CHP-3
Java Basic Syntax

When we consider a Java program, it can be defined as a collection of objects that communicate via invoking each other's methods. Let us now briefly look into what do class, object, methods and instance variables mean.

- **Object -** Objects have states and behaviors. Example: A dog has states-color, name, breed as well as behaviors -wagging, barking, eating. An object is an instance of a class.
- **Class -** A class can be defined as a template/blue print that describes the behaviors/states that object of its type support.
- **Methods -** A method is basically a behavior. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.
- **Instance Variables -** Each object has its unique set of instance variables. An object's state is created by the values assigned to these instance variables.

First Java Program:

Let us look at a simple code that would print the words *Hello World*.

```
public class MyFirstJavaProgram{
/* This is my first java program.
* This will print 'Hello World' as the output
*/
public static void main(String[]args){
System.out.println("Hello World");// prints Hello World
}
}
```

Let's look at how to save the file, compile and run the program. Please follow the steps given below:

- Open notepad and add the code as above.
- Save the file as: MyFirstJavaProgram.java.
- Open a command prompt window and go o the directory where you saved the class. Assume it's C:\.
- Type ' javac MyFirstJavaProgram.java ' and press enter to compile your code. If there are no errors in your code, the command prompt will take you to the next line(Assumption : The path variable is set).

- Now, type ' java MyFirstJavaProgram ' to run your program.
- You will be able to see ' Hello World ' printed on the window.

```
C :> javac MyFirstJavaProgram.java
C :> java MyFirstJavaProgram
HelloWorld
```

Basic Syntax:

About Java programs, it is very important to keep in mind the following points.

- **Case Sensitivity -** Java is case sensitive, which means identifier **Hello** and **hello** would have different

meaning in Java.

- **Class Names -** For all class names, the first letter should be in Upper Case.

If several words are used to form a name of the class, each inner word's first letter should be in Upper Case.
Example *class MyFirstJavaClass*

- **Method Names -** All method names should start with a Lower Case letter.

If several words are used to form the name of the method, then each inner word's first letter should be in Upper Case.
Example *public void myMethodName()*

- **Program File Name -** Name of the program file should exactly match the class name.

When saving the file, you should save it using the class name (Remember Java is case sensitive) and append '.java' to the end of the name (if the file name and the class name do not match your program will not compile).
Example : Assume 'MyFirstJavaProgram' is the class name, then the file should be saved as*'MyFirstJavaProgram.java'*

- **public static void main(String args[]) -** Java program processing starts from the main() method, which is a mandatory part of every Java program.

Java Identifiers:

All Java components require names. Names used for classes, variables and methods are called identifiers.
In Java, there are several points to remember about identifiers. They are as follows:

- All identifiers should begin with a letter (A to Z or a to z), currency character ($) or an underscore (_).
- After the first character, identifiers can have any combination of characters.
- A keyword cannot be used as an identifier.
- Most importantly identifiers are case sensitive.
- Examples of legal identifiers:age, $salary, _value, __1_value
- Examples of illegal identifiers: 123abc, -salary

Java Modifiers:

Like other languages, it is possible to modify classes, methods, etc., by using modifiers. There are two categories of modifiers:

- **Access Modifiers:** default, public, protected, private
- **Non-access Modifiers:** final, abstract, strictfp

We will be looking into more details about modifiers in the next section.

Java Variables:

We would see following type of variables in Java:

- Local Variables
- Class Variables (Static Variables)
- Instance Variables (Non-static variables)

Java Arrays:

Arrays are objects that store multiple variables of the same type. However, an array itself is an object on the heap.
We will look into how to declare, construct and initialize in the upcoming chapters.

Java Enums:

Enums were introduced in java 5.0. Enums restrict a variable to have one of only a few predefined values. The values in this enumerated list are called enums.
With the use of enums, it is possible to reduce the number of bugs in your code.
For example, if we consider an application for a fresh juice shop, it would be possible to restrict the glass size to small, medium and large. This would make sure that it would not allow anyone to order any size other than the small, medium or large.

Example:

```
Class FreshJuice{
e
num FreshJuiceSize{ SMALL, MEDUIM, LARGE }
FreshJuiceSize size;
}
p
```

```
ublic class FreshJuiceTest{
public static void main(String args[]){
FreshJuice juice =new FreshJuice();
juice.size =FreshJuice.FreshJuiceSize.MEDUIM ;
}
}
```

Note: enums can be declared as their own or inside a class. Methods, variables, constructors can be defined inside enums as well.

Java Keywords:

The following list shows the reserved words in Java. These reserved words may not be used as constant or variable or any other identifier names.

abstract assert boolean break
byte case catch char
class const continue default
do double else enum
extends final finally float
for goto if implements
import instanceof int interface
long native new package
private protected public return
short static strictfp super
switch synchronized this throw
throws transient try void
volatile while

Comments in Java

Java supports single-line and multi-line comments very similar to c and c++. All characters available inside any comment are ignored by Java compiler.

```
public class MyFirstJavaProgram{
/* This is my first java program.
* This will print 'Hello World' as the output
* This is an example of multi-line comments.
*/
public static void main(String[]args){
// This is an example of single line comment
/* This is also an example of single line comment. */
System.out.println("Hello World");
}
}
```

Using Blank Lines:

A line containing only whitespace, possibly with a comment, is known as a blank line, and Java totally ignores it.

Inheritance:

Java classes can be derived from classes. Basically, if you need to create a new class and here is already a class that has some of the code you require, then it is possible to derive your new class from the already existing code.

This concept allows you to reuse the fields and methods of the existing class without having to rewrite the code in a new class. In this scenario, the existing class is called the superclass and the derived class is called the subclass.

Interfaces:

In Java language, an interface can be defined as a contract between objects on how to communicate with each other. Interfaces play a vital role when it comes to the concept of inheritance.

An interface defines the methods, a deriving class(subclass) should use. But the implementation of the methods is totally up to the subclass.

What is Next?

The next section explains about Objects and classes in Java programming. At the end of the session, you will be able to get a clear picture as to what are objects and what are classes in Java.

CHP-4
Java Object & Classes

Java is an Object-Oriented Language. As a language that has the Object Oriented feature, Java supports the following fundamental concepts:

- Polymorphism
- Inheritance
- Encapsulation
- Abstraction
- Classes
- Objects
- Instance
- Method
- Message Parsing

In this chapter, we will look into the concepts Classes and Objects.

- **Object -** Objects have states and behaviors. Example: A dog has states-color, name, breed as well as behaviors -wagging, barking, eating. An object is an instance of a class.
- **Class -** A class can be defined as a template/blue print that describes the behaviors/states that object of its type support.

Objects in Java:

Let us now look deep into what are objects. If we consider the real-world we can find many objects around us, Cars, Dogs, Humans, etc. All these objects have a state and behavior.
If we consider a dog, then its state is - name, breed, color, and the behavior is - barking, wagging, running
If you compare the software object with a real world object, they have very similar characteristics.
Software objects also have a state and behavior. A software object's state is stored in fields and behavior is shown via methods.
So in software development, methods operate on the internal state of an object and the object-to-object communication is done via methods.

CHAPTER

TUTORIALS POINT Simply Easy Learning

Classes in Java:

A class is a blue print from which individual objects are created.
A sample of a class is given below:

```
public class Dog{
String breed;
int age;
```

```
String color;
void barking(){
}
void hungry(){
}
void sleeping(){
}
}
```

A class can contain any of the following variable types.

- **Local variables:** Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
- **Instance variables:** Instance variables are variables within a class but outside any method. These variables are instantiated when the class is loaded. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
- **Class variables:** Class variables are variables declared within a class, outside any method, with the static keyword.

A class can have any number of methods to access the value of various kinds of methods. In the above example, barking(), hungry() and sleeping() are methods.

Below mentioned are some of the important topics that need to be discussed when looking into classes of the Java Language.

Constructors:

When discussing about classes, one of the most important subtopic would be constructors. Every class has a constructor. If we do not explicitly write a constructor for a class the Java compiler builds a default constructor for that class.

Each time a new object is created, at least one constructor will be invoked. The main rule of constructors is that they should have the same name as the class. A class can have more than one constructor.

Example of a constructor is given below:

```
public class Puppy{
public Puppy(){
}
public Puppy(String name){
// This constructor has one parameter, name.
}
}
```


Java also supports Singleton Classes where you would be able to create only one instance of a class.

Singleton Classes

The Singleton's purpose is to control object creation, limiting the number of objects to one only. Since there is only one Singleton instance, any instance fields of a Singleton will occur only once per class, just like static fields. Singletons often control access to resources such as database connections or sockets.

For example, if you have a license for only one connection for your database or your JDBC driver has trouble with multithreading, the Singleton makes sure that only one connection is made or that only one thread can access the connection at a time.

Implementing Singletons:

Example 1:

The easiest implementation consists of a private constructor and a field to hold its result, and a static accessor method with a name like getInstance().

The private field can be assigned from within a static initializer block or, more simply, using an initializer. The getInstance( ) method (which must be public) then simply returns this instance:

```
// File Name: Singleton.java
public class Singleton{
private static Singleton singleton =new Singleton();
/* A private Constructor prevents any other
* class from instantiating.
*/
private Singleton(){}
```

```
/* Static 'instance' method */
public static Singleton getInstance(){
return singleton;
}
/* Other methods protected by singleton-ness */
protected static void demoMethod(){
System.out.println("demoMethod for singleton");
}
}
// File Name: SingletonDemo.java
public lassSingletonDemo{
public staticvoid main(String[] args){
Singleton tmp =Singleton.getInstance();
tmp.demoMethod();
}
}
```

This would produce the following result:

```
demoMethod for singleton
```


Example 2:

Following implementation shows a classic Singleton design pattern:

```
public class ClassicSingleton{
private static ClassicSingleton instance =null;
protected ClassicSingleton(){
// Exists only to defeat instantiation.
}
public static ClassicSingleton getInstance(){
if(instance ==null){
instance =new ClassicSingleton();
}
return instance;
}
}
```

The ClassicSingleton class maintains a static reference to the lone singleton instance and returns that reference from the static getInstance() method.

Here ClassicSingleton class employs a technique known as lazy instantiation to create the singleton; as a result, the singleton instance is not created until the getInstance() method is called for the first time. This technique ensures that singleton instances are created only when needed.

Creating an Object:

As mentioned previously, a class provides the blueprints for objects. So basically an object is created from a class. In Java the new keyword is used to create new objects.

There are three steps when creating an object from a class:

- **Declaration:** A variable declaration with a variable name with an object type.
- **Instantiation:** The 'new' keyword is used to create the object.
- **Initialization:** The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

Example of creating an object is given below:

```
public class Puppy{
public Puppy(String name){
// This constructor has one parameter, name.
System.out.println("Passed Name is :"+ name );
}
public static void main(String[]args){
// Following statement would create an object myPuppy
Puppy myPuppy =new Puppy("tommy");
}
}
```

If we compile and run the above program, then it would produce the following result:

```
PassedNameis:tommy
```

Accessing Instance Variables and Methods:

Instance variables and methods are accessed via created objects. To access an instance variable the fully qualified path should be as follows:


```
/* First create an object */
ObjectReference = new Constructor();
/* Now call a variable as follows */
ObjectReference.variableName;
/* Now you can call a class method as follows */
ObjectReference.MethodName();
```

Example:

This example explains how to access instance variables and methods of a class:

```
public class Puppy{
int puppyAge;
public Puppy(String name){
// This constructor has one parameter, name.
System.out.println("Passed Name is :"+ name );
}
public void setAge(int age ){
puppyAge = age;
}
public int getAge(){
System.out.println("Puppy's age is :"+ puppyAge );
return puppyAge;
}
public static void main(String[]args){
/* Object creation */
Puppy myPuppy =newPuppy("tommy");
/* Call class method to set puppy's age */
myPuppy.setAge(2);
/* Call another class method to get puppy's age */
myPuppy.getAge();
/* You can access instance variable as follows as well */
System.out.println("Variable Value :"+ myPuppy.puppyAge );
}
}
```

If we compile and run the above program, then it would produce the following result:

```
PassedName is:tommy
Puppy's age is :2
Variable Value :2
```

Source file declaration rules:

As the last part of this section, let's now look into the source file declaration rules. These rules are essential when declaring classes, *import* statements and *package* statements in a source file.

- There can be only one public class per source file.
- A source file can have multiple non public classes.

- The public class name should be the name of the source file as well which should be appended by **.java** at the end. For example : The class name is . *public class Employee{}* Then the source file should be as Employee.java.
- If the class is defined inside a package, then the package statement should be the first statement in the source file.
- If import statements are present then they must be written between the package statement and the class declaration. If there are no package statements then the import statement should be the first line in the source file.
- Import and package statements will imply to all the classes present in the source file. It is not possible to

declare different import and/or package statements to different classes in the source file.
Classes have several access levels and there are different types of classes; abstract classes, final classes, etc. I will be explaining about all these in the access modifiers chapter.
Apart from the above mentioned types of classes, Java also has some special classes called Inner classes and Anonymous classes.

Java Package:

In simple, it is a way of categorizing the classes and interfaces. When developing applications in Java, hundreds of classes and interfaces will be written, therefore categorizing these classes is a must as well as makes life much easier.

Import statements:

In Java if a fully qualified name, which includes the package and the class name, is given, then the compiler can easily locate the source code or classes. Import statement is a way of giving the proper location for the compiler to find that particular class.
For example,the following line would ask compiler to load all the classes available in directory java_installation/java/io

```
import java.io.*;
```

A Simple Case Study:

For our case study, we will be creating two classes. They are Employee and EmployeeTest.
First open notepad and add the following code. Remember this is the Employee class and the class is a public class. Now, save this source file with the name Employee.java.
The Employee class has four instance variables name, age, designation and salary. The class has one explicitly defined constructor, which takes a parameter.

```
import java.io.*;
public class Employee{
String name;
int age;
String designation;
double salary;
// This is the constructor of the class Employee
public Employee(String name){
```



```
this.name = name;
}
// Assign the age of the Employee to the variable age.
public void empAge(int empAge){
age = empAge;
}
/* Assign the designation to the variable designation.*/
public void empDesignation(String empDesig){
designation = empDesig;
}
/* Assign the salary to the variable salary.*/
public void empSalary(double empSalary){
salary = empSalary;
}
/* Print the Employee details */
public void printEmployee(){
System.out.println("Name:"+ name );
System.out.println("Age:"+ age );
System.out.println("Designation:"+ designation );
System.out.println("Salary:"+ salary);
}
}
```

As mentioned previously in this tutorial, processing starts from the main method. Therefore in-order for us to run this Employee class there should be main method and objects should be created. We will be creating a separate class for these tasks.
Given below is the *EmployeeTest* class, which creates two instances of the class Employee and invokes the methods for each object to assign values for each variable.

Save the following code in EmployeeTest.java file

```
import java.io.*;
publicclassEmployeeTest{
publicstaticvoid main(String args[]){
/* Create two objects using constructor */
Employee empOne =newEmployee("James Smith");
Employee empTwo =newEmployee("Mary Anne");
// Invoking methods for each object created
empOne.empAge(26);
empOne.empDesignation("Senior Software Engineer");
empOne.empSalary(1000);
empOne.printEmployee();
empTwo.empAge(21);
empTwo.empDesignation("Software Engineer");
empTwo.empSalary(500);
empTwo.printEmployee();
}
}
```

Now, compile both the classes and then run *EmployeeTest* to see the result as follows:

```
C :> javac Employee.java
C :> vi EmployeeTest.java
C :> javac EmployeeTest.java
C :> java EmployeeTest
Name:JamesSmith
```



```
Age:26
Designation:SeniorSoftwareEngineer
Salary:1000.0
Name:MaryAnne
Age:21
Designation:SoftwareEngineer
Salary:500.0
```

What is Next?

Next session will discuss basic data types in Java and how they can be used when developing Java applications.