Lecture 3

Object Oriented Programming with Java

Basic Syntax of a Class

```
class class-name {
        public static void main(String args[]) {
                statement1;
                statement2;
                    …
                    …
        }
}
```

First.java

```
→  class First {


→       public static void main(String args[]) {
→               System.out.println("Hello World");
→       }
→  }
```

Compiling & Running the Program

- → **Compiling:** is the process of translating source code written in a particular programming language into computer-readable machine code that can be executed.
- → $ javac First.java
- → This command will produce a file 'First.class', which is used for running the program with the command 'java'.

- → **Running**: is the process of executing program on a computer.
- → $ java First

Printing on Screen

- → System.out.println(“Hello World”); – outputs the string “Hello World” followed by a new line on the screen.
- → System.out.print(“Hello World”); - outputs the string “Hello World” on the screen. This string is not followed by a new line.
- → Some Escape Sequence –
- → \n – stands for new line character
- → \t – stands for tab character

Comments

→ “//”: Anything written after // is not executed

→ Multiline comment /* … */

Variables

- A variables can be considered as a name given to the location in memory where values are stored.
  - Variables can change its value.
- One syntax of variable declaration

  datatype variableName;
- Local, Instance , Static variables
- Variable Name? Rules?

```
class First {
    private int instance_var;  //instance variable
    public static String name = "UPES";  //static variable
        public static void main(String args[]) {
            System.out.println("Hello World");
            int local_var;
        }
}
```

```
class  Demo {
  public static String name = “UPES";          //Declared static variable: can be accessed without object creation
            public static void main (String[] args) {
            System.out.println(“The static Name is : "+Demo.name);
     }
}
```

Variable Naming Convention

- → Characters 'a' through 'z', 'A' through 'Z', '0' through '9', character '_', and character '$' are used
- → A name can't contain space character.
- → Do not start with a digit.
- → A Variable name can be of any length.
- → Upper and lower case count as different characters. So SUM and Sum are different names.
- → A name can not be a reserved word.
  - → Reserved word: has a predefined meaning in Java, e.g., int, double, true, etc.
- → A name must be used once for a given part of the program.

→ int 1name; // cannot start with digit

→ int number = 0; //Correct

→ char double; //double is a reserve word

→ double i,j,k   // must end with “;”

Variable Declaration

- → Before using any name, it must be declared.
- → Needed only once in one program
- → Generally, done initially
- → Syntax

datatype name;

```
double total;      // stores the total value
int index;
int a,b , c, sum,   interest;
```

Types of Numbers

→ int

4 bytes

Whole numbers like 0, 575, -345 etc.

→ double

8 bytes

Numbers with decimal point like 12.453, 3.432, 0.0000002

Character

→ Characters

‘a’, ‘A’, ‘c’ , ‘?’ , ‘3’ , ‘ ’

→ Enclosed in single quotes

→ variable declaration

```
char ch;
```

→ assignment

```
ch = ‘a’;
```

String

→ Strings are sequence of characters enclosed in double quotes

→ Declaration

```
String var;
```

→ Assignment

```
var = “ram”;
var = “this is a string variable with spaces”;
var = “a”; // single character can be stored
var = “”;  // empty string
```

→ The sequence of characters enclosed in double quotes, printed in println() are also strings.

→ E.g. System.out.println(“Hello World ! “);

Boolean Data type

→ This data type can store only two values; true and false.

→ Declaring a boolean variable is the same as declaring any other primitive data type like int, float, char.

```
boolean   response = false;     //Valid
boolean   answer =  true;       //Valid
boolean   answer =  9943;       //Invalid,
boolean   response = "false"; //  Invalid,
```

→ This is return type for relational & conditional operators (we will look them after few slides).

Expression

- An expression is a combination of constants (like 10, 20 etc.), operators (like +, *, etc.), few variables(stored in the memory) and parentheses (to preserve the order, like “(” and “)”) which is then used to calculate a value.
- Example:
  - x = 1;
  - y = 100 + x;

- Must follow the BODMAS rule of algebra
- Operators are Important part of any expression.

Operators

- → Unary Operator
- → Arithmetic Operator
- → Shift Operator
- → Relational Operator
- → Bitwise Operator
- → Logical Operator
- → Ternary Operator and
- → Assignment Operator

Unary Operator

- → The Java unary operators require only one operand.
- → Useful for:
  - → incrementing/decrementing a value by one
  - → negating an expression
  - → inverting the value of a Boolean
- → Example:

```
int x=10;
System.out.println(x++);        //10 (11)
System.out.println(++x);        //12
System.out.println(x--);        //12 (11)
System.out.println(--x);        //10
```

Arithmetic Operators

→ Arithmetic operators are used to perform addition, subtraction, multiplication, and division.

```
int a=10;
int b=5;
System.out.println(a+b);      //15
System.out.println(a-b);      //5
System.out.println(a*b);      //50
System.out.println(a/b);      //2
System.out.println(a%b);      //0
```

Shift Operators

→ Left Shift Operator: to shift all of the bits in a value to the left side of a specified number of times.

→ System.out.println(10<<2); //10*2^2=10*4=40

→ Right Shift Operator: to shift all of the bits in a value to the right side of a specified number of times.

→ System.out.println(10>>2); //10/2^2=10/4=2

Relational Operator

- → Relational Operators in Java are used to compare two variables for
  - → Equality (==),
  - → non-equality (!=),
  - → greater than (>), greater than or equal to (>=)
  - → less than (<), less than or equal to (<=)
- → Java relational operator always returns a boolean value => true or false.
- → The == and != operators can be used with any **primitive data types** as well as **objects**.
- → The <, >, <=, and >= can be used with primitive data types that can be represented in numbers. It will work with char, byte, short, int, etc. but not with boolean. These operators are not supported for objects.

Logical Operator

→ AND (&&), OR (||) ,

```
int a=10;

int b=5;

int c=20;

System.out.println(a<b&&a<c);          //        false   &&   true = false

System.out.println(a<b||a<c);          //        false   ||    true = true
```

Bitwise Operator

→ Bitwise OR (|)

→ Bitwise AND (&)

→ Bitwise XOR (^)

→ Bitwise Complement (~)

→ These are performed bit by bit. For example:

Bitwise OR Operation of 12(1100 in binary) and 25 (11001 in binary)

```
  0001100
| 0011001
_____________
  0011101 = 29 ( Decimal Value)
```

Ternary Operator

→ Java Ternary operator is used as one line replacement for if-then-else statement

→ It is the only conditional operator which takes three operands.

```
public class OperatorExample{
public static void main(String args[]){
int a=2;
int b=5;
int min=(a<b)?a:b;
System.out.println(min);
}}
```

OUTPUT : 2

Assignment Operator

→ It is used to assign the value on its right to the operand on its left.

→ int x; // Declaration (Once)

x=20 // assignment

→ int x = 20; // assignment and declaration together

20 = x ; // not possible – compilation error

Left hand side is always a variable for assignment

→ Example:

```
public class OperatorExample{
public static void main(String args[]){
int a=10;
int b=20;
a+=4;                    //a=a+4 (a=10+4)
b-=4;                    //b=b-4 (b=20-4)
System.out.println(a);
System.out.println(b);
}}
```

Conditional statements and Loops

→ **Alternation**: if, if-else, switch

```
if (<boolean expression>){
//body starts
    <statement(s)>
//body ends
}
```

```
if(<boolean expression>){
                <statement(s)>
}
else{
                <statement(s)>
}
```

```
switch(expression){
        case value1:
            statements;
          break;
        case value2:
            statements;
          break;
……
        case valueN:
            statements;
          break;
        default:
            statements;
       }
```

→ **Loop**: for, while, do-while

```
while (Boolean expression){
        //statements, block of  statements
}

for (initialisation; boolean expression; update)
{
        // statements, block of  statements;
}
```

```
class Fibonacci {
    /** Print out the Fibonacci sequence for values < 50 */
    public static void main(String[] args) {
        int lo = 1;

        int hi = 1;

        System.out.println(lo);
        while (hi < 50) {
            System.out.println(hi);
            hi = lo + hi;       // new hi
            lo = hi - lo;       /* new lo is (sum - old lo)
                                   that is, the old hi */
        }
    }
}
```

println as an example of function overloading

→ Notice that the **println** method accepts an integer argument in the Fibonacci example, whereas it accepted a string argument in the HelloWorld (previous class) example.

→ The **println** method is an example of one of many methods that are overloaded so that they can accept arguments of different types.

→ The runtime system decides which method to actually invoke based on the number and types of arguments which is passed to it.

Named Constants

- → In Fibonacci program if we want to change the constant 50 to some other value, it would be very tedious task if there are many occurence of the constant 50 in various different locations.
- → Named constants is a constant that can be referred by a name.
  - → To make a named variable, a constant value, we use a keyword **final**
  - → Its value cannot be changed.
  - → E.g. static final int MAX = 50
  - → Static makes this named constant field **not** to be associated with any instance of the class.

More Examples

```
class Suit {
    final static int CLUBS    = 1;
    final static int DIAMONDS = 2;
    final static int HEARTS   = 3;
    final static int SPADES   = 4;
}
```

- → To refer to a static member of a class we use the name of the class followed by dot and the name of the member.
- → For above example, suits in a program would be accessed as Suit.HEARTS, Suit. SPADES, and so on, thus grouping all the suit names within the single name Suit.
- → Another example is: out is a static field of class System.

Classes and Objects

- Every object has a class that defines its data and behavior.
- Each class has three kinds of members:
  - Data variables associated with a class and its objects. Fields store results of computations performed by the class.
  - Methods contain the executable code of a class.
  - Classes and interfaces can be members of other classes or interfaces (we will see interfaces soon).

```
class Point {
        public double x, y;
    }
```

→ Members of a class can have various levels of visibility or accessibility.

→ The public declaration of x and y in the Point class means that any code with access to a Point object can read and modify those fields.

→ Other levels of accessibility limit member access to code in the class itself or to other related classes (private and protected).

Access Modifiers

```
public class Deck {
    public static final int DECK_SIZE = 52;
    private Card[] cards = new Card[DECK_SIZE];
    public void print() {
        for (int i = 0; i < cards.length; i++)
            System.out.println(cards[i]);
    }
    // ...
}
```

- → **Public:** so that anyone can find out how many cards are in a deck.
- → Cards field is declared private, which means that only the methods in the current class can access it.
  - → This prevents anyone from manipulating our cards directly.
  - → The modifiers public and private are access modifiers because they control who can access a class, interface, field, or method

Objects

- → Creating an object from a class definition is also known as **instantiation**; thus, objects are often called **instances**.
- → Newly created objects are allocated within an area of system memory known as the **heap**.
- → **Object References:**
  - → All objects are accessed via object references. Any variable that may appear to hold an object contains a reference to that object.
  - → The types of such variables are known as reference types
  - → Primitive types variables hold values of that type.
  - → If and object is just declared and not assigned any memory then object reference is **null**

- → Each Point object is unique and has its own copy of the x and y fields.
- → Changing x in object lowerLeft, does not affect the value of x in the object upperRight
- → The fields (Member/Data variables) in objects are known as **instance variables**, because there is a unique copy of the field in each object (instance) of the class.
- → When we use **new** to create an object, a special piece of code, known as a **constructor**, is invoked to perform any initialization the object might need.

```
Point lowerLeft = new Point();
Point upperRight = new Point();
Point middlePoint = new Point();
lowerLeft.x = 0.0;  lowerLeft.y = 0.0;
upperRight.x = 1280.0;  upperRight.y = 1024.0;
middlePoint.x = 640.0;  middlePoint.y = 512.0;
```

→ A constructor has the same name as the class that it constructs and is similar to a method, including being able to accept arguments.

→ If a constructor is not declared in the class, the compiler creates one that takes no arguments and does nothing.

→ When we say "new Point()“: a Point object would be allocated and because we passed in **no arguments**, the no-argument constructor be invoked to initialize it

Static or Class Fields

- We obtain class-specific fields by declaring them static, and they are therefore commonly called static fields.
- We sometimes need **Fields** that are shared among all objects of that class.
  - These shared variables are known as class variables; variables specific to the class as opposed to objects of the class.
- For example, a Point object to represent the origin might be common and we provide it as a static field in the Point class:

```
public static Point origin = new Point();
```

- If this declaration appears inside the declaration of the Point class, there will be exactly one piece of data called Point.origin that always refers to an object at (0.0, 0.0).
- This static field is there no matter how many Point objects are created, even if none are created.
- The values of x and y are zero because that is the default for numeric fields that are not explicitly initialized to a different value

The Garbage Collector

- → ***new*** is used to create an object,
- → To get rid of the object when we no longer want it, we can simply stop referring to it.
- → Unreferenced objects are automatically reclaimed by a *garbage collector*, which runs in the background and tracks object references.
- → When an object is no longer referenced, the garbage collector can remove it from the storage allocation heap, although it may defer actually doing so until a good/appropriate time

Methods and Parameters

- → Objects of the Point class are exposed to manipulation by any code that has a reference to a Point object IF its fields (variables) are declared public.
- → The object orientated design suggests for hiding the implementation of a class behind operations performed on its data.
- → Operations of a class are declared via its methods
  - → Methods: Instructions that operate on an object's data to obtain results.
  - → Hiding data behind methods so that it is inaccessible to other objects is the fundamental basis of **data encapsulation**

Invoking a Method

- → Well-designed classes usually hide their data so that it can be changed only by methods of that class.
- → To invoke a method, an object reference is provided to the target object and the method name, separated by a dot.
  - → Arguments are passed to the method as a comma-separated list of values enclosed in parentheses.
  - → Methods without arguments require empty parentheses.

```
public double distance(Point that) {
    double xdiff = x - that.x;
    double ydiff = y - that.y;
    return Math.sqrt(xdiff * xdiff + ydiff * ydiff);
}
```

→ The sqrt method of the Math library class calculates the square root of the sum of the squares of the differences between the two x and y coordinates of two different objects.

→ We will invoke **distance()** function this way:

double d = lowerLeft.distance(upperRight);

this Reference

- How the receiving object would know its own reference.
  - For example, the receiving object might want to add itself to a list of objects somewhere.
- An implicit reference named ***this*** is available to methods, and this is a reference to the current (receiving) object.
- Considering another method of Point named move, which sets the x and y fields to specified values:

```
public void move(double x, double y) {
        this.x = x;
        this.y = y;
}
```

Static or Class Methods

- → Similar to per-class static fields, we have per-class static methods, often known as class methods.
- → Class methods are declared using the static keyword and are therefore also known as static methods.
- → The implementation of distance in the previous example uses the static method Math.sqrt to calculate a square root.
- → A static method cannot directly access non-static members.
- → When a static method is invoked, there's no specific object for the method to operate on, and so no this reference.
- → In general, static methods perform class-related tasks and non-static methods perform object-related tasks.

Tomorrow Lab

→ First program: Hello world

→ Basic Java Programs, looking at if, if-else, Loops, etc.

Books and Online Resources

- The Java Programming Language $3^{rd}/4^{th}$ Edition, Ken Arnold, James Gosling, Pearson
- The Complete Reference Java 7th Edition, Herbert-Schild, TMH.
- Head First Java, Kathy Sierra, Bert Bates.

- Web Resources
  - https://www.youtube.com/watch?v=J_d1fJy90GY&list=PLbRMhDVUMngcx5xHChJ-f7ofxZI4JzuQR&index=1 (NPTEL Youtube Lectures Playlist IIT Kharagpur)
  - https://ocw.mit.edu/courses/6-092-introduction-to-programming-in-java-january-iap-2010/pages/lecture-notes/
  - https://introcs.cs.princeton.edu/java/lectures/