**UNIT – 2**

1. Classes: A ‘*class*’ is a new data type and the basic point to be learned in OOP. This new ‘type’ can be used to create ‘objects’ of that type. A class is a template for an object and an object is an instance of a class. Note that the words object and instance are used interchangeably.
2. A class is declared by the use of the ‘class’ keyword.

Class declaration

class classname {

type instance-variable1;

type instance-variable2;

// ...

type instance-variableN;

type methodname1(parameter-list) {

// body of method

}

type methodname2(parameter-list) {

// body of method

}

// ...

type methodnameN(parameter-list) {

// body of method

}

}

1. The data or variables defined within a class are called instance variables because each instance of the class (i.e., objects) contains their own copy of these variables. The code is contained within ‘methods’. Collectively, methods and variables defined within a class are called members of the class.
2. All methods have the same general form as main(). Note that many methods will not be specified as public or static. Applets don’t require a main() method at all!!
3. Example:

class Box {

double width;

double height;

double depth;

}

A class as above defines a new data type called Box, which is used to declare objects of type Box. Note that only a template is created here but not an actual object.

To actually create a Box object, you will use a statement like the following:

Box mybox = new Box(); // create a Box object called mybox

After this statement executes, mybox will be an instance of Box. Thus, it will have “physical” reality.

Each time you create an instance of a class, you are creating an object that contains its own copy of each instance variable defined by the class. Thus, every Box object will contain its own copies of the instance variables width, height, and depth. To access these variables, use the dot (.) operator.

Ex: mybox.width = 100;

1. Example Program:

/\* A program that uses the Box class.

Call this file BoxDemo.java

\*/

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);

}

}

1. Example program with two objects:

// This program declares two Box objects.

class Box {

double width;

double height;

double depth;

}

class BoxDemo2 {

public static void main(String args[]) {

Box mybox1 = new Box();

Box mybox2 = new Box();

double vol;

// assign values to mybox1's instance variables

mybox1.width = 10;

mybox1.height = 20;

mybox1.depth = 15;

/\* assign different values to mybox2's

instance variables \*/

mybox2.width = 3;

mybox2.height = 6;

mybox2.depth = 9;

// compute volume of first box

vol = mybox1.width \* mybox1.height \* mybox1.depth;

System.out.println("Volume is " + vol);

// compute volume of second box

vol = mybox2.width \* mybox2.height \* mybox2.depth;

System.out.println("Volume is " + vol);

}

}

1. When we create a class, we are creating a new data type. It can used to declare objects of that type. However, obtaining objects of a class is a two-step process.

First, we 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. Next we acquire an actual,physical copy of the object and assign it to that variable.

This can be done using the *new* operator. The new operator dynamically allocates (at run time) memory for an object and returns a reference to it. This reference is the address in memory of the object allocated by new. This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated.

1. Box mybox = new Box();

This statement combines the two steps just described. It can be rewritten like this to show each step more clearly:

Box mybox; // declare reference to object

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

The first line declares mybox as a reference to an object of type Box. After this lineexecutes, mybox contains the value null, which indicates that it does not yet point to anactual object. Any attempt to use mybox at this point will result in a compile-time error. Thenext line allocates an actual object and assigns a reference to it to mybox. After the secondline executes, you can use mybox as if it were a Box object. But in reality, mybox simplyholds the memory address of the actual Box object. The effect of these two lines of codeis depicted in the Figure below.

****

 **Figure 2.1: Object Representation**

1. It is important to understand that *new* allocates memory for an object during run time. The advantage of this approach is that your program can create as many or as few objects as it needs during the execution of your program. However, since memory is finite, it is possible that new will not be able to allocate memory for an object because insufficient memory exists. If this happens, a run-time exception will occur.
2. Assigning object reference variables: Consider the fragment below:

Box b1 = new Box();

Box b2 = b1;

It might be thought that b2 is being assigned a reference to a copy of the object referred to by b1. Through this, we 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 thesame object. The assignment of b1 to b2 did not allocate any memory or copy any part ofthe original object. It simply makes b2 refer to the same object as does b1. Thus, anychanges made to the object through b2 will affect the object to which b1 is referring, sincethey are the same object.

****

 **Figure 2.2: Object Reference**

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 objectwithout 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.

1. Methods**:** This is the general form of a method:

*type name*(*parameter-list*) {

// body of method

}

Here, *type* specifies the type of data returned by the method. This can be any valid type, including class types that you create. If the method does not return a value, its return type must be void. The name of the method is specified by *name*. This can be any legal identifier but not those that are already used by other items within the current scope. The *parameter-list* is a sequence of type and identifier pairs separated by commas. Parameters are variables that receive the value of the arguments passed to the method when it is called.

NOTE: If the method has no parameters, then the parameter list will be empty.

Methods that have a return type other than void return a value to the calling routine using the following form of the return statement:

return *value*;

Here, *value* is the value returned.

1. Adding a method to the ‘Box’ class:

// This program includes a method inside the box class.

class Box {

double width;

double height;

double depth;

// display volume of a box

void volume() {

System.out.print("Volume is ");

System.out.println(width \* height \* depth);

}

}

class BoxDemo3 {

public static void main(String args[]) {

Box mybox1 = new Box();

Box mybox2 = new Box();

// assign values to mybox1's instance variables

mybox1.width = 10;

mybox1.height = 20;

mybox1.depth = 15;

/\* assign different values to mybox2's

instance variables \*/

mybox2.width = 3;

mybox2.height = 6;

mybox2.depth = 9;

// display volume of first box

mybox1.volume();

// display volume of second box

mybox2.volume();

}

}

1. Consider the following lines:

mybox1.volume();

mybox2.volume();

The first line here invokes the volume( ) method on mybox1. That is, it calls volume( ) relative to the mybox1 object, using the object’s name followed by the dot operator. Thus, the call to mybox1.volume( ) displays the volume of the box defined by mybox1, and the call to mybox2.volume( ) displays the volume of the box defined by mybox2. Each time volume( ) is invoked, it displays the volume for the concerned box.

1. When an instance variable is accessed by code that is not part of the class in which that instance variable is defined, it must be done through an object, by use of the dot operator. However, when an instance variable is accessed by code that is part of the **s**ame class as the instance variable, that variable can be referred to directly. The same thing applies to methods.
2. Example for returning a value:

// Now, volume() returns the volume of a box.

class Box {

double width;

double height;

double depth;

// compute and return volume

double volume() {

return width \* height \* depth;

}

}

class BoxDemo4 {

public static void main(String args[]) {

Box mybox1 = new Box();

Box mybox2 = new Box();

double vol;

// assign values to mybox1's instance variables

mybox1.width = 10;

mybox1.height = 20;

mybox1.depth = 15;

/\* assign different values to mybox2's

instance variables \*/

mybox2.width = 3;

mybox2.height = 6;

mybox2.depth = 9;

// get volume of first box

vol = mybox1.volume();

System.out.println("Volume is " + vol);

// get volume of second box

vol = mybox2.volume();

System.out.println("Volume is " + vol);

}

}

1. Method that takes Parameters:

// This program uses a parameterized method.

class Box {

double width;

double height;

double depth;

// compute and return volume

double volume() {

return width \* height \* depth;

}

// sets dimensions of box

void setDim(double w, double h, double d) {

width = w;

height = h;

depth = d;

}

}

class BoxDemo5 {

public static void main(String args[]) {

Box mybox1 = new Box();

Box mybox2 = new Box();

double vol;

// initialize each box

mybox1.setDim(10, 20, 15);

mybox2.setDim(3, 6, 9);

// get volume of first box

vol = mybox1.volume();

System.out.println("Volume is " + vol);

// get volume of second box

vol = mybox2.volume();

System.out.println("Volume is " + vol);

}

}

1. **Constructors**: Because the requirement for initialization is so common, Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor.
2. A *constructor* initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method. The constructor is automatically called immediately after the object is created, before the new operator completes.
3. Constructors have no return type, not even void. This is because the 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 so that the code creating an instance will have a fully initialized, usable object immediately.
4. Example:

/\* Here, Box uses a constructor to initialize the dimensions of a box. \*/

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;

}

}

class BoxDemo6 {

public static void main(String args[]) {

// declare, allocate, and initialize Box objects

Box mybox1 = new Box();

Box mybox2 = new Box();

double vol;

// get volume of first box

vol = mybox1.volume();

System.out.println("Volume is " + vol);

// get volume of second box

vol = mybox2.volume();

System.out.println("Volume is " + vol);

}

}

1. Constructors with different parameters are needed obviously. For this requirements, parameterised constructors can be used.

/\* Here, Box uses a parameterized constructor to

initialize the dimensions of a box.

\*/

class Box {

double width;

double height;

double depth;

// This is the constructor for Box.

Box(double w, double h, double d) {

width = w;

height = h;

depth = d;

}

// compute and return volume

double volume() {

return width \* height \* depth;

}

}

class BoxDemo7 {

public static void main(String args[]) {

// declare, allocate, and initialize Box objects

Box mybox1 = new Box(10, 20, 15);

Box mybox2 = new Box(3, 6, 9);

double vol;

// get volume of first box

vol = mybox1.volume();

System.out.println("Volume is " + vol);

// get volume of second box

vol = mybox2.volume();

System.out.println("Volume is " + vol);

}

}

1. **The ‘*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.
2. **Garbage Collection**: Since objects are dynamically allocated by using the *new* operator, it is important to know how such objects are destroyed and their memory released for later reallocation. Java handles de-allocation for you automatically. The technique that accomplishes this is called garbage collection. 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. Garbage collection only occurs periodically during the execution of the program. It will not occur simply because one or more objects exist that are no longer used.

Furthermore, different Java run-time implementations will take varying approaches to garbage collection, but for the most part, we don’t have to think about it while writing programs.

1. **Method Overloading**: 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.

// Demonstrate method overloading.

class OverloadDemo {

void test() {

System.out.println("No parameters");

}

// Overload test for one integer parameter.

void test(int a) {

System.out.println("a: " + a);

}

// Overload test for two integer parameters.

void test(int a, int b) {

System.out.println("a and b: " + a + " " + b);

}

// Overload test for a double parameter

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);

}

}

This program generates the following output:

No parameters

a: 10

a and b: 10 20

double a: 123.25

Result of ob.test(123.25): 15190.5625

1. 'test( )' is overloaded four times. The first version takes no parameters, the second takes one integer parameter, the third takes two integer parameters, and the fourth takes one double parameter. The fact that the fourth version of test( ) also returns a value is of no consequence relative to overloading, since return types do not play a role in overload resolution.

When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method’s parameters. However, this match need not always be exact. In some cases, Java’s automatic type conversions can play a role in overload resolution.

1. Example:

// Automatic type conversions apply to overloading.

class OverloadDemo {

void test() {

System.out.println("No parameters");

}

// Overload test for two integer parameters.

void test(int a, int b) {

System.out.println("a and b: " + a + " " + b);

}

// Overload test for a double parameter

void test(double a) {

System.out.println("Inside test(double) a: " + a);

}

}

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)

}

}

This program generates the following output:

No parameters

a and b: 10 20

Inside test(double) a: 88

Inside test(double) a: 123.2

1. Method overloading supports polymorphism because it is one way that Java implements

the “one interface, multiple methods” paradigm.

In languages that do not support method overloading, each method must be given a unique name. However, frequently you will want to implement essentially the same method for different types of data.

Consider the absolute value function. In languages that do not support overloading, there are usually three or more versions of this function, each with a slightly different name. For instance, in C, the function abs( ) returns the absolute value of an integer, labs( ) returns the absolute value of a long integer, and fabs( ) returns the absolute value of a floating-point value.

Since C does not support overloading, each function has to have its own name, even though all three functions do essentially the same thing. This makes the situation more complex, conceptually, than it actually is. Although the underlying concept of each function is the same, you still have three names to remember.

This situation does not occur in Java, because each absolute value method can use the same name. Indeed, Java’s standard class library includes an absolute value method, called abs( ). This method is overloaded by Java’s Math class to handle all numeric types. Java determines which version of abs( ) to call based upon the type of argument.

1. **Constructor Overloading**: In addition to overloading normal methods, you can also overload constructor methods. In fact, for most real-world classes that you create, overloaded constructors will be the norm, not the exception. To understand why, let’s return to the Box class.
2. Example:

/\* Here, Box defines three constructors to initialize

the dimensions of a box various ways.

\*/

class Box {

double width;

double height;

double depth;

// constructor used when all dimensions specified

Box(double w, double h, double d) {

width = w;

height = h;

depth = d;

}

// constructor used when no dimensions specified

Box() {

width = -1; // use -1 to indicate

height = -1; // an uninitialized

depth = -1; // box

}

// constructor used when cube is created

Box(double len) {

width = height = depth = len;

}

// compute and return volume

double volume() {

return width \* height \* depth;

}

}

class OverloadCons {

public static void main(String args[]) {

// create boxes using the various constructors

Box mybox1 = new Box(10, 20, 15);

Box mybox2 = new Box();

Box mycube = new Box(7);

double vol;

// get volume of first box

vol = mybox1.volume();

System.out.println("Volume of mybox1 is " + vol);

// get volume of second box

vol = mybox2.volume();

System.out.println("Volume of mybox2 is " + vol);

// get volume of cube

vol = mycube.volume();

System.out.println("Volume of mycube is " + vol);

}

}

The output produced by this program is shown here:

Volume of mybox1 is 3000.0

Volume of mybox2 is -1.0

Volume of mycube is 343.0

As it can be seen, the proper overloaded constructor is called based upon the parameters specified when new is executed.

1. **‘static’ Keyword**: A class member must be accessed only in conjunction with an object of its class. However, 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.

1. 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);

}

}

1. **Command line Arguments**: A command-line argument is the information that directly follows the program’s name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy—they are stored as strings in a String array passed to the args parameter of main( ). The first command-line argument is stored at args[0], the second at args[1], and so on.

// Display all command-line arguments.

class CommandLine {

public static void main(String args[]) {

for(int i=0; i<args.length; i++)

System.out.println("args[" + i + "]: " + args[i]);

}

}

Try executing this program, as shown here:

java CommandLine this is a test 100 -1

When you do, you will see the following output:

args[0]: this

args[1]: is

args[2]: a

args[3]: test

args[4]: 100

args[5]: -1

1. **Arrays**: An array is a group of like-typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index. Arrays offer a convenient means of grouping related information.
2. **One-Dimensional Arrays**: A one-dimensional array is, essentially, a list of like-typed variables. To create an array, you first must create an array variable of the desired type.

The general form of a one-dimensional array declaration is

*type var-name*[ ];

Here, *type* declares the element type (also called the base type) of the array. The element type determines the data type of each element that comprises the array. Thus, the element type for the array determines what type of data the array will hold.

Ex: int month\_days[];

The value of month\_days is set to null, which represents an array with no value. To link month\_days with an actual, physical array of integers, you must allocate one using new and assign it to month\_days. ‘new’ is a special operator that allocates memory.

The general form of **new** as it applies to one-dimensional arrays appears as follows:

*array-var* = new *type* [*size*];

This example allocates a 12-element array of integers and links them to **month\_days**:

month\_days = new int[12];

After this statement executes, **month\_days** will refer to an array of 12 integers. Further, all elements in the array will be initialized to zero.

All array indexes start at zero. For example, this statement assigns the value 28 to the second element of **month\_days**:

month\_days[1] = 28;

1. Example:

// Demonstrate a one-dimensional array.

class Array {

public static void main(String args[]) {

int month\_days[];

month\_days = new int[12];

month\_days[0] = 31;

month\_days[1] = 28;

month\_days[2] = 31;

month\_days[3] = 30;

month\_days[4] = 31;

month\_days[5] = 30;

month\_days[6] = 31;

month\_days[7] = 31;

month\_days[8] = 30;

month\_days[9] = 31;

month\_days[10] = 30;

month\_days[11] = 31;

System.out.println("April has " + month\_days[3] + " days.");

}

}

1. An *array initializer* is a list of comma-separated expressions surrounded by curly braces. The commas separate the values of the array elements. The array will automatically be created large enough to hold the number of elements you specify in the array initializer. There is no need to use **new**.

// An improved version of the previous program.

class AutoArray {

public static void main(String args[]) {

int month\_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31,

30, 31 };

System.out.println("April has " + month\_days[3] + " days.");

}

}

1. **Multidimensional Arrays**: In Java, *multidimensional arrays* are actually arrays of arrays. These, as you might expect, look and act like regular multidimensional arrays. However, as you will see, there are a couple of subtle differences. To declare a multidimensional array variable, specify each additional index using another set of square brackets.
2. Example:

// Demonstrate a two-dimensional array.

class TwoDArray {

public static void main(String args[]) {

int twoD[][]= new int[4][5];

int i, j, k = 0;

for(i=0; i<4; i++)

for(j=0; j<5; j++) {

twoD[i][j] = k;

k++;

}

for(i=0; i<4; i++) {

for(j=0; j<5; j++)

System.out.print(twoD[i][j] + " ");

System.out.println();

}

}

}

This program generates the following output:

0 1 2 3 4

5 6 7 8 9

10 11 12 13 14

15 16 17 18 19

1. When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension. You can allocate the remaining dimensions separately.



 **Figure 2.3: Multidimensional Arrays**

1. // Manually allocate differing size second dimensions.

class TwoDAgain {

public static void main(String args[]) {

int twoD[][] = new int[4][];

twoD[0] = new int[1];

twoD[1] = new int[2];

twoD[2] = new int[3];

twoD[3] = new int[4];

int i, j, k = 0;

for(i=0; i<4; i++)

for(j=0; j<i+1; j++) {

twoD[i][j] = k;

k++;

}

for(i=0; i<4; i++) {

for(j=0; j<i+1; j++)

System.out.print(twoD[i][j] + " ");

System.out.println();

}

}

}

This program generates the following output:

0

1 2

3 4 5

1. 7 8 9
2. It is possible to initialize multidimensional arrays. To do so, simply enclose each dimension’s initializer within its own set of curly braces.

// Initialize a two-dimensional array.

class Matrix {

public static void main(String args[]) {

double m[][] = {

{ 0\*0, 1\*0, 2\*0, 3\*0 },

{ 0\*1, 1\*1, 2\*1, 3\*1 },

{ 0\*2, 1\*2, 2\*2, 3\*2 },

{ 0\*3, 1\*3, 2\*3, 3\*3 }

};

int i, j;

for(i=0; i<4; i++) {

for(j=0; j<4; j++)

System.out.print(m[i][j] + " ");

System.out.println();

}

}

}

1. **Nested Classes**: We can 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 does not exist independently of A. A nested class that is declared directly within its enclosing class scope is a member of its enclosing class. It is also possible to declare a nested class that is local to a block.

There are two types of nested classes: *static* and *non-static*. A static nested class is one that has the **static** modifier applied. Since it is static, it must access the non-static members of its enclosing class through an object. It cannot refer to non-static members of its enclosing class directly. Static nested classes are not used much.

1. The most important type of nested class is the *inner* class. An inner class is a 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.

// Demonstrate an inner class.

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);

}

}

}

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

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*