Chapter One

Introduction to Object Oriented Programming (OOP in Java)

What is Object-Oriented Programming?

Object Oriented Programming (OOP) is a different method of thinking about programming using object and classes. We **think separately** about **data** and how the **methods** interact with data in non-OOP. Object-Oriented Programming, however, forces us to think in terms of objects and the interaction between objects. An object is a self-contained entity that describes not only certain data, but also the methods to maintain the data.

An object-oriented approach identifies the keywords in the problem. The keywords would be the **object** in the implementation and the hierarchy defines the relationship between **these objects**. The term object is used here to describe a limited well-defined structure, **containing all the information about some entity - data type and methods to manipulate the data.**

Why do we use Object-Oriented Programming?

OOP enable us to model real-world problem through programs in more logical manner - Because objects are discrete entities, we can debug, modify and maintain them more easily - If our objects are thoughtfully designed, we can **reuse** much of our code than is the case with procedural programming. OOP is the most usable and maintainable programming due to the following basic features.

Basic Concept/features of OOP

Abstraction

Abstraction is the way of collecting relevant information (attribute and methods) from the existing problem.

- Abstraction: The act of identifying objects to model the problem domain.
- Classes are abstracted from concepts/Real world problem.
- This is the first step of identifying **classes** and **attributes** that will be used in your applications.
- Objects are instantiated from classes.

Encapsulation

A goal of OOP is to differentiate the use of an object from the implementation of that object. One method to accomplish this is through the usage of data binding/hiding. Data hiding enable us to completely encapsulate an object's data members. The data-binding paradigm does not allow non-member methods to access data members directly. It is a good programming practice to make our entire data members private while defining a good interface for the object.

Encapsulation is the grouping of related items into one unit.

- Attributes and behaviors /methods are encapsulated to create objects.
- Implementation details are hidden from the outside world.
- The packaging of operations and attributes representing state into an object type so that state is **accessible** or **modifiable only through the objects' interface.**

Inheritance

Inheritance is a technique for creating a new class (subclass) from an existing class (superclass) by adding more functionality to it. We say that the new class inherits all the functionality from the existing class.

- A subclass is derived from a superclass. Example: An `Employee` is a `Person`.
- The subclass inherits the attributes and behavior of the superclass.
- The subclass can override the behavior of the superclass.
- Inheritance supports code re-use.

Polymorphism

The term polymorphism is derived from a Greek term meaning **many form use of a single method**.

- A method can have many different forms of behavior.
- A single method may be defined upon more than one class and may take on different implementations in each of those classes.

Message passing

- Objects communicate by sending messages.
- Messages convey some form of information.
- An object requests another object to carry out an activity by sending it a message.
- Most messages pass arguments back and forth through the object using **dot operator**.

Classes

A class is **user-defined data type** that is used to implement an abstract object, giving us the capability to use OOP. A class includes members. A member can be either data known as a data member or a method, known as a member method.

Objects

An object is an example /instance of a class. An object includes all the data necessary to represent the item and the methods that manipulate the data. Objects can be uniquely identified by its name and it defines a state that is represented by the values of its attribute at a particular point of time.

Members of a class

A class has one or more variable types, called members. The two types of members are *data members* and *methods member*.

Data members: data members of a class are exactly variables.

Methods member: are methods/ functions defined within a class that act on the data members in the class.

Example:

```
class student {
        String Stud_name;  // data members
        String Stud_IDNo;
        int Age;
 public static void main(String[] args) { // Main function
      student stud;
      stud.Stud_name=”Abel”;
      stud.Age=24;
System.out.println(stud.Stud_name); //methods member
System.out.println(stud.Age);
System.out.println(“This is your  first java  code!”);
   } }
```

Assignment

1. What is Programming language, what is the use of programming languages? Give some examples of programming language and classify them as OOP and non-OOP language.
2. Discuss the advantage of object-oriented programming language (OOP) over non-OOP languages (Unstructured and Structured Programming language).

What is Java?

Java is a computer programming language created by Sun Microsystems. Java is used mainly on the Internet and uses a virtual machine which has been implemented in most browsers to translate Java code into a specific application on different computer system. With most programming languages, you either compile or interpret a program so that you can run it on your computer. The Java programming language is unusual in that a program is both compiled and interpreted. The most common Java programs are applications and applets. Applications are standalone programs. Applets are similar to applications, but they don't run standalone. Instead, applets adhere to a set of conventions that lets them run within a Java-compatible browser. Applets are web based applications.

The Java programming language is a high-level language that can be described as,

- Object oriented
- Portable
- Distributed
- Multithreaded

What is JDK?

- The **Java Development Kit** (**JDK**) is a Sun Microsystems product aimed at Java developers. The JDK has as its primary components a collection of programming tools, including:
- java – the loader for Java applications. This tool is an interpreter and can interpret the class files generated by the javac compiler. Now a single launcher is used for both development and deployment. The old deployment launcher, jre, no longer comes with Sun JDK, and instead it has been replaced by this new java loader.
- javac – the compiler, which converts source code into Java bytecode
- appletviewer – this tool can be used to run and debug Java applets without a web browser
- javadoc – the documentation generator, which automatically generates documentation from source code comments
- jar – the archiver, which packages related class libraries into a single JAR file. This tool also helps manage JAR files.
- javaws – the Java Web Start launcher for JNLP applications
- jdb – the debugger
- jstat – Java Virtual Machine statistics monitoring tool (experimental)

What does the "Java Virtual Machine (JVM)" mean?

The Java Virtual machine (JVM) is the application that executes a Java program and it is included in the Java package.

A JVM is distributed along with a set of standard class libraries that implement the Java application programming interface (API). Appropriate APIs bundled together form the Java Runtime Environment (JRE).

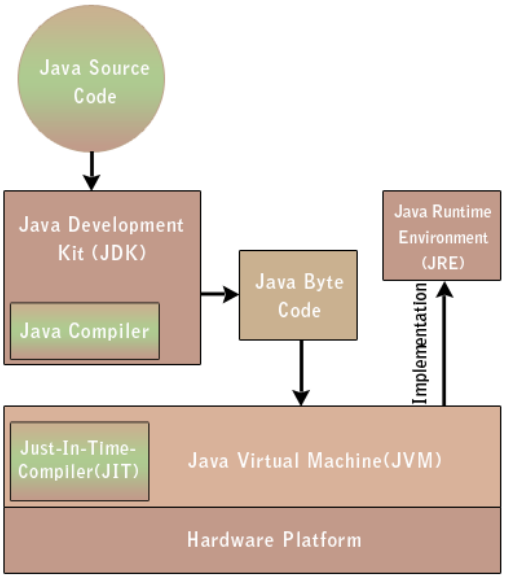
JVMs are available for many hardware and software platforms. The use of the same bytecode for all JVMs on all platforms allows Java to be described as a "compile once, run anywhere" programming language, as opposed to "write once, compile anywhere", which describes cross-platform compiled languages. Thus, the JVM is a crucial component of the Java platform.

As long as a computer has a Java VM (Virtual Machine), a Java program can run on these machines,

- Windows 2000
- Linux
- Solaris
- MacOS

Difference between JVM (Java Virtual Machine) and JDK (Java Development Kit)

The Java Virtual Machine (JVM) executes the Java programs bytecode (.class file). The bytecode is generated after the compilation of the program by the Java compiler. The Java Virtual Machine is the software program and data structure that is on the top of the hardware. The Java virtual machine is called "virtual" because it is an abstract computer (that runs compiled programs) defined by a specification. The Java Virtual Machine is the abstraction between the compiled Java program and used hardware and operating system.



The JDK (Java Development Kit) is used for developing java applications. The JDK includes JRE, set of API classes, Java compiler, Webstart and additional files needed to write Java applications. The JDK (Java Development Kit) contains software development tools which are used to compile and run the Java program. Both JDK and JRE contain the JVM.

Java Virtual Machine (JVM) is an abstract computing machine. Java Runtime Environment (JRE) is an implementation of the JVM. Java Development Kit (JDK) contains JRE along with various development tools like Java libraries, Java source compilers, Java debuggers, bundling and deployment tools.

JVM becomes an instance of JRE at runtime of a java program. It is widely known as a runtime interpreter. The Java virtual machine (JVM) is the cornerstone on top of which the Java technology is built upon. It is the component of the Java technology responsible for its hardware and platform independence.

In general java is:

- An OOP that uses class and objects.
- A very portable object-oriented programming language
- A large supporting class library that covers many general needs.
- Can create Applets, Applications, Java Server Pages, and more.
- An open standard - the language specification is publicly, freely available.

Your First Java program and program components:

Here is a Java program that displays the message "Hello World!"

```
// A program to display the message
// Hello World! on standard output
public class HelloWorld {
    public static void main(String[] args) {
      System.out.println("Hello World!");
   }
}   // end of class HelloWorld
```

The command that actually displays the message is:

```
System.out.println("Hello World!");
```

This command is an example of a subroutine call statement. It uses a "built-in subroutine/method" named **System.out.println( )** to do the actual work. Recall that a subroutine consists of the instructions for performing some task, chunked together and given a name. That name can be used to "call" the subroutine whenever that task needs to be performed. A built-in subroutine is one that is already defined as part of the language and therefore is automatically available for use in any program.

- When you run this program, the message "Hello World!" (Without the quotes) will be displayed on standard output. The computer will type the output from the program, Hello World!
- You must be curious about all the other stuff in the above program. Part of it consists of comments. Comments in a program are entirely ignored by the computer; they are there for human readers only. This doesn't mean that they are unimportant. Programs are meant to be read by people as well as by computers, and without comments; a program can be very difficult to understand. Java has two types of comments. The first type, used in the above program, begins with // and extends to the end of a line.

The computer ignores the // and everything that follows it on the same line. Java has another style of comment that can extend over many lines. That type of comment begins with /* and ends with */.

- Everything else in the program is required by the rules of Java syntax. All programming in Java is done inside "**classes**." The first line in the above program (not counting the comments) says that this is a class named HelloWorld. "HelloWorld," the name of the class, also serves as the name of the program. In order to define a program, a class must include a subroutine named **main( )**, with a definition that takes the form:

```
public static void main(String[] args) {
      //statements
}
```

When you tell the Java interpreter to run the program, the interpreter calls the **main( )** subroutine, and the statements that it contains are executed. These statements make up the script that tells the computer exactly what to do when the program is executed. The main() routine can call subroutines that are defined in the same class or even in other classes, but it is the main() routine that determines how and in what order the other subroutines are used.

The word "public" in the first line of main() means that this routine can be called from **outside the program**. This is essential because the main() routine is called by the Java interpreter, which is something external to the program itself. ***Static*** indicates that main() is a class method not an instance method ,it is static.

The definition of the subroutine -- that is, the instructions that say what it does -- consists of the sequence of "statements" enclosed between braces, { }. A program is defined by a public class that takes the form:

```
public class program-name {
     optional-variable-declarations-and-subroutines/methods
      public static void main(String[] args) {
        // statements
     }
      optional-variable-declarations-and-subroutines/methods
}
```

Variables

What is variable in the concept of programming language?

Programs manipulate data that are stored in memory. In a high-level language such as Java, **names** are used to refer to data. It is the job of the computer to keep track of where in memory the data is actually stored; the programmer only has to remember the name. **A name used in this way to refer to data stored in memory is called a variable.**

Variable is not a name for the data itself but for a location in memory that can hold data. You should think of a variable as a **container or box where you can store data that you will need to use later**. The variable refers directly to the box and only indirectly to the data in the box. Since the data in the box can change, a variable can refer to different data values at different times during the execution of the program, but it always refers to the same box.

In Java, the **only** way to get data into a variable that is into the box that the variable names are with an assignment statement. An assignment statement takes the form:

```
datatype variablename = value;//variable declaration
```

Data type is an identifier of the variable that can tell the type of data that the variable can hold.

The Java programming language defines the following kinds of variables:

Class Variables (Static Fields)

Variable that holds data that will be shared among all instances of a class. These variables declared with static keyword. A class variable is any field declared with the static modifier; this tells the compiler that there is exactly one copy of this variable in existence regardless of how many times the class has been instantiated. The code static int numcol = 6; would create such a static field.

Instance Variables (Non-Static Fields)

Instance variables hold data for an instance of a class. Technically speaking, objects store their individual states in "non-static fields", that is, fields declared without the static keyword. Non-static fields are also known as **instance** variables because their values are unique to each instance of a class (to each object, in other words)

Local Variables

Local variables are variables that used within a block of codes. Similar to how an object stores its state in fields, a method will often store its temporary state in local variables. The syntax for declaring a local variable is similar to declaring a field (Ex, int count = 0 ;). There is no special keyword designating a variable as local; that determination comes entirely from the location in which the variable is declared, which is between the opening and closing braces of a method. As such, local variables are only visible to the methods in which they are declared; they are not accessible from the rest of the class.

Naming of variables

Names are fundamental to programming. In programs, names are used to refer to many different sorts of things. In order to use those things, a programmer must understand the rules for giving names to things and the rules for using the names to work with those things. That is, the programmer must understand the syntax and the semantics of names.

A variable in Java is designed to hold only one particular type of data; it can legally hold that type of data and no other. The compiler will consider it to be a syntax error if you try to violate this rule. We say that Java is a strongly typed language because it enforces this rule.

Every programming language has its own set of rules and conventions for the kinds of names that you're allowed to use, and the Java programming language is no different. The rules and conventions for naming your variables can be summarized as follows:

- ✓ Variable names are case-sensitive.
- ✓ A variable's name can be any legal identifier, an unlimited-length sequence of Unicode letters and digits, beginning with a letter.
- ✓ When choosing a name for your variables, use full words instead of cryptic abbreviations. Doing so will make your code easier to read and understand. In many cases it will also make your code self-documenting. Also keep in mind that the name you choose ***must not be a keyword or reserved word*** like class, main, static, void etc.
- ✓ If the name you choose consists of only one word, spell that word in all lowercase letters. If it consists of more than one word, capitalize the first letter of each subsequent word.

A variable can be used in a program only if it has first been declared. A variable declaration statement is used to declare one or more variables and to give them names. When the computer executes a variable declaration, it sets aside memory for the variable and associates the variable's name with that memory.

A simple variable declaration takes the form:

```
datatype-name variable-name;
```

The **variable-name-or-names** can be a single variable name or a list of variable names separated by commas. (We'll see later that variable declaration statements can actually be somewhat more complicated than this.) Good programming style is to declare only one variable in a declaration statement, unless the variables are closely related in some way.

For example:

```
int numberOfStudents;
String name;
double x, y;
boolean isFinished;
char firstInitial, middleInitial, lastInitial;//literal
```

It is also good style to include a comment with each variable declaration to explain its purpose in the program, or to give other information that might be useful to a human reader. For example:

```
double principal; // Amount of money invested.
double interestRate; // Rate as a decimal, not percentage.
```

Primitive Data Types

The Java programming language is statically-typed, which means that all **variables must first be declared before they can be used.** This involves stating the variable's type and name, as you've already seen:

int product= 1; int sum=0;

Doing so tells your program that a field named "product" and "sum" exist and hold numerical data, and have an initial value of "1" and "0" respectively. A variable's data type determines the values it may contain, plus the operations that may be performed on it. In addition to int, the Java programming language supports seven other primitive data types. A primitive type is predefined by the language and is named by a reserved keyword. Primitive values do not share state with other primitive values. The eight primitive data types supported by the Java programming language are:

- **byte**: The **byte** data type is an **8-bit** signed two's complement integer. It has a minimum value of -128 and a maximum value of 127 (inclusive). The byte data type can be useful for saving memory in large arrays, where the memory savings actually matters.
- **short**: The short data type is a 16-bit signed two's complement integer. It has a minimum value of -32,768 and a maximum value of 32,767 (inclusive). As with int, the same guidelines apply: you can use a short to save memory in large arrays, in situations where the memory savings actually matters.

- **int**: The int data type is a **32-bit** signed two's complement integer. It has a minimum value of -2,147,483,648 and a maximum value of 2,147,483,647 (inclusive). For integral values, this data type is generally the default choice unless there is a reason (like the above) to choose something

else. This data type will most likely be large enough for the numbers your program will use, but if you need a wider range of values, use long instead.

- **long**: The long data type is a **64-bit** signed two's complement integer. It has a minimum value of -9,223,372,036,854,775,808 and a maximum value of 9,223,372,036,854,775,807 (inclusive). Use this data type when you **need a range of values wider** than those provided by int.

- **float**: The float data type is a single-precision **32-bit** floating point. As with the recommendations for byte and short, use a float (instead of double) if you need to save memory in large arrays of floating point.

- **double**: The double data type is a **double-precision 64-bit** floating point. For decimal values, this data type is generally the default choice.

- **boolean**: The boolean data type has only two possible values: true and false. Use this data type for simple flags that track true/false conditions. This data type represents **one bit** (1 or 0) of information, but its "size" isn't something that's precisely defined.

- **char**: The char data type is a **single 16-bit Unicode character**.

In addition to the eight primitive data types listed above, the Java programming language also provides special support for **character strings** via the java.lang.String class. Enclosing your character string within double quotes will automatically create a new String object.

The Java programming language also supports a few special escape sequences for char and String literals: \b (backspace), \t (tab), \n (line feed), \f (form feed), \r (carriage return), \" (double quote), \' (single quote), and \\ (backslash).

There's also a special null literal that can be used as a value for any reference type. null may be assigned to any variable, except variables of primitive types. There's little you can do with a null value beyond testing for its presence. Therefore, null is often used in programs as a marker to indicate that some object is unavailable.

Arrays

An array is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed. This section discusses arrays in greater detail.

The following statement allocates enough memory for arrayOfInts to contain ten integer elements.

```
int[] arrayOfInts = new int[10]
```

In general, when creating an array, you use the new operator, plus the data type of the array elements, plus the number of elements desired enclosed within square brackets ('[' and ']').

elementType[] *arrayName* = new *elementType*[*arraySize*]

Now that some memory has been allocated for your array, you can assign values to its elements and retrieve those values:

```
for (int j = 0; j < arrayOfInts.length; j ++) {
   arrayOfInts[j] = j;
   System.out.println("[j] = " + arrayOfInts[j]); }
```

Finally, you can use the built-in length property to determine the size of any array. The code

```
System.out.println(anArray.length);// will print the array's size to standard output.
```

Each item in an array is called an element, and each element is accessed by its numerical index. Numbering /indexing begin with 0.

Creating, Initializing, and Accessing an Array

Like declarations for variables of other types, an array declaration has two components: the array's type and the array's name. An array's type is written as type [], where type is the data type of the contained elements; the square brackets are special symbols indicating that this variable holds an array. The size of the array is not part of its type (which is why the brackets are empty). As with variables of other types, the declaration does not actually create an array, it simply tells the compiler that this variable will hold an array of the specified type.

One way to create an array is with the new operator. The next statement in the ArrayDemo program allocates an array with enough memory for ten integer elements and assigns the array to the anArray variable.

```
anArray = new int[10];  // create an array of integers
```

The next few lines assign values to each element of the array:

```
anArray[0] = 100; // initialize first element
anArray[1] = 200; // initialize second element
anArray[2] = 300; // etc.
```

Each array element is accessed by its numerical index:

```
System.out.println("Element 1 at index 0: " + anArray[0]);
System.out.println("Element 2 at index 1: " + anArray[1]);
System.out.println("Element 3 at index 2: " + anArray[2]);
```

Alternatively, you can use the shortcut syntax to create and initialize an array:

```
int[] anArray = {100, 200, 300, 400, 500, 600, 700, 800, 900, 1000};
```

Here the length of the array is determined by the number of values provided between { and }.

You can also declare an array of arrays (also known as a multidimensional array) by using two or more sets of square brackets, such as String[][] names. Here String is the data type and names is the name of multidirectional arrays. Each element, therefore, must be accessed by a corresponding number of index values.

In the Java programming language, a multidimensional array is simply an array whose components are themselves arrays. A consequence of this is that the rows are allowed to vary in length, as shown in the following MultiDimArrayDemo program:

```
class MultiDimArrayDemo {
   public static void main(String[] args) {
      String[][] names = {{"Mr. ", "Mrs. ", "Ms. "},
                          {"Smith", "Jones"}};
      System.out.println(names[0][0] + names[1][0]); //Mr. Smith
      System.out.println(names[0][2] + names[1][1]); //Ms. Jones
   }
}
```

Strings

A sequence of character data is called a **string** and is implemented in the Java environment by the **String class** (a member of the java.lang package

Java provides a String class to deal with sequences of characters.

String Studentname = "Keria Mohammed";

The String class provides a variety of methods to operate on String objects.

The **equals()** method is used to compare Strings.

The **length()** method returns the number of characters in the String.

Java lets you to concatenate strings together easily using the + operator.

Example:

```
class student{
     string firstname=”Abel”;
     String middlename=”Degu”;
 Public static void main(String[] args)  {
      Student stud1=new Student();
    System.out.println(firstname + “  ” + middlename);
    System.out.println(firstname.length( ));
  }    }
```

Control Statements

The control statements are used to control the flow of execution of the program. This execution order depends on the supplied data values and the conditional logic. Java contains the following types of control statements:

1- Selection Statements

2- Repetition Statements

3- Branching Statements like break, continue, return (Read from books / internet)

Selection statements:

1. **If Statement**

This is a control statement to execute a single statement or a block of code, when the given condition is true and if it is false then it skips **if** block and rest code of program is executed.

Syntax:

```
if(conditional_expression){
    <statements>;
    ...;
    ...;
}
```

Example: If n%2 evaluates to 0 then the "if" block is executed. Here it evaluates to 0 so if block is executed. Hence **"This is even number"** is printed on the screen.

```
int n = 10;
 if(n%2 = = 0){
 System.out.println("This is even number");
}
```

2. If-else Statement

The **"if-else"** statement is an extension of if statement that provides another option when 'if' statement evaluates to "false" i.e. else block is executed if **"if"** statement is false.

Syntax:

```
if(conditional_expression){
    <statements>;
    ...;
    ...;
 }
else{
    <statements>;
    ....;
    ....;
}
```

Example: If n%2 doesn't evaluate to 0 then else block is executed. Here n%2 evaluates to 1 that is not equal to 0 so else block is executed. So **"This is not even number"** is printed on the screen.

```
int n = 11;
 if(n%2 = = 0){
System.out.println("This is even number");
     }
else{
     System.out.println("This is not even number");     }
```

The following program, IfElseDemo, assigns a grade based on the value of a test score: an A for a score of 90% or above, a B for a score of 80% or above, and so on.

```
class IfElseDemo {
   public static void main(String[] args) {
     int testscore = 76;
     char grade;
     if (testscore >= 90) {
        grade = 'A';
     } else if (testscore >= 80) {
        grade = 'B';
     } else if (testscore >= 70) {
```

```
        grade = 'C';
    } else if (testscore >= 60) {
        grade = 'D';
    } else {
        grade = 'F';
    }
    System.out.println("Grade = " + grade);
}  }
```

You may have noticed that the value of testscore can satisfy more than one expression in the compound statement: 76 >= 70 and 76 >= 60. However, once a condition is satisfied, the appropriate statements are executed (grade = 'C';) and the remaining conditions are not evaluated.

3. Switch Statement

This is an easier implementation to the if-else statements. The keyword **"switch"** is followed by an expression that should evaluates to byte, short, char or int primitive data types, only. In a switch block there can be one or more **labeled cases**. The expression that creates labels for the case must be unique. The switch expression is matched with each case label. Only the matched case is executed, if no case matches then the default statement (if present) is executed.

Syntax:

```
switch(control_expression){
    case expression 1:
        <statement>;
    case expression 2:
        <statement>;
      ...
      ...
    case expression n:
        <statement>;
    default:
        <statement>;
}//end switch
```

Example: Here expression "day" in switch statement evaluates to 5 which matches with a case labeled "5" so code in case 5 is executed that results to output **"Friday"** on the screen.

```
int day = 5;
switch (day) {
      case 1:
          System.out.println("Monday");
          break;
      case 2:
          System.out.println("Tuesday");
          break;
     case 3:
         System.out.println("Wednesday");
         break;
     case 4:
         System.out.println("Thrusday");
         break;
     case 5:
         System.out.println("Friday");
         break;
     case 6:
         System.out.println("Saturday");
         break;
     case 7:
          System.out.println("Sunday");
          break;
     default:
          System.out.println("Invalid
entry");
          break;
}
```

The following code example, SwitchDemo, declares an int named month whose value represents a month. The code displays the name of the month, based on the value of month, using the switch statement.

```
public class SwitchDemo {
   public static void main(String[] args) {
      int month = 8;
      String monthString;
      switch (month) {
         case 1:  monthString = "January";      break;
         case 2:  monthString = "February";     break;
         case 3:  monthString = "March";        break;
         case 4:  monthString = "April";        break;
         case 5:  monthString = "May";          break;
         case 6:  monthString = "June";         break;
         case 7:  monthString = "July";         break;
         case 8:  monthString = "August";       break;
         case 9:  monthString = "September";    break;
```

```
        case 10: monthString = "October";       break;
        case 11: monthString = "November";      break;
        case 12: monthString = "December";      break;
        default: monthString = "Invalid month"; break;
      }
      System.out.println(monthString);
  }}
```

The body of a switch statement is known as a switch block. A statement in the switch block can be labeled with one or more case or default labels. The switch statement evaluates its expression, and then executes all statements that follow the matching case label.

- *You could also display the name of the month with if-then-else statements:*

Deciding whether to use if-then-else statements or a switch statement is based on readability and the expression that the statement is testing. An if-then-else statement can test expressions based on ranges of values or conditions, whereas switch statement tests expressions based only on a single integer, enumerated value, or String object.

Another point of interest is the break statement. Each break statement terminates the enclosing switch statement. Control flow continues with the first statement following the switch block. The break statements are necessary because without them, statements in switch blocks fall through: All statements after the matching case label are executed in sequence, regardless of the expression of subsequent case labels, until a break statement is encountered.

Repetition Statements:

1. While loop statements:

This is a looping or repeating statement. It executes a block of code or statements till the given conditions are true. The expression must be evaluated to a Boolean value. It continues testing the condition and executes the block of code. When the expression results to false control comes out of loop.

Syntax:

```
while(expression){
    <statement>;
    ...;
    ...;
 }
```

Example: Here expression i<=10 is the condition which is checked before entering into the loop statements. When i is greater than value 10 control comes out of loop and next statement is executed. So here i contains value "1" which is less than number "10" so control goes inside of the loop and prints current value of i and increments value of i. Now again control comes back to the loop and condition is checked. This procedure continues until i becomes greater than value "10". So, this loop prints values 1 to 10 on the screen.

```
int i = 1;
  //print 1 to 10
    while (i <= 10){
  System.out.println("Num " + i);
        i++;     }
```

The while statement evaluates expression, which must return a boolean value. If the expression evaluates to true, the while statement executes the *statement*(s) in the while block. The while statement continues testing the expression and executing its block until the expression evaluates to false. Using the while statement to print the values from 1 through 10 can be accomplished as in the following WhileDemo program:

```
class WhileDemo {
    public static void main(String[] args){
        int count = 1;
        while (count < 11) {
            System.out.println("Count is: " + count);
            count++;
        }    }  }
```

2. Do-while Statements

The Java programming language also provides a do-while statement, which can be expressed as follows:

```
do {
    statement(s)
} while (expression);
```

The difference between do-while and while is that do-while evaluates its expression at the bottom of the loop instead of the top. Therefore, the statements within the do block are always executed at least once, as shown in the following DoWhileDemo program:

```
class DoWhileDemo {
    public static void main(String[] args){
        int count = 1;
        do {
            System.out.println("Count is: " + count);
            count++;
        } while (count <= 11);
    }}
```

3. The for Statement

The for statement provides a compact way to iterate over a range of values. Programmers often refer to it as the "for loop" because of the way in which it repeatedly loops until a particular condition is satisfied. The general form of the for statement can be expressed as follows:

for (*initialization*; *termination*; *increment*) {
   statement(s)
}

When using this version of for statement, keep in mind that:

- The ***initialization*** expression initializes the loop; it's executed once, as the loop begins.
- When the ***termination*** expression evaluates to false, the loop terminates.

The ***increment*** expression is invoked after each iteration through the loop; it is perfectly acceptable for this expression to increment or decrement a value.

The following program, ForDemo, uses the general form of for statement to print the numbers 1 through 10 to standard output:

```
class ForDemo {
    public static void main(String[] args){
        for(int i=1; i<11; i++){
            System.out.println("Count is: " + i);
        }    }   }
```

The scope of this variable extends from its declaration to the end of the block governed by the for statement, so it can be used in the termination and increment expressions as well. If the variable that controls for statement is

not needed outside of the loop, it's best to declare the variable in the initialization expression. The names i, j, and k are often used to control for loops; declaring them within the initialization expression limits their life span and reduces errors.

The three expressions of the for loop are optional; an infinite loop can be created as follows:

```
for ( ; ; ) {    // infinite loop
        // your code goes here
}
```

The for statement also has another form designed for iteration through arrays .This form is sometimes referred to as the enhanced for statement, and can be used to make your loops more compact and easy to read. To demonstrate, consider the following array, which holds the numbers 1 through 10:

```
int[] numbers = {1,2,3,4,5,6,7,8,9,10};
```

The following program, EnhancedForDemo, uses the enhanced for loop through the array:

```
class EnhancedForDemo {
    public static void main(String[] args){
        int[] numbers = {1,2,3,4,5,6,7,8,9,10};
        for (int item : numbers) {
          System.out.println("Count is: " + item);
        }    }    }
```

In this example, the variable item holds the current value from the numbers array.

Chapter Two

More on Classes and Objects

2.1 Overview

When you faced with a problem domain and you have to architect a solution. Given that Java is an object-oriented language, you should perform an object-oriented design. In this process, you will end up with classes, objects, attributes, and behaviors/methods.

Object-oriented design process involves the following three tasks:

- Dividing the problem domain into types of objects/classes,
- Modeling the relationships between classes
- Modeling the attributes and behaviors /methods of each type.

These tasks are not listed in any particular order. Most likely, you will perform these tasks iteratively throughout the design process.
In an object-oriented design:

- You identify the fundamental objects of the problem domain, the "things" involved.
- You then classify the objects into types by identifying groups of objects that have common characteristics and behaviors.

The types of objects you identify in the problem domain become "types" in your solution. The program you write will create and manipulate objects of these types.

The fundamental task of abstraction in an object-oriented design is to identify classes and objects in the problem domain and then to identify attributes and methods.

- Classes combine data and methods.
- A class defines a data type.
- Advantages: Classes correspond to concepts in the problem domain.
- Classes reduce complexity by increasing *coherence*.

Class = data + methods.

Everything (data and methods) in Java is contained in classes. So far, you've been using classes to hold methods (main and perhaps a few other static methods). These methods use local variables, which are completely private to the method, and which disappear when the method returns.

2.2 Advantage of thinking a solution in terms of "*Classes*"

Classes used to model problem.

One advantage of encapsulating data and methods in a class is to make programming objects that reflect "objects" in the problem domain. If your problem deals with **sales,** then you'll very likely have classes called *Order* and *Produc*t.

Classes used to reduce complexity.

Another advantage of classes is reducing *complexity*. Complexity limits the size and reliability of programs. Complexity is reduced by **increasing *cohesion*** (**putting things together that belong together**) and **reducing *coupling*** (interconnections).

2.3 Modeling real world problems:

The main task of OOP is to modeling the real-world problem using **classes** and **objects**. Classes and objects are models that used *to represent the real-world problem that supposed to be solved*. A model is a representation of simple important aspects of the real world. Model used in software development like, representation of input, objects, object interaction and classes. Abstraction is the main source for modeling. These models represent classes and objects. Thus, classes and objects are a representation of reality, abstracted from the complex real world.

Finding classes and objects

- The first step in finding classes and objects is studying the problem domain.
- Classes and objects should initially emerge from the problem domain itself.
- You should prepare to spend a good deal of time in investigation.
- Observe first hand documents by working with the end users.
- Take the specialist in that area and problem domain experts.
- Look for previous results on similar problem domain.
- Construct a **prototype** – simple skeleton of real system, using classes, objects and including expected interactions between.

After this is done, the classes and objects can be used to

1. Identify the potential/necessary attribute or fields
2. Identify methods required to access its data.

Class name
List of Attributes
List of Methods

Examples of systems that need OO solution

- **Example 1:** Consider ***Cost Sharing System***, use abstraction feature and list possible classes, attributes, and methods
- **Example 2:** Consider ***Dormitory System***, use abstraction technique and list possible classes, attributes, and methods. Do the same for Car Registration System.

Class is similar to database table. If you are familiar with databases, a *class* is very similar to a *table definition*.

A class has two main sections: Private and public

- A **private** part can only be accessible by member methods (by all methods defined inside the class).
- A **public** part can be used by other methods and classes outside the given class.

Examples of defining Classes and Objects in java

Example 1: it shows how to define private and public sections of a class

```
public class Myclass {
   Private int x;  // accessed only inside Myclass
   Public float y;  //can be accessed outside Myclass
   Private method1 ( ) {} //can't access data outside this class
   Public method2 ( ) {  } //can access data from other class
 }
```

Example 2: Here is the class again that represents information about a student.

```
// Purpose: Information about a student.
public class Student {
   public String fName; // First name
   public String lName;  // Last name
   private int    id;       // Student id
}
```

2.4 Use *new* operator to create a new object

A class defines what fields an *object* will have when it's created. When a new Student object is created, a block of memory is allocated, which is big enough to hold these three fields -- as well as some extra overhead that all objects have.

Student stud;
stud = new Student(); // Create Student object with **new** operator.
Or we can write in one line in the following way:
Student stud=new Student ( );

To create a new object, write new followed by the name of the class, followed by parentheses. Later we'll see that we can specify arguments in the parentheses when creating new objects.

Access public fields with dot notation

All public fields can be referenced using **dot notation** (later we'll see better ways to access and set fields). To reference a field, write the object name, then a dot, then the field name or method name.

Student stud; // Declare a variable to hold a Student object.
stud = new Student(); // Create a new Student object with default values.
stud.firstName = "Rishan"; // Set values of the fields.
stud.lastName = "Mezegeb";
stud.id   = 1234;

A class definition is a template for creating objects. A class defines which fields an object has in it. You need to use new to create a new object from the class by allocating memory and assigning a default value to each field. A little later you will learn how to write a ***constructor*** to control the initialization.

2.5 Constructor

☞ Constructor can be Default and Parameterized

A constructor is a special initialization method that is called automatically whenever an instance of a class is created. The constructor member method has, by definition, the same name as the corresponding class. The constructor has no return value specification. The Java run-time system makes sure that the constructor of a class is called when instantiating an object.

Constructor Method

- Automatically invoked when a class instance is created using the *new* operator
  - Has same name as the class
  - Has no return type
  - Java creates a default constructor if no constructor has been explicitly written for a class

Constructors are used to *initialize the data members* of the class. If we have data members that must be initialized, we must have a constructor

Default (Non-Parameterized) constructors

Used to initialize objects using default values, let us look at the following example.

```
public class Employee{
     String empName;
     String address;
      int id;
        public Employee( )  // Default constructor
          {   empName = " ";
              address = " ";
               int=0;
          }
} //end class
```

The constructor Employee () is a *default* or a *non-parameterized* constructor, because it doesn't use parameters. Default parameter is used to initialize the object with default value

Parameterized constructors

The parameterized constructors can accept values into the constructor that has different parameters in the constructor. The value would be transferred from the **main ( ) method to the constructor either with direct values or through variables.**

Parameterized Constructor has parameter list to receive arguments for populating the instance attributes.

When you create a new instance (a new object) of a class using the new keyword, a *constructor* for that class is called. Constructors are used to initialize the instance variables (fields) of an object. Constructors are similar to methods, but with some important differences.

- **Constructor name is class name** i.e., they must have the *same name as the class.*
- **Default constructor**. If you don't define a constructor for a class, a *default (parameter less) constructor* is automatically created by the compiler. The default constructor calls the default parent constructor (super ( )) and initializes all instance variables to default value (zero for numeric types, null for object references, and false for Booleans).

Difference between constructor and method in java

There are many differences between constructors and methods. They are given below.

Java Constructor	Java Method
Constructor is used to initialize the state of an object.	Method is used to expose behavior of an object.
Constructor must not have return type.	Method must have return type.
Constructor is invoked implicitly.	Method is invoked explicitly.
The java compiler provides a default constructor if you don't have any constructor.	Method is not provided by compiler in any case.
Constructor name must be same as the class name.	Method name may or may not be same as class name.

There can be more than one constructor in a class, distinguished by their parameters. When the class is initialized, Java will call the right constructor with matching arguments and type.

Example 1: //without constructors'

```
public class Circle {
    public int  x, y; // centre of the circle
    public int r;    // radius of circle
   //Methods to return circumference and area
    public double circumference() {
                return 2*3.14*r;
    }
    public double area() {
                return 3.14 * r * r;
    }
```

```
        public static void main(String args[]) {
            Circle aCircle;  // creating reference
            aCircle = new Circle(); // creating object
            aCircle.x = 10;  // assigning value to data field
            aCircle.y = 20;
            aCircle.r = 5;
            double  area = aCircle.area(); // invoking method
            double circumf = aCircle.circumference();
         System.out.println("Radius="+aCircle.r+" Area="+area);
         System.out.println("Radius="+aCircle.r+" Circumference ="+circumf);
    }}
```

Example 2: // Write the above code using constructors

```
public class Circle {
    public int  x, y; // centre of the circle
    public int r;    // radius of circle
  Circle( )
   {   x=0; y=0; r=0;   }
Circle(int valx , int valy , int valr)
    {      x=valx;   y=valy ;   r=valr;   }
 //Methods to return circumference and area
    public double circumference( )
  {  return 2*3.14*r;      }
public double area( )
  {    return 3.14 * r * r;     }
 public static void main (String args[]){
            Circle c1=new Circle();
System.out.println("The value of x is " + c1.x);
System.out.println("The value of yis "+ c1.y);
System.out.println("The value of r is " + c1.r);
          Circle c2=new Circle(4,2,5);
System.out.println("The value of x is " + c2.x);
System.out.println("The value of y is " + c2.y);
```

```
System.out.println("The value of r is " + c2.r);
System.out.println("Area of c2 is" +c2.area( ));
System.out.println("Circumference of c2 is "+c2. circumference( ));
  }}
```

Example 3: It contains two constructors - Default and Parameterized. This example is about a polygon, Cube

```
public class Cube1 {
   int length;
   int width;
   int height;
   public int getVolume() {
            return (length *width * height);
   }
   Cube1( ) {
            length = 0;
            width = 0;
            height = 0;
   }
Cube1 (int l, int b, int h) {
            length = l;
            width = b;
            height = h;
   }
public static void main(String[] args) {
   Cube1 cubeObj1, cubeObj2;
   cubeObj1 = new Cube1( );
   cubeObj2 = new Cube1(10, 10, 10);
   System.out.println("Volume of Cube is : " + cubeObj1.getVolume());
   System.out.println("Volume of Cube is : " + cubeObj2.getVolume());
   }}
```

If such a class requires a default constructor, its implementation must be provided. Any attempt to call the default constructor will be a compile time error if an explicit default constructor is not provided in such a case.

Summary

- Constructor is a special method that gets invoked "automatically" at the time of object creation.
- Constructor is normally used for initializing objects with default values unless different values are supplied.
- Constructor has the same name as the class name.
- Constructor cannot return values.
- A class can have more than one constructor as long as they have different signature (i.e., different input arguments syntax).

Chapter Three

Inheritance and Polymorphism

3.1. Inheritance:

To know the concept of inheritance clearly you must have the idea of class and its features like methods, data members, access controls, constructors, keywords this, super etc.

As the name suggests, inheritance means to take something that is already made. It is one of the most important features of Object Oriented Programming. It is the concept that is used for reusability purpose. Inheritance is the mechanism through which we can derive classes from other classes. The derived class is called as child class or the subclass or we can say the extended class and the class from which we are deriving the subclass is called the base class or the parent class. To derive a class in java the keyword extends is used.

In inheritance, a new class is defined by means of an older, pre-existing class. This leads to a situation in which, the new class has all the functionality of the older, pre-existing class and, additionally, introduces its own specific functionality. We say the new class(child or subclass or derived class) inherits the functionality of another existing class (base or super class).

One advantage of OOP is the *re-usage* of code. The capability to define custom data types using classes enables us to reuse the code that we develop. In real world, we may need an object that is almost similar to an already developed object but not exactly similar. Inheritance enables us to reuse an object more quickly; thus making slight adjustments where necessary.

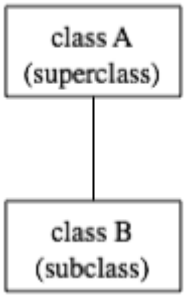
To create a new derived class based on another pre-existing class, use the following syntax:

```
modifier class SubClass extends BaseClass {
        // Body of the Subclass
}
```

The following kinds of inheritance are there in java.

Simple Inheritance: One level inheritance

When a subclass is derived simply from its parent class then this mechanism is known as simple inheritance. In case of simple inheritance there is only a sub class and its parent class. It is also called single inheritance or one level inheritance.



In Java, to create a class named "B" as a subclass of a class named "A", you would write: -

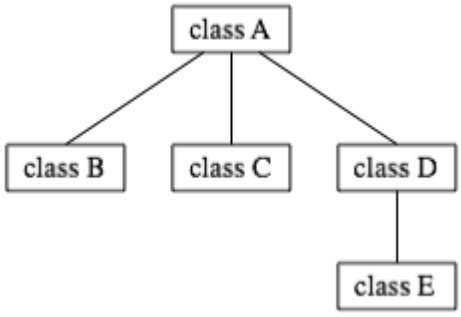
```
class B extends A {

    // additions to, and modifications of,

      }
```

Single level inheritance Example:

```
public class baseclass {
 public  int x,y;
  public baseclass(){
     x=0; y=0;
      System.out.println("Baseclass is executed");
   }
public baseclass(int x1,int y1){
    x=x1;
    y=y1;
     }}
  public class subclass extends baseclass {
 public int z;
  public subclass(){
       super();
       this.z=0;
       System.out.println("Subclass is executed");
   }
public subclass(int x1,int y1,int z1){
      super(x1,y1);
      this.z=z1;
} }
public class Testsubclass {
public static void main(String args[]){
   subclass sc=new subclass();
   System.out.println(sc.x);// output 0
   sc=new subclass(4,8,1);
System.out.println(sc.x);// output 4
baseclass bc=new baseclass (2,5);
System.out.println(bc.x);
    }}
```

Multilevel inheritance:

Several classes can be declared as subclasses of the same superclass. The subclasses, which might be referred to as "sibling classes," share some structures and behaviors -- namely, the ones they inherit from their common superclass. The superclass expresses these shared structures and behaviors. In the diagram to the left, classes B, C, and D are sibling classes. Inheritance can also extend over several "generations" of classes. This is shown in the diagram, where class E is a subclass of class D which is itself a subclass of class A. In this case, class E is considered to be a subclass of class A, even though it is not a direct subclass. This whole set of classes form a small class hierarchy. Such type of inheritance is called multilevel inheritance.



Multilevel Inheritance Example:

```
class A {
   System.out.println("Class A is executed");    }
class B extends A{
   System.out.println("Subclass B Derived from class A executed");   }
class C extends B{
   System.out.println("Subclass C Derived from class B executed");    }
 public static void main(String args[]){
   C c  = new C();
      }}
```

The mechanism of inheriting the features of more than one base class into a single class is known as multiple inheritances. Java does not support multiple inheritances but the multiple inheritances can be achieved by using the interface.

In Java Multiple Inheritances can be achieved through use of Interfaces by implementing more than one interface in a class.

Using *super* and *this* keyword

The super is java keyword. As the name suggest super is used to access the members of the super class. It is used for two purposes in java. The first use of keyword super is to access the hidden data variables of the super class hidden by the sub class.

Suppose class A is the super class that has two instance variables as int a and float b. class B is the subclass that also contains its own data members named a and b. then we can access the super class (class A) variables a and b inside the subclass class B just by calling the following command.

super.member;

Here member can either be an instance variable or a method. This form of super is most useful to handle situations where the local members of a subclass hide the members of a super class having the same name. The following examples clarify all the confusions.

```
class A{
 int a;
 float b;
 void Show(){
  System.out.println("b in super class:  " + b);
 }}
class B extends A{
 int a;
 float b;
 B( int p, float q){
  a = p;
  super.b = q; //using variable b from super class

  this.b=6;
 }
void Show(){
   super.Show(); //calling super method show()
   System.out.println("b in super class:  " + super.b);

  //display value of super variable b
   System.out.println("a in sub class:    " + a);

   System.out.println("b in subclass :  "+  b);

//or this.b,used to display values of subclass variable b
 }}
 class testsubclass{

 public static void main(String[] args){
    B subobj = new B(1, 5);
    subobj.Show();
  }}
```

The second use of super to call super class constructor: The second use of the keyword super in java is to call super class constructor in the subclass. This functionality can be achieved just by using the following command.

super (param-list);

Here parameter list is the list of the parameter requires by the constructor in the super class. super must be the first statement executed inside a super class constructor. If we want to call the default constructor then we pass the empty parameter list. The following program illustrates the use of the super keyword to call a super class constructor.

```
class A{
  int a;
  int b;
  int c;
  A(int p, int q, int r){
    a=p;
    b=q;
    c=r;
  }}

  class B extends A{
    int d;
    B(int l, int m, int n, int o){
      super(l,m,n);
      d=o;
    }
   void Show(){
     System.out.println("a = " + a);
     System.out.println("b = " + b);
     System.out.println("c = " + c);
     System.out.println("d = " + d);
   }}

  Class testsupermethod{
     public static void main(String args[]){
       B b = new B(4,3,8,7);
       b.Show();
     }  }
```

In general, the keyword *super()* has two general forms. The *first* is *call of superclass constructor*. The *second* is used to access a *member (data and method) of the superclass* that has been *hidden* by a member of the subclass. Additional examples to see each use of the super keyword:

Example 1:

```
public class person {
  private String name;
  private int age;
public person(String name,int age)    {
       this.name = name;
       this.age = age;
    }
public void printData1(){
System.out.println("Name->"+name);
System.out.println("Age->"+age);
}
}//end class
public class student extends person {
    String idno ;
    String department;
public student(String studname,int studage,String
id, String dept){
//calling superclass constructor
super(studname,studage);
    this.idno=id;
     this.department=dept;
}
public void printData2(){
 System.out.println("Id No.->"+idno);
 System.out.println("Department->"+department);
}  }
public class Teststudent {
    public static void main(String args[]){
    student stud = new student("Abreham", 24,
"TER/123/01", "IT");
    stud.printData1();
    stud.printData2();
}  }
```

Example 2:

```
public class ClassA{
int i;
}//end class
public class classB extends ClassA{
int i;//this i hides the i in class ClassA
public ClassB(int a,int b){
super.i = a;//i in ClassA
i = b;//i in ClassB
}
public void show(){
System.out.println("i in superclass : "+super.i);
System.out.println("i in subclass : "+i);
}}
public class TestSuper{
public static void main(String args[]){
ClassB clsB = new ClassB(10,20);
clsB.show();
}}
```

Although the instance variable i in Class B hides the i in Class A, super allows access to the i defined in the superclass. Super can also be used to call methods that are hidden by subclasses.

Use of *final keyword* with Inheritance

A class can be declared final if its definition is complete and no subclasses are desired or required. A compile-time error occurs if the name of a final class appears in the extended clause of another class declaration; this implies that a final class can't have any subclasses.

Final keyword prevents *overriding*. To disallow a method from being overridden, specify final as a modifier at the start of its declaration. Methods declared as final can't be overridden. Have a look at the following example.

```
public class ClassA{
public final void firstMethod( ){
System.out.println("This is a final method declared only once");
}}
public class ClassB extends ClassA{
public void firstMethod( ){
    System.out.println("illegal to declare this method again");
} }
```

Final prevents *inheritance*. Sometimes, you will want to prevent a class from being inherited. To do this, precede the class declaration with the keyword *final*. Declaring a class as final implicitly declares *all of its methods* as final too. Consider the following example.

```
public final class ClassA{
//…
}//end class
public class ClassB extends ClassA{//illegal inheritance
}//end class
```

Constructor order dependencies

Instantiating a subclass object begins a chain of constructor calls in which the subclass constructor, before performing its own tasks, invokes its direct superclass's constructor (calling the superclass's default constructor or no-argument constructor). Similarly, if the superclass was derived from another class, the superclass constructor would be required to invoke the constructor of the next class up in the hierarchy, and so on. The last constructor called in the chain is always the constructor of class *Object*. The original subclass constructor's body finishes executing *last*.

Constructor example

Ex1). The following three classes demonstrate how constructors are called for subclass and superclass.

```
public class ParentClass {
  public ParentClass() {
System.out.println( "ParentClass constructor was called" );
  }}
public class ChildClass extends ParentClass {
  public ChildClass() {
    System.out.println( "ChildClass constructor was called" );
  }}
public class test {
  public static void main(String[] args) {
    ChildClass cc = new ChildClass();
  }}
```

When you extend a class, the new class must choose one of its superclass's constructors to invoke.

Ex 2) When an object is created, it's necessary to call the constructors of all super classes to initialize their fields. Java does this automatically at the beginning *if you don't*.

```
Public class Base{
   System.out.println("Base class accessed first");
    }
public class Derived1 extends Base{
   System.out.println(" Derived1 class is accessed secondly");
  }
Public  class Derived2 extends Derived1 {
   System.out.println(" Derived2 class is accessed Thirdly");
  }
public class  test {
public static void main(String[] args)
Derived2 d2=new Derived2();//It automatically creates default constructors of //base and super constructors
 }}
```

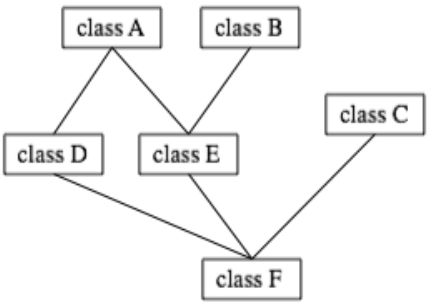
Private Methods Are Not Inherited

As we noted in the previous topics, an instance variable (or method) that is private in a base class is not directly accessible in the definition of a method for *any other class, not even in a method definition for a derived class*. Note that private methods are just like private variables in terms of not being directly available. But in the case of methods, the restriction is more dramatic. A private variable can be accessed indirectly (through public methods of the class). A private method is simply not available. It is just as if the private method were not inherited.

This should not be a problem. Private methods should just be used as helping functions, and so their use should be limited to the class in which they are defined. If you want a method to be used as a helping method in a number of inherited classes, then it is not *just* a helping method, and you should make the method public.

Interfaces: Multiple inheritances in java

Some object-oriented programming languages, such as C++, allow a class to extend two or more super classes. This is called multiple inheritance. In the illustration below, for example, class E is shown as having both class A and class B as direct super classes, while class F has three direct super classes.



Multiple inheritance (NOT allowed in Java)

Such multiple inheritance is not allowed in Java. The designers of Java wanted to keep the language reasonably simple, and felt that the benefits of multiple inheritances were not worth the cost in increased complexity. However, Java does have a feature that can be used to accomplish many of the same goals as multiple inheritances called interface.

In Java, interface is a reserved word with an additional, technical meaning. An "interface" in this sense consists of a set of instance method interfaces, without any associated implementations. A class can implement an interface by providing an implementation for each of the methods specified by the interface.

Interfaces are similar to abstract classes but all methods are *abstract* and all properties are *static final*. Interfaces can be inherited (i.e. you can have a sub-interface). As with classes the *extends* keyword is used for inheritance. Java does not allow *multiple inheritance* for classes (i.e. a subclass being the extension of more than one superclass). An *interface* is used to tie elements of several classes together. Interfaces are also used to separate *design* from *coding* as class method headers are specified but not their bodies.

3.2. Polymorphism in Java

Polymorphism in java is a concept by which we can perform a *single action by different ways*. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in java: compile time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

3.2.1. Method Overloading in Java

If a class has multiple methods by same name but different parameters, it is known as **Method Overloading**.

If we have to perform only one operation, having same name of the methods increases the readability of the program.

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int, int) for two parameters, and b(int, int, int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs. So, we perform method overloading to figure out the program quickly.

Different ways to overload the method

There are two ways to overload the method in java

1. By changing the data type
2. By changing number of arguments

In java, Method Overloading is not possible by changing the return type of the method.

Example:

```
class Maths{
  void sum(int a,int b){
System.out.println(a+b);
}
  void sum(int a,int b,int c){
System.out.println(a+b+c);
}
      void add(int a,int b){
      System.out.println(a+b);
      }
      void add(double a,double b){
      System.out.println(a+b);
      }
public static void main(String args[]){
Maths m=new Maths ();
m.sum(10,10,10);
m.sum(20,20);
m.add(10.5,10.5);
m.add(20,20);      }  }
```

3.2.2. Method Overriding in Java

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in java**.

In other words, if subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

- Method overriding is used to provide specific implementation of a method that is already provided by its super class.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

1. method must have same name as in the parent class
2. method must have same parameter as in the parent class.
3. must be IS-A relationship (inheritance).

Example:

```
class Vehicle{

void run(){

System.out.println("Vehicle is running");

} }

class Bike2 extends Vehicle{

void run(){

System.out.println("Bike is running safely");

}

public static void main(String args[]){

Bike2 obj = new Bike2();

obj.run();

}
```

Difference between method overloading and method overriding in java

No.	Method Overloading	Method Overriding
1)	used *to increase the readability* of the program.	used *to provide the specific implementation* of the method that is already provided by its super class.
2)	performed *within class*.	occurs *in two classes* that have IS-A (inheritance) relationship.
3)	*parameter must be different*.	*parameter must be same*.
4)	example of *compile time polymorphism*.	example of *run time polymorphism*.
5)	can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading.	*Return type must be same or covariant* in method overriding.

3.2.3. Abstraction in Java

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only important things to the user and hides the internal details for example sending SMS, you just type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

3.2.3.1 Abstract class in Java

A class that is declared with abstract keyword, is known as abstract class in java. It can have abstract and non-abstract methods (method with body).

Abstract class in Java

A class that is declared as abstract is known as abstract class. It needs to be extended and its method implemented. It cannot be instantiated.

Example abstract class

1. abstract class A{}

Abstract method

A method that is declared as abstract and does not have implementation is known as abstract method.

Example abstract method

1. abstract void printStatus();//no body in abstract

Example of abstract class that has abstract method

In this example, Bike the abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

```
abstract class Bike{
  abstract void run();
}
  class Honda4 extends Bike{
void run(){System.out.println("running safely..");}
public static void main(String args[]){
 Bike obj = new Honda4();
 obj.run();  }  }
```

Understanding the real scenario of abstract class

In this example, Shape is the abstract class, its implementation is provided by the Rectangle and Circle classes.

Mostly, we don't know about the implementation class (i.e. hidden to the end user) and object of the implementation class is provided by the factory method.

A factory method is the method that returns the instance of the class. We will learn about the factory method later.

In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

```
abstract class Shape{
abstract void draw();
}
//In real scenario, implementation is provided by others i.e. unknown by end user
class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle");}
}
class Circle1 extends Shape{
void draw(){System.out.println("drawing circle");}
}
//In real scenario, method is called by programmer or user
class TestAbstraction1{
public static void main(String args[]){
Shape s=new Circle1();//In real scenario, object is provided through method e.g. getShape() method s.draw();


} }
```

Another example of abstract class in java

```
 abstract class Bank{
abstract int getRateOfInterest();
}
class SBI extends Bank{
int getRateOfInterest(){return 7;}
}
class PNB extends Bank{
int getRateOfInterest(){return 7;}
}
class TestBank{
public static void main(String args[]){
Bank b=new SBI();//if object is PNB, method of PNB will be invoked
int interest=b.getRateOfInterest();
System.out.println("Rate of Interest is: "+interest+" %");
}}
```

Abstract class having constructor, data member, methods etc.

An abstract class can have data member, abstract method, method body, constructor and even main() method.

```
//example of abstract class that have method body
 abstract class Bike{
   Bike(){System.out.println("bike is created");}
   abstract void run();
   void changeGear(){System.out.println("gear changed");}
 }
 class Honda extends Bike{
 void run(){System.out.println("running safely..");}
 }
 class TestAbstraction2{
 public static void main(String args[]){
  Bike obj = new Honda();
  obj.run();
  obj.changeGear();
 }  }
```

Rule: If there is any abstract method in a class, that class must be abstract.

```
class Bike12{
abstract void run();
}
```

Rule: If you are extending any abstract class that have abstract method, you must either provide the implementation of the method or make this class abstract.

Another real scenario of abstract class

The abstract class can also be used to provide some implementation of the interface. In such case, the end user may not be forced to override all the methods of the interface.

Note: If you are beginner to java, learn interface first and skip this example.

```
interface A{
void a();
void b();
void c();
void d();
}
abstract class B implements A{
public void c(){System.out.println("I am C");}
}
class M extends B{
public void a(){System.out.println("I am a");}
public void b(){System.out.println("I am b");}
public void d(){System.out.println("I am d");}
}
class Test5{
public static void main(String args[]){
A a=new M();
a.a();
a.b();
a.c();
a.d();
}}
```

3.2.3.2. Interface in Java

An interface in java is a blueprint of a class. It has static constants and abstract methods only. The interface in java is a mechanism to achieve fully abstraction. There can be only abstract methods in the java interface not method body. It is used to achieve fully abstraction and multiple inheritance in Java.

- Java Interface also represents IS-A relationship.
- It cannot be instantiated just like abstract class.

Why use Java interface?

There are mainly three reasons to use interface. They are given below.

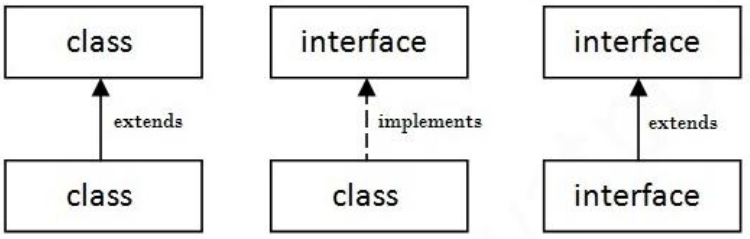
- It is used to achieve fully abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

The java compiler adds public and abstract keywords before the interface method and public, static and final keywords before data members.

In other words, Interface fields are public, static and final by default, and methods are public and abstract.

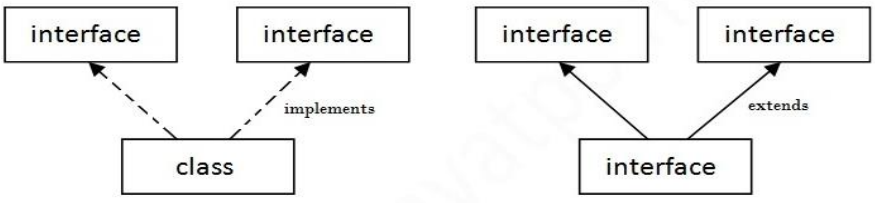
Understanding relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface but a class implements an interface.



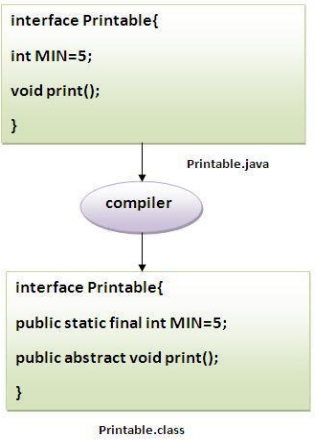
Simple example of Java interface

In this example, Printable interface have only one method, its implementation is provided in the A class.



Multiple Inheritance in Java

```
interface printable{
void print();
class A6 implements printable{
public void print(){System.out.println("Hello");}
public static void main(String args[]){
A6 obj = new A6();
obj.print();
 }  }
```



Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.

```
interface Printable{
void print();
}
interface Showable{
void show();
}
class A7 implements Printable,Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}
public static void main(String args[]){
A7 obj = new A7();
obj.print();
obj.show();
 } }
```

Q) Multiple inheritance is not supported through class in java but it is possible by interface, why?

As we have explained in the inheritance chapter, multiple inheritance is not supported in case of class. But it is supported in case of interface ***because there is no ambiguity as implementation is provided by the implementation class.*** For example:

```
interface Printable{
void print();
}
interface Showable{
void print();
}
class testinterface1 implements Printable,Showable{
public void print(){System.out.println("Hello");}
```

```
public static void main(String args[]){
testinterface1 obj = new testinterface1();
obj.print();
 }  }
```

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class A, so there is no ambiguity.

Interface inheritance

A class implements interface but one interface extends another interface.

```
interface Printable{
void print();
}
interface Showable extends Printable{
void show();
}
class Testinterface2 implements Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}
public static void main(String args[]){
Testinterface2 obj = new Testinterface2();
obj.print();
obj.show();
 }  }
```

Nested Interface in Java

Note: An interface can have another interface i.e. known as nested interface. We will learn it in detail in the nested classes chapter. For example:

```
interface printable{
 void print();
 interface MessagePrintable{
   void msg();
 } }
```

Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated. But there are many differences between abstract class and interface that are given below.

Abstract class	Interface
can have abstract and non-abstract methods.	can have only abstract methods.
doesn't support multiple inheritance.	supports multiple inheritance.
can have final, non-final, static and non-static variables.	has only static and final variables.
can have static methods, main method and constructor.	can't have static methods, main method or constructor.
can provide the implementation of interface.	can't provide the implementation of abstract class.
The abstract keyword is used to declare abstract class.	interface keyword is used to declare interface.
e.g.: public class Shape{ public abstract void draw();}	e.g.: public interface Drawable{ void draw();}

☞ Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

Example of abstract class and interface in Java

Let's see a simple example where we are using interface and abstract class both.

```
//Creating interface that has 4 methods
interface A{
void a();//bydefault, public and abstract
void b();
void c();
void d();
}
  //Creating abstract class that provides the implementation of one method of A interface
abstract class B implements A{
public void c(){System.out.println("I am C");}
}
//Creating subclass of abstract class, now we need to provide the implementation of rest of the methods class
M extends B{
public void a(){System.out.println("I am a");}
public void b(){System.out.println("I am b");}
public void d(){System.out.println("I am d");}
}
//Creating a test class that calls the methods of A interface
class Test5{
public static void main(String args[]){
A a=new M();
a.a();
a.b();
a.c();
a.d();  }}
```