Java is used to group related classes. Think of it as a folder in a file directory. We use packages to avoid name conflicts, and to write a better maintainable code. Packages are divided into two categories:

Built-in Packages (packages from the Java API)

User-defined Packages (create your own packages)

Built-in Packages

The Java API is a library of prewritten classes, that are free to use, included in the Java Development Environment. The library contains components for managing input, database programming, and much much more.

The library is divided into **packages** and **classes**. Meaning you can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package.

To use a class or a package from the library, you need to use the import keyword:

Syntax

```
import package.name.Class; // Import a single class
import package.name.*; // Import the whole package
```

Import a Class

If you find a class you want to use, for example, the Scanner class, **which is used to get user input**, write the following code:

Example

```
import java.util.Scanner;
```

In the example above, java.util is a package, while Scanner is a class of the java.util package.
To use the Scanner class, create an object of the class and use any of the available methods found inthe Scanner class documentation. In our example, we will use the nextLine() method, which is used to read acomplete line:

Example

Using the Scanner class to get user input:

```
import java.util.Scanner;
class MyClass {
        public static void main(String[] args) {
                Scanner myObj = new Scanner(System.in);
                System.out.println("Enter username");
                String userName = myObj.nextLine();
                System.out.println("Username is: " + userName);
}
}
```

Import a Package

There are many packages to choose from. In the previous example, we used the Scanner class from the java.util package. This package also contains date and time facilities, random-number generator and other
utility classes.
To import a whole package, end the sentence with an asterisk sign (*). The following example will import ALL the
classes in the java.util package:

Example

```
import java.util.*;
```

User-defined Packages

To create your own package, you need to understand that Java uses a file system directory to store them. Just like
folders on your computer:

Example

```
└── root
└── mypack
└── MyPackageClass.java
```

To create a package, use the package keyword:

MyPackageClass.java

```
package mypack;
class MyPackageClass {
        public static void main(String[] args) {
                System.out.println("This is my package!");
        }
}
```

Save the file as **MyPackageClass.java**, and compile it:

```
C:\Users\Your Name>javac MyPackageClass.java
```

- Then compile the package:
- C:\Users\Your Name>javac -d . MyPackageClass.java
- This forces the compiler to create the "mypack" package.
- The -d keyword specifies the destination for where to save the class file. You can use any directory name, like c:/user (windows), or, if you want to keep the package within the same directory, you can use the dot sign ".", like in the example above.

Note: The package name should be written in lower case to avoid conflict with class names.

When we compiled the package in the example above, a new folder was created, called "mypack".

To run the **MyPackageClass.java** file, write the following:

```
C:\Users\Your Name>java mypack.MyPackageClass
```

The output will be:

```
This is my package!
```