Introduction to Computer Programming

Using the Java Programming Language

7. Objects & Classes

What are Objects & Classes?

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

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

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

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

Objects

Let us now look deep into what are objects. If we consider the real world we can find many objects around us, Cars, Dogs, Humans, etc. All these objects have a state and behavior.

If we consider a dog, then its state is - name, breed, color, and the behavior is - barking, wagging, running.

If you compare the software object with a real world object, they have very similar characteristics.

Software objects also have a state and behavior. A software object's state is stored in fields and behavior is shown via methods.

So in software development, methods operate on the internal state of an object and the object-to-object communication is done via methods.

Classes

A class is a blue print from which individual objects are created.

A sample of a class is given below:

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

      public void barking(){

      }

      public void hungry(){

      }

      public void sleeping(){

      }
}
```

A class can contain any of the following variable types.

Local variables: Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.

Instance variables: Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.

Class variables: Class variables are variables declared within a class, outside any method, with the static keyword.

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

Constructors

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

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

Example of a constructor is given below:

```
public class Puppy{
      public Puppy(){
      }

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

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

Note: We have two different types of constructors. We are going to discuss constructors in detail in coming units.

Creating an Object

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

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

Declaration: A variable declaration with a variable name with an object type.

Instantiation: The '*new*' keyword is used to create the object.

Initialization: The '*new*' keyword is followed by a call to a constructor. This call initializes the new object.

Example of creating an object is given below:

```
public class Puppy{

      public Puppy(String name){
            // This constructor has one parameter, name.
            System.out.println("Passed Name is :" + name );
      }

      public static void main(String []args){
            // Following statement would create an object myPuppy
            Puppy myPuppy = new Puppy( "tommy" );
      }
}
```

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

```
Passed Name is :tommy
```

Accessing Methods

Methods are accessed via created objects. To access an instance method, the following process should be used:

```
/* First create an object */
ObjectReference = new Constructor();

/* Now you can call a class method as follows */
ObjectReference.MethodName();
```

Example:
This example explains how to access methods of a class:

```
public class Puppy{
      int puppyAge;

      public Puppy(String name){
            // This constructor has one parameter, name.
            System.out.println("Name chosen is :" + name );
      }

      public void setAge( int age ){
            puppyAge = age;
      }

      public int getAge( ){
            System.out.println("Puppy's age is :" + puppyAge );
            return puppyAge;
      }

      public static void main(String []args){
            /* Object creation */
            Puppy myPuppy = new Puppy( "tommy" );

            /* Call class method to set puppy's age */
            myPuppy.setAge( 2 );

            /* Call another class method to get puppy's age */
            myPuppy.getAge( );

            /* You can access instance variable as follows as
well */
            System.out.println("Variable Value :" +
myPuppy.puppyAge );
      }
}
```

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

```
Name chosen is :tommy
Puppy's age is :2
Variable Value :2
```

How to Properly Overload Methods

Method Overloading is a feature that allows a class to have two or more methods having same name, if their argument lists are different. In the last tutorial we discussed constructor overloading that allows a class to have more than one constructors having different argument lists.

Argument lists could differ in:

1. Number of parameters.
2. Data type of parameters.
3. Sequence of Data type of parameters.

Method overloading is also known as **Static Polymorphism**.

Points to note:

1. *Static Polymorhism* is also known as compile time binding or early binding.
2. *Static Binding* happens at compile time. Method overloading is an example of static binding where binding of method call to its definition happens at Compile time.

Method Overloading Examples

As discussed above, method overloading can be done by having different argument list. Lets see examples of each and every case.

Example 1 – Overloading - Different Number of parameters in argument list.

When methods name are same but number of arguments are different.

```
class DisplayOverloading {
     public void disp(char c) {
          System.out.println(c);
     }

     public void disp(char c, int num) {
          System.out.println(c + " "+num);
     }
}

class Sample {
    public static void main(String args[]) {
        DisplayOverloading obj = new DisplayOverloading();
        obj.disp('a');
        obj.disp('a',10);
    }
}
```

Output

```
a
a 10
```

In the above example – method disp() has been overloaded based on the number of arguments – We have two definition of method disp(), one with one argument and another with two arguments.

Example 2: Overloading – Difference in data type of arguments

In this example, method disp() is overloaded based on the data type of arguments – Like example 1 here also, we have two definition of method disp(), one with char argument and another with int argument.

```
class DisplayOverloading2 {
     public void disp(char c) {
         System.out.println(c);
     }

     public void disp(int c) {
        System.out.println(c );
     }
 }

class Sample2 {
     public static void main(String args[]) {
         DisplayOverloading2 obj = new DisplayOverloading2();
         obj.disp('a');
         obj.disp(5);
     }
}
```

Output

```
a
5
```

Example 3: Overloading – Sequence of data type of arguments

Here method disp() is overloaded based on sequence of data type of arguments – Both the methods have different sequence of data type in argument list. First method is having argument list as (char, int) and second is having (int, char). Since the sequence is different, the method can be overloaded without any issues.

```
class DisplayOverloading3 {
    public void disp(char c, int num) {
        System.out.println("I'm the first definition of method disp");
    }

    public void disp(int num, char c) {
        System.out.println("I'm the second definition of method disp"
);
    }
}

class Sample3 {
    public static void main(String args[]) {
        DisplayOverloading3 obj = new DisplayOverloading3();
        obj.disp('x', 51 );
        obj.disp(52, 'y');
    }
}
```

Output

```
I'm the first definition of method disp
I'm the second definition of method disp
```

Creating a Class – Step by Step

This section will guide you through the creation of a class, one step at a time. Many important concepts are gained from this exercise. Take the time to review each of the sections in order.

What concepts you will be learning

While you are working through this guide, keep in mind the following terms and concepts. At the completion of this exercise, you should have a good grasp of the following:

- Designing a Class
- Determining Object Behavior
- Instance Variables
- Implementing Methods
- Constructors
- Using Classes

The vocabulary that you should be familiar with when you are done include:

- Attributes
- Nouns
- Method Calls
- Encapsulation
- Behaviors
- Verbs
- Access Specifier
- Constructor

Designing the class before you begin typing

1. One of the advantages of object-oriented design is it allows a programmer to create a new abstract data type, which is reusable in other situations.

2. When designing a new data type, two components must be identified - attributes and behaviors.

3. Consider the icons used in computer operating systems. The attributes that describe the icon are things like a graphic pattern, colors, size, name, and its position on the screen. Some of its behaviors would include renaming and moving its position.

4. The attributes of an object are the nouns that describe that object. These will become the private data members of a class.

5. The behaviors of an object are the verbs that denote the actions of that object or what it does. These will become the member functions of a class.

Determining Object Behavior

1. In this section, you will learn how to create a simple class that describes the behavior of a bank account. Before you start programming, you need to understand how the objects of your class behave. Operations that can be carried out with a checking account could consist of:

- Accept a deposit
- Process a check
- Get the current balance

2. In Java, these operations are expressed as method calls. For example, assume we have an object checking of type CheckingAccount. The methods that invoke the required behaviors:

```
checking.processDeposit(1000);
checking.processCheck(250);
System.out.println("Balance: " + checking.getBalance());
```

These methods form the behavior of the CheckingAccount class. The behavior is the complete list of the methods that you can apply to objects of a given class. An object of type CheckingAccount can be viewed as a "black box" that can carry out its methods.

3. To construct objects of the CheckingAccount class, it is necessary to declare an object variable:

```
CheckingAccount checking;
```

Object variables such as checking are references to objects. Instead of holding an object itself, a reference variable holds the information necessary to find the object in memory.

4. This object variable checking does not refer to any object at all. An attempt to invoke a method on this variable would cause the compiler to generate an error indicating that the variable had not been initialized. To initialize the variable, it is necessary to create a **new** CheckingAccount object using the *new* operator:

```
checking = new CheckingAccount();
```

This call creates a new object and returns a reference to the newly created object. To use an object, you must assign that reference to an object variable.

5. We will implement the CheckingAccount so that a newly created account has an initial balance of 1000.0 dollars.

```
       // open a new account
      double initialDeposit = 1000.0;
      CheckingAccount checking = new CheckingAccount();

      // set initial balance to 1000.0
      checking.processDeposit(intialDeposit);
```

6. Objects of the CheckingAccount class can be used to carry out meaningful tasks without knowing how the CheckingAccount objects store their data or how the CheckingAccount methods do their work. This is an important aspect of object-oriented programming.

7. Once we understand how to use objects of the CheckingAccount class, it is possible to design a Java class that implements its behaviors. To describe object behavior, you need to implement a class.

```
public class CheckingAccount   {
       // CheckingAccount data

       // CheckingAccount constructors

       // CheckingAccount methods
 }
```

Three methods have already been identified

```
 public class CheckingAccount   {
       // CheckingAccount data
       // CheckingAccount constructors

      public double getBalance()   {
            // method implementation
      }

      public void processDeposit( double amount )   {
      // method implementation
      }

      public void processCheck( double amount )   {
      // method implementation
      }
}
```

8. A method header consists of the following parts:

a. An *access specifier* (such as **public**). The access specifier controls which other methods can call this method. Most methods should be declared as public so all other methods in your program can call them.

b. The *return type* of the method (such as **double** or **void**). The return type is the type of the value that the method computes. For example, in the CheckingAccount class, the getBalance method returns the current account balance which is a floating-point number, so its return type is **double**. The processDeposit and processCheck methods don't return any value. To indicate that a method does not return a value, you use the special type **void**.

c. The name of the method (such as processDeposit).

d. A list of the parameters of the method. The parameters are the input to the method. The processDeposit and processCheck methods each have one parameter, the amount of money to deposit or withdraw. The type of parameter, such as **double**, and name for each parameter, such as amount, must be specified. If a method has no parameters, like getBalance, it is still necessary to supply a pair of parentheses () behind the method name.

9. Once the method header has been specified, the implementation of the method must be supplied in a block that is delimited by braces {...}. The CheckingAccount methods will be implemented later in Section D.

Instance Variables

1. Each object must store its current state. The state is the set of values that describe the object and that influence how an object reacts to method calls. In the case of our checking account objects, the state is the current balance and an account identifier.

2. Each object stores its state in one or more *instance variables*.

Program 3-1

```
public class CheckingAccount   {
       ...
       private double myBalance;
       private String myAccountNumber;
       // CheckingAccount methods
}
```

3. An instance variable declaration consists of the following parts:

a. An access specifier (such as private). Instance variables are generally declared with the access specifier **private**. That means, they can be accessed only by methods of the same class, not by any other method. In particular, the balance variable can be accessed only by the processDeposit, processCheck, and getBalance methods.

b. The type of the variable (such as **double**).

c. The name of the variable (such as myBalance).

4. If instance variables are declared private, then all data access must occur through the public methods. This means that the instance variables of an object are effectively hidden from the programmer who uses a class. They are only of concern to the programmer that implements the class. The process of hiding data is called encapsulation. Although it is possible in Java to define instance variables as **public** (leave them unencapsulated), it is very uncommon in practice. We will make all instance variables private in this course.

5. For example, because the balance instance variable is **private**, the instance variable in other code cannot be accessed:

```
double balance = checking.myBalance;  // compiler Error!
```

However, the public getBalance method to inquire about the balance can be called:

```
double balance = checking.getBalance();  // OK
```

Implementing Methods

1. An implementation must be provided for every method of the class. The implementation for three methods of the CheckingAccount class is given below.

```
public class CheckingAccount {
  private double myBalance;
  private String myAccountNumber;

      public double getBalance() {
         return myBalance;
      }

      public void processDeposit( double amount ) {
         myBalance =  myBalance + amount;
      }

      public void processCheck( double amount ) {
         myBalance =  myBalance – amount;
      }
}
```

2. The implementation of the methods is straightforward. When some amount of money is deposited or withdrawn, the balance increases or decreases by that amount.

3. The getBalance method simply returns the current balance. A **return** statement obtains the value that a method returns and exits the method immediately. The return value becomes the value of the method call expression. The syntax of a **return** statement is:

```
return <expression>;
```

or

```
return;
```

Constructors

1. The final requirement to implement the CheckingAccount class is to define a constructor. The purpose of a constructor is to initialize the instance variables of an object.

```
public class CheckingAccount {
  ...
  public CheckingAccount() {
         myBalance = 0;
         myAccountNumber = "NEW";
  }
  ...
}
```

2. Constructors always have the same name as their class. Similar to methods, constructors are generally declared as public to enable any code in a program to construct new objects of the class. Unlike methods, constructors do not have return types.

3. Constructors are always invoked together with the new operator:

```
new CheckingAccount();
```

The **new** operator allocates memory for the objects, and the constructor initializes it. The value of the new operator is the reference to the newly allocated and constructed object. In most cases, you want to declare and store a reference to an object in an object variable as follows:

```
CheckingAccount checking = new CheckingAccount();
```

4. If you do not initialize an instance variable that is a number, it is initialized automatically to zero. Even though initialization is handled automatically for instance variables, it's a matter of good style to initialize every object's instance variable(s) explicitly.

5. An object variable is initialized to a special value called **null**, which indicates that an object variable does not yet refer to an actual object.

6. Many classes have more than one constructor. For example, you can supply a second constructor for the CheckingAccount class that sets the balance and accountNumber instance variables to initial values, which are the parameters of the constructor:

```
public class CheckingAccount {
  ...
  public CheckingAccount() {
         myBalance = 0;
         myAccountNumber = "NEW";
  }

  public CheckingAccount(double initialBalance, String acctNum){
         myBalance = initialBalance;
         myAccountNumber = acctNum;
  }
  ...
}
```

The second constructor is used if you supply a number and a string as construction parameters.

```
CheckingAccount checking = new CheckingAccount(5000, "A123");
```

7. Note that there are two constructors of the same name. Whenever you have multiple methods (or constructors) with the same name, the name is said to be overloaded. The compiler figures out which one to call by looking at the parameters. For example, if you construct a new checkingAccount object with

```
CheckingAccount checking = new CheckingAccount();
```

then the compiler picks the first constructor. If you construct an object with

```
CheckingAccount checking = new CheckingAccount(5000, "A123");
```

then the compiler picks the second constructor.

8. The implementation of the CheckingAccount class is complete and is given below:

```
public class CheckingAccount {
     private double myBalance;
     private String myAccountNumber;

     public CheckingAccount() {
         myBalance = 0;
         myAccountNumber = "NEW";
     }

     public CheckingAccount(double initialBalance, String acctNum){
         myBalance = initialBalance;
         myAccountNumber = acctNum;
     }

     public double getBalance() {
         return myBalance;
     }

     public void processDeposit(double amount) {
         myBalance =  myBalance + amount;
     }

     public void processCheck(double amount) {
         myBalance =  myBalance – amount;
     }
}
```

Using Classes

1. Using the CheckingAccount class is best demonstrated by writing a program that solves a specific problem. We want to study the following scenario:
An interest bearing checking account is created with a balance of $1,000. For two years in a row, add 2.5% interest. How much money is in the account after two years?

2. Two classes are required: the CheckingAccount class that was developed in the preceding sections, and a second class called CheckingAccountTester. The main method of the CheckingAccountTester class constructs a CheckingAccount object, adds the interest twice, then prints out the balance.

```
public class CheckingAccountTester  {
     public static void main( String[] args ) {
          CheckingAccount checking = new CheckingAccount("A123", 1000);

          final double INTEREST_RATE = 2.5;
          double interest;
          interest = checking.getBalance() * INTEREST_RATE / 100;
          checking.processDeposit(interest);

          System.out.println("Balance after year 1 is $" + checking.getBalance());

          interest = account.getBalance() * INTEREST_RATE / 100;
          checking.processDeposit(interest);

          System.out.println("Balance after year 2 is $" + checking.getBalance());
     }
}
```

3. The classes can be distributed over multiple files or kept together in a single file. If kept together, the class with the main method must be declared as **public**. The **public** attribute cannot be specified for any other class in the same file since a Java source file can contain only one **public** class.

4. Care must be taken to ensure that the name of the file matches the name of the public class. For example, a single file containing both the CheckingAccount class and the CheckingAccountTester class must be contained in a file called CheckingAccountTester.java, not CheckingAccount.java.

Summary / Review

The topics in this lesson are critical in your study of computer science. The concepts of abstraction and object-oriented programming (OOP) will continue to be developed in future lessons. Before you solve the lab exercise, you are encouraged to play with the CheckingAccount class and implement objects using all the behaviors of the class.