Chapter-2

Introducing Classes

2.1 Class Fundamentals

What is a Class?

It defines a new data type. Once defined, this new type can be used to create objects of that type. Thus, **a class is a template for an object**, and an **object is an instance of a class**.

2.1.1 The General Form of a Class

```
class classname
{
type instance-variable1;
type instance-variable2;
 ...

type methodname1(parameter-list)
{
// body of method
}

type methodnameN(parameter-list)
{
// body of method
}

}
```

- A class is declared by use of the **class** keyword
- The data, or variables, defined within a **class** are called *instance variables.*
- Collectively, the methods and variables defined within a class are called *members* of the class.
- Variables defined within a class are called instance variables because each instance or object of the class contains its own copy of these variables.
- Notice that the general form of a class does not specify a **main( )** method. Java classes do not need to have a **main( )** method.
- You only specify one if that class is the starting point for your program. Further, applets don't require a **main( )** method at all.

Note

- C++ programmers will notice that the class declaration and the implementation of the methods are stored in the same place and not defined separately.
- This sometimes makes for very large .java files, since any class must be entirely defined in a single source file.
- This design feature was built into Java because it was felt that in the long run, having specification, declaration, and implementation all in one place makes for code that is easier to maintain

```
class Add
{
   int a,b,sum;
   int sum()
   {
   a=10;     b=20;
   return sum= a+b;
   }
}
```

```
class MainAdd
{
   public static void main(String args[])
   {
      Add obj= new Add();
      System.out.println("sum ="+ obj.sum());
   }
}
```

2.1.2 Declaring a class:

```
Class Box
{
  Double width;
Double height;
Double depth;
}
```

```
// This class declares an object of type Box.
class BoxDemo {
public static void main(String args[]) {
Box mybox = new Box();
double vol;
// assign values to mybox's instance variables
mybox.width = 10;
mybox.height = 20;
mybox.depth = 15;
// compute volume of box
vol = mybox.width * mybox.height * mybox.depth;
System.out.println("Volume is " + vol);
}
}
```

2.1.3 Declaring Objects:

Box mybox = new Box();

OR

1) Box mybox; // declare reference to object

2) mybox = new Box(); // allocate a Box object

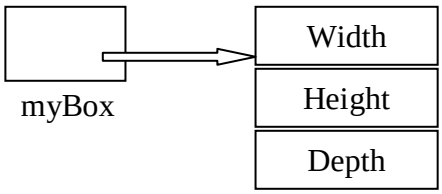
Obtaining objects of a class is a two-step process.

1) First, you must declare a variable of the class type.
   - This variable does not define an object.
   - Instead, it is simply a variable that can *refer to an object.*
2) Second, you must acquire an actual, physical copy of the object and assign it to that variable.
   - You can do this using the new operator.
   - The new operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it

Statement: **Effect**

Box myBox

mybox = new Box();



- The first line declares mybox as a reference to an object of type Box.
- After this line executes, mybox contains the value null, which indicates that it does not yet point to an actual object.
- Any attempt to use mybox at this point will result in a compile-time error.
- The next line allocates an actual object and assigns a reference to it to mybox.
- After the second line executes, you can use mybox as if it were a Box object.
- But in reality, mybox simply holds the memory address of the actual Box object.

Note

- Those readers familiar with C/C++ have probably noticed that object references appear to be similar to pointers.
- This suspicion is, essentially, correct. An object reference is similar to a memory pointer.
- The main difference—and the key to Java's safety—is that you cannot manipulate references as you can actual pointers.
- Thus, you cannot cause an object reference to point to an arbitrary memory location or manipulate it like an integer.

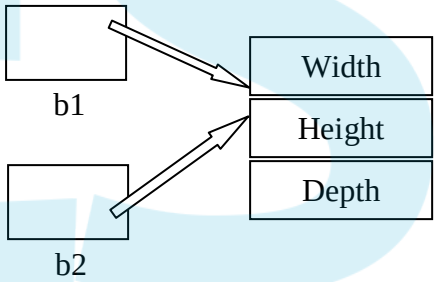
Assigning Object Reference Variables

Object reference variables act differently than you might expect when an assignment takes place.

For example, what do you think the following fragment does?

Box b1 = new Box();

Box b2 = b1;



You might think that b1 and b2 refer to separate and distinct objects.
However, this would be wrong. Instead, after this fragment executes, b1 and b2 will both refer to the *same object.*

Although b1 and b2 both refer to the same object, they are not linked in any other way.

For example, a subsequent assignment to b1 will simply *unhook b1 from the original object* without affecting the object or affecting b2. For example:

```
Box b1 = new Box();
Box b2 = b1;
// ...
b1 = null;
```

Here, b1 has been set to null, but b2 still points to the original object.


Note

- ***When you assign one object reference variable to another object reference variable,*** *you are not creating any copy of the object, actually you are only making a copy of the reference.*

2.1.4 Introduction to methods:

This is the general form of a method:

```
type name(parameter-list)
{
// body of method
}
```

2.1.5 Returning a Value

```
int sum()
{
  return a+b
}
```

There are two important things to understand about returning values:

• The type of data returned by a method must be compatible with the return type specified by the method. For example, if the return type of some method is **boolean,** you could not return an integer.

• The variable receiving the value returned by a method (such as **vol, in this case) must** also be compatible with the return type specified for the method.

2.2 Constructor: Default Constructor

- Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor.
- Constructor is automatically called immediately after the object is created, before the new operator completes.
- Implicit return type of a class' constructor is the class type itself. It is the constructor's job to initialize the internal state of an object

A simple example of using a constructor:

```
class Add {
    int a,b, sum;
Add()
{
   a=10;
   b=10;
}

int sum() {
      return sum=a+b;
   }
 } // end of Add class
```

Constructor

```
class test_class
{
  public static void main(String args[])
  {
    Add obj=new Add();
    System.out.println("sum="+ obj.sum());
  }
}
```

Constructor of Box class

```
class Box
{
    double width;
    double height;
    double depth;
                // This is the constructor for Box.
                Box() {
                System.out.println("Constructing Box");
                width = 10;
                height = 10;
                depth = 10;
                }
  // compute and return volume
  double volume()
  {
   return width * height * depth;
  }
}
```

2.2.1 Parameterized Constructors

- Box( ) constructor in the previous example does initialize a Box object, but it is not very useful as all boxes have the same dimensions.
- What is needed is a way to construct Box objects of various dimensions. The easy solution is to add parameters to the constructor.

A Simple example showing the use of Parameterized constructor:

```
class AddClass
{
   int a,b, sum;
AddClass()
{
}
AddClass(int x, int y)
{
  a=x;
  b=y;
}

int sum() {
     return sum=a+b;
  }
 }
```

```
class TestClass
{
  public static void main(String args[])
 {
    AddClass obj=new AddClass(10,20);
System.out.println("sum="+ obj.sum(10,20));
}
}
```

2.3 this Keyword:

- Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the "**this**" keyword.
- "**this**" can be used inside any method to refer to the current object. That is, this is always a reference to the object on which the method was invoked.
- You can use "**this**", anywhere a reference to an object of the current class type is permitted.

```
// A redundant use of this.
Box(double w, double h, double d)
{
  this.width = w;
  this.height = h;
  this.depth = d;
}
```

The use of this is redundant, but perfectly correct. Inside Box( ), this will always refer to the invoking object.
While it is redundant in this case, this is useful in Instance variable hiding

2.3.1 Instance Variable Hiding

- **When a local variable has the same name as an instance variable, the local variable hides the instance variable**.
- While it is usually easier to simply use different names, there is another way around this situation. Because **this** lets you refer directly to the object, you can use it to resolve any name space collisions that might occur between instance variables and local variables.

```
class ThisClass
{
   int a,b;
   ThisClass(int a, int b)
   {
     a=a;
     b=b;
   }
}
```

```
class MainTest
{
  public static void main(String a[])
  {
     ThisClass obj= new ThisClass(10,20);
     System.out.println("obj.a= "+ obj.a);
     System.out.println("obj.b= "+ obj.b);
  }
}
```

Output:
obj.a= 0
obj.b= 0

Here local variable a and b hides the visibility of instance variables a and b
Hence the output of instance variable a and b is 0, as the local variable dies after the end of method.

Solution to this problem is achieved using this keyword:

```
class ThisClass
{
   int a,b;
   ThisClass (int a, int b)
   {
     this.a=a;
     this.b=b;
   }
}
```

```
class MainTest
{
  public static void main(String a[])
  {
     ThisClass obj= new ThisClass(10,20);
     System.out.println("obj.a= "+ obj.a);
     System.out.println("obj.b= "+ obj.b);
  }
}
```

Output:
obj.a= 10
obj.b= 20

A word of caution:

- *The use of this in such a context can sometimes be confusing, and some programmers are careful not to use local variables and formal parameter names that hide instance variables. Of course, other programmers believe the contrary—that it is a good convention to use the same names for clarity, and use this to overcome the instance variable hiding. It is a matter of taste which approach you adopt.*
- *Although this is of no significant value in the examples just shown, it is very useful in certain situations.*

2.4 Garbage Collection

- Since objects are dynamically allocated by using the new operator, you might be wondering how such objects are destroyed and their memory released for later reallocation.
- In some languages, such as C++, dynamically allocated objects must be manually released by use of a delete operator. Java takes a different approach; it handles deallocation for you automatically. The technique that accomplishes this is called **garbage collection**.
- It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.
- There is no explicit need to destroy objects as in C++.
- Garbage collection only occurs sporadically (if at all) during the execution of your program.
- It will not occur simply because one or more objects exist that are no longer used.

2.5 The finalize( ) Method (a bit similar to destructor method of C++)

- Sometimes an object will need to perform some action when it is destroyed.
- For example, if an object is holding some non-Java resource such as a file handle or character font, then you might want to make sure these resources are freed before an object is destroyed.
- To handle such situations, Java provides a mechanism called finalization.
- By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.

```
protected void finalize( )
{
// finalization code here
}
```

- Here, the keyword **protected is a specifier that prevents access to finalize( ) by code defined** outside its class.
- It is important to understand that **finalize( ) is only called just prior to garbage collection.**
- It is not called when an object goes out-of-scope, for example. This means that you cannot know when—or even if—**finalize( ) will be executed.**
- Therefore, your program should provide other means of releasing system resources, etc., used by the object. It must not rely on **finalize( )** for normal program operation.

2.6 Overloading Methods

- In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different.
- When this is the case, the methods are said to be overloaded, and the process is referred to as method overloading.
- Method **overloading is one of the ways that Java supports polymorphism**.
- When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call.
- Thus, **overloaded methods must differ in the type and/or number of their parameters**.
- While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call

```
// Demonstrate method overloading.
class OverloadDemo {
void test() {
System.out.println("No parameters");
}
void test(int a) {
System.out.println("a: " + a);
}
void test(int a, int b) {
System.out.println("a and b: " + a + " " + b);
}
double test(double a) {
System.out.println("double a: " + a);
return a*a;
}
}
```

```
class Overload {
public static void main(String args[]) {
OverloadDemo ob = new OverloadDemo();
double result;
// call all versions of test()
ob.test();
ob.test(10);
ob.test(10, 20);
result = ob.test(123.25);
System.out.println("Result of ob.test(123.25): " +
result);
}
}
```

Automatic type conversions apply to overloading

```
class OverloadDemo {
void test() {
System.out.println("No parameters");
}
void test(int a, int b)
{
System.out.println("a and b: " + a + " " + b);
}
```

```
void test(double a)
{
System.out.println("Inside test(double) a: " + a);
} } // end of OverloadDemo class
class Overload {
public static void main(String args[]) {
OverloadDemo ob = new OverloadDemo();
Int i = 88;
ob.test();
ob.test(10, 20);
ob.test(i); // this will invoke test(double)
ob.test(123.2); // this will invoke test(double)
}
}
```

Example Showing Overloading of Constructors

```
class Add_Class
{
   int a,b, sum;
add_class(int x,){
 a=x;
 b=x;
}
add_class(int x, int y){
   a=x;
   b=y;
}
add_class(){
    x=y=0;
}
```

```
int sum()
    {
      return sum=a+b;
   }
} // end of Add_Class
class test_class
{
  public static void main(String args[])
 {
    Add_Class obj=new Add_Class();
    System.out.println("sum="+ obj.sum(10,20));
}
}
```

2.7 Using Objects as Parameters

```
class Test {
int a, b;
Test(int i, int j) {
a = i;
b = j;
}
// return true if o is equal to the invoking object
boolean equals(Test o) {
if(o.a == a && o.b == b) return true;
else return false;
}
}
```

```
class PassOb {
public static void main(String args[]) {
Test ob1 = new Test(100, 22);
Test ob2 = new Test(100, 22);
Test ob3 = new Test(-1, -1);
System.out.println("ob1 == ob2: " + ob1.equals(ob2));
System.out.println("ob1 == ob3: " + ob1.equals(ob3));
}
}
```

This program generates the following output:
ob1 == ob2: true
ob1 == ob3: false

A Closer Look at Argument Passing

Pass by value

```
// Primitive types are passed by value.
class Test {
void meth(int i, int j) {
i *= 2;
j /= 2;
}
}
```

```
class CallByValue {
public static void main(String args[]) {
Test ob = new Test();
int a = 15, b = 20;
System.out.println("a and b before call: " + a + " " + b);
ob.meth(a, b);
System.out.println("a and b after call: " + a + " " + b);
}
}
```

The output from this program is shown here:
a and b before call: 15 20
a and b after call: 15 20

Pass by reference

```
// Objects are passed by reference.
class Test {
int a, b;
Test(int i, int j) {
a = i;
b = j;
}
// pass an object
void meth(Test o) {
o.a *= 2;
o.b /= 2;
}
}
```

```
class CallByRef
 {
public static void main(String args[]) {
Test ob = new Test(15, 20);
System.out.println("ob.a and ob.b before call: " + ob.a + ob.b);
ob.meth(ob);
System.out.println("ob.a and ob.b after call: " + ob.a + ob.b);
}
}
```

This program generates the following output:
ob.a and ob.b before call: 15 20
ob.a and ob.b after call: 30 10

REMEMBER

When a primitive type is passed to a method, it is done by use of call-by-value. Objects are implicitly passed by use of call-by-reference.

2.8 Returning Objects:

- A method can return any type of data, including class types that you create.

// Returning an object.

```
class Test {
int a;
Test(int i) {
a = i;
}
Test incrByTen() {
Test temp = new Test(a+10);
return temp;
 }
}
```

```
class RetOb {
public static void main(String args[]) {
Test ob1 = new Test(2);
Test ob2;
ob2 = ob1.incrByTen();
System.out.println("ob1.a: " + ob1.a);
System.out.println("ob2.a: " + ob2.a);
ob2 = ob2.incrByTen();
System.out.println("ob2.a after second increase: "+ ob2.a);
}
}
```


2.9 Recursion

- Java supports recursion. Recursion is the process of defining something in terms of itself. As it relates to Java programming, recursion is the attribute that allows a method to call itself. A method that calls itself is said to be recursive.

Program showing use of Recursion:

```
// A simple example of recursion.
class Factorial {
// this is a recursive method
int fact(int n) {
int result;
if(n==1) return 1;
result = fact(n-1) * n;
return result;
}
}
```

```
class Recursion {
public static void main(String args[]) {
Factorial f = new Factorial();
System.out.println("Factorial of 3 is " + f.fact(3));
System.out.println("Factorial of 4 is " + f.fact(4));
System.out.println("Factorial of 5 is " + f.fact(5));
}
}
```

The output from this program is shown here:
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120

2.10 Access controls:

Public: visible to all the classes and packages:

Default: visible to all the classes of that package but not visible to other packages

Private: visible to only that class

Protected: (range between public and default)visible to all the classes and subclasses in same package and all inherited classes from any package

Private protected: (lies between private and protected) visibility to only inherited classes but from any package.

2.11 Understanding Static:

- There will be times when you will want to define a class member that will be used independently of any object of that class.
- It is possible to create a member that can be used by itself, without reference to a specific instance.
- To create such a member, precede its declaration with the keyword static. When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object.
- You can declare both methods and variables to be static. The most common example of a static member is main( ). main( ) is declared as static because it must be called before any objects exist.
- **Instance variables declared as static are, essentially, global variables.**
- **When objects of its class are declared, no copy of a static variable is made. Instead, all instances of the class share the same static variable**.

Methods declared as static have several restrictions:

- They can only call other static methods.
- They must only access static data.
- They cannot refer to this or super in any way. (The keyword super relates to inheritance and is described in the next chapter.)
- If you need to do computation in order to initialize your static variables, you can declare a static block which gets executed exactly once, when the class is first loaded.

The following example shows a class that has a static method, some static variables, and a Static initialization block:

```
// Demonstrate static variables, methods, and blocks.
class UseStatic {
static int a = 3;
static int b;
static void meth(int x) {
System.out.println("x = " + x);
System.out.println("a = " + a);
System.out.println("b = " + b);
}
static {
System.out.println("Static block initialized.");
b = a * 4;
}
   public static void main(String args[])
   {
     meth(42);
   }
}  // end of UseStatic class
```

As soon as the UseStatic class is loaded, all of the static statements are run. First, a is set to 3, then the static block executes (printing a message), and finally, b is initialized to a * 4 or 12. Then main( ) is called, which calls meth( ), passing 42 to x. The three println( ) statements refer to the two static variables a and b, as well as to the local variable x.

Note: It is illegal to refer to any instance variables inside of a **static** method. Here is the output of the program

Outside of the class in which they are defined, **static** methods and variables can be used independently of any object. To do so, you need only specify the name of their class followed by the dot operator.

For example, if you wish to call a **static** method from outside its class, you can do so using the following general form:

classname.method( )

Here, *classname* is the name of the class in which the **static** method is declared. As you can see, this format is similar to that used to call non-**static** methods through object reference variables. A **static** variable can be accessed in the same way by use of the dot operator on the name of the class. This is how Java implements a controlled version of global functions and global variables.

```
class StaticDemo {
  static int a = 42;
  static int b = 99;
  static void callme()
  {
  System.out.println("a = " + a);
  }
} // end of StaticDemo class
```

```
class StaticByName {
 public static void main(String args[])
   {
    StaticDemo.callme();
    System.out.println("b = " + StaticDemo.b);
   }
}
```

2.12 Introducing Nested and Inner Classes

- It is possible to define a class within another class; such classes are known as nested classes.
- The scope of a nested class is bounded by the scope of its enclosing class.
- Thus, if class B is defined within class A, then B is known to A, but not outside of A.
- A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class.

There are two types of nested classes: static and non-static.

- A static nested class is one which has the static modifier applied. Because it is static, it must access the members of its enclosing class through an object. That is, it cannot refer to members of its enclosing class directly. Because of this restriction, static nested classes are hardly used.
- The most important type of nested class is the inner class. **An inner class is non-static nested class**. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do. Thus, an inner class is fully within the scope of its enclosing class.

The following program illustrates how to define and use an inner class. The class named Outer has one instance variable named outer_x, one instance method named test( ), and defines one inner class called Inner.

```
class Outer {
int outer_x = 100;
void test() {
Inner inner = new Inner();
inner.display();
}
// this is an inner class
class Inner {
void display() {
System.out.println("display: outer_x = " + outer_x);
} }   // end of Inner class
} // end of outer class
```

```
class InnerClassDemo
{
public static void main(String args[])
{
Outer outer = new Outer();
outer.test();
}
}
```

Output from this application is shown here:

```
display: outer_x = 100
```

2.13 Varargs: Variable-Length Arguments :

- A method that takes a variable number of arguments is called varargs method.
- Ex: system.out.prinln(); method takes variable number of arguments.
- Previously before introducing this concept people use to go with array concept:

Example program showing use of variable length arguments using Array:

```
// Use an array to pass a variable number of
// arguments to a method. This is the old-style
// approach to variable-length arguments.
class PassArray
{
static void vaTest(int v[])
{
System.out.print("Number of args: " + v.length +" Contents: ");
for(int x : v)
System.out.print(x + " ");
System.out.println();
}
```

```
public static void main(String args[])
{
// Notice how an array must be created to
// hold the arguments.
int n1[] = { 10 };
int n2[] = { 1, 2, 3 };
int n3[] = { };
vaTest(n1); // 1 arg
vaTest(n2); // 3 args
vaTest(n3); // no args
}
}
```

output of the program is:

```
Number of args: 1 Contents: 10
Number of args: 3 Contents: 1 2 3
Number of args: 0 Contents:
```

Example program showing use of variable length in JAVA:

```
// Demonstrate variable-length arguments.
class VarArgs {
// vaTest() now uses a vararg.
static void vaTest(int ... v)
{
System.out.print("Number of args: " + v.length + " Contents: ");
for(int x : v)
System.out.print(x + " ");
System.out.println();
}
```

```
public static void main(String args[])
{
// Notice how vaTest() can be called
//with a variable number of arguments.
vaTest(10); // 1 arg
vaTest(1, 2, 3); // 3 args
vaTest(); // no args
}
}
```

Note: Loop is called: for each loop


Note

- Remember, the varargs parameter must be last. For example, the following declaration is incorrect:
- int doIt(int a, int b, double c, int ... vals, boolean stopFlag) { // Error!
- **There is one more restriction to be aware of:** there must be only one varargs parameter. For example, this declaration is also invalid:
- int doIt(int a, int b, double c, int ... vals, double ... morevals) { // Error!
- The attempt to declare the second varargs parameter is illegal.

Overloading Vararg Methods:

You can have multiple methods with same name but different type of variable arguments:

Ex:

```
Static void add(int … v)
{
}
 static void add(float … v)
{
}
```


Note

- ***A varargs method can also be overloaded by a non-varargs method.***
- ***For example, vaTest(int x)*** *is a valid overload of* ***vaTest( ) in the foregoing program. This version is invoked only when one int argument is present.***
- ***When two or more int arguments are passed, the varargs version vaTest(int…v) is used.***

Varargs and Ambiguity:

```
static void vaTest(int ... v)
{
}
static void vaTest(boolean ... v)
{
}
```

Calling of function
vaTest(1, 2, 3); // OK
vaTest(true, false, false); // OK
vaTest(); // Error: Ambiguous!
which function to call, is it Boolean or int


```
static void vaTest(int ... v) { // ...
static void vaTest(int n, int ... v) { // ...
vaTest(1)// ambiguity
```