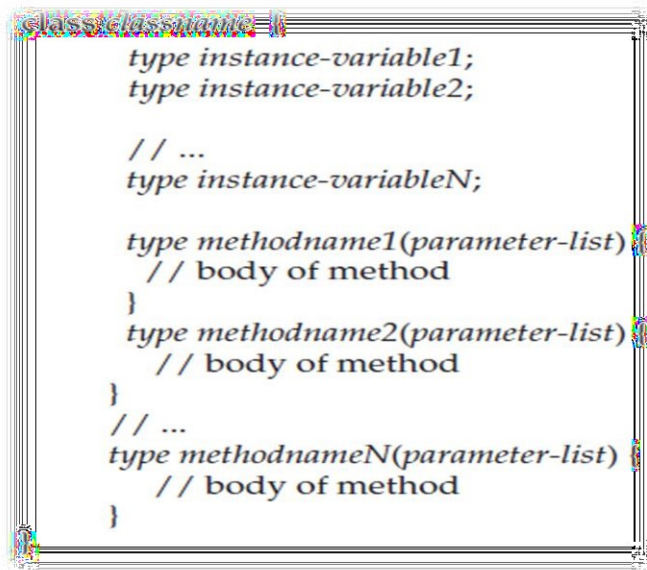


Introducing Classes:

- The class is at the core of Java. It is the logical construct upon which the entire Java language is built because it defines the shape and nature of an object.
- As such, the class forms the basis for object-oriented programming in Java. Any concept you wish to implement in a Java program must be encapsulated within a class.
- Classes have been used since from the beginning. However, until now, only the simplest form of a class has been used.
- The classes created in the preceding sessions primarily exist simply to encapsulate the main() method, which has been used to demonstrate the basics of the Java syntax.
- Perhaps the most important thing to understand about a class is that 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.

The General Form of a Class:

- When we define a class, we declare its exact form and nature.
 - We do this by specifying the data that it contains and the code that operates on that data.
 - While very simple classes may contain only code or only data, most real-world classes contain both.
- A class is declared by use of the class keyword. The classes that have been used up to this point are actually very limited examples of its complete form.
- A simplified general form of a class definition is shown here.....



```
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  
}
```

- The data, or variables, defined within a class are called instance variables. The code is contained within methods.

-
- Collectively, the methods and variables defined within a class are called members of the class.
 - In most classes, the instance variables are acted upon and accessed by the methods defined for that class.
 - Thus, as a general rule, it is the methods that determine how a class data can be used.
 - For each instance of the class (that is, each object of the class) contains its own copy of these variables.
 - Thus, the data for one object is separate and unique from the data for another.
 - Notice that the general form of a class does not specify a main() method. Java classes do not need to have a main() method.
 - We only specify one if that class is the starting point for your program. Further, applets don't require a main() method at all.
 - Let's begin our study of the class with a simple example. Here is a class called Box that defines three instance variables: width, height, and depth. Currently, Box does not contain any methods.

```
class Box {  
    double width;  
    double height;  
    double depth;  
}
```

- As stated, a class defines a new type of data. In this case, the new data type is called Box. We will use this name to declare objects of type Box.
- It is important to remember that a class declaration only creates a template; it does not create an actual object.
- To actually create a Box object, we will use a statement like the following.....

Box mybox = new Box();

- As mentioned earlier, 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, we will use the dot(.) operator. The dot operator links the name of the object with the name of an instance variable.
 - For example, to assign the width variable of mybox the value 100, you would use the following statement: **mybox.width = 100;**
 - This statement tells the compiler to assign the copy of width that is contained within the mybox object the value of 100.
- In general, you use the dot operator to access both the instance variables and the methods within an object.

-
- Here is a complete program that uses the Box class...

```
/* 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);
    }
}
```

Declaring Objects:

- We can use class to declare objects of that type. However, obtaining objects of a class is a two-step process.
 - First, we 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.
 - Second, we 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.
- This reference is, more or less, 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.
- Let's look at the details of this procedure. In the preceding sample programs, a line similar to the following is used to declare an object of type Box:

Box mybox = new Box();

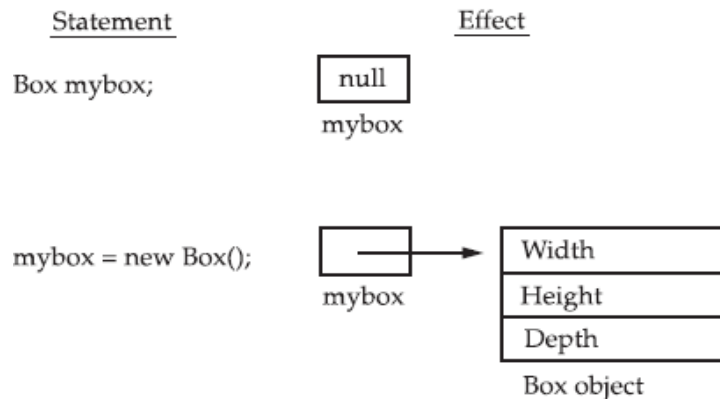
- This statement combines the two steps just described. It can be rewritten like this to show each step more clearly:

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 line executes, mybox contains the value null, which indicates that it does not yet point to an actual object.
- The effect of these two lines of code is depicted in Figure 6-1.

FIGURE 6-1 Declaring an object of type Box



- Let's once again review the distinction between a class and an object.
 - A class creates a new data type that can be used to create objects.
 - That is, a class creates a logical framework that defines the relationship between its members.
 - When we declare an object of a class, we are creating an instance of that class.
- Thus, a class is a logical construct. An object has physical reality. (That is, an object occupies space in memory.).

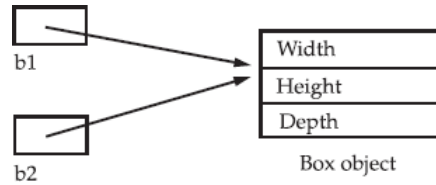
Assigning Object Reference Variables:

- Object reference variables act differently than we might expect when an assignment takes place.
- For example, what do you think the following fragment does?

Box b1 = new Box();

Box b2 = b1;

- We might think that b2 is being assigned a reference to a copy of the object referred to by b1. That is, 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 the same object.
- The assignment of b1 to b2 did not allocate any memory or copy any part of the original object. It simply makes b2 refer to the same object as does b1.
- Thus, any changes made to the object through b2 will affect the object to which b1 is referring, since they are the same object.



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

Introducing Methods:

- As mentioned, classes usually consist of two things: instance variables and methods. However, there are some fundamentals that we need to learn now so that you can begin to add methods to your classes.
- 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 other than those 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 essentially variables that receive the value of the arguments passed to the method when it is called.
 - 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.

Adding a Method to the Box Class:

- Although it is perfectly fine to create a class that contains only data, it rarely happens.

-
- Most of the time, we will use methods to access the instance variables defined by the class.
 - In fact, methods define the interface to most classes. This allows the class implementor to hide the specific layout of internal data structures behind cleaner method abstractions.
 - In addition to defining methods that provide access to data, we can also define methods that are used internally by the class itself.
 - Let's begin by adding a method to the Box class.
 - It may have occurred to you while looking at the preceding programs that the computation of a box's volume was something that was best handled by the Box class rather than the BoxDemo class.
 - After all, since the volume of a box is dependent upon the size of the box, it makes sense to have the Box class compute it.
 - To do this, you must add a method to Box, as shown here.

```
// 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();
    }
}
```

Returning a Value from a method:

- While the implementation of volume() does move the computation of a box's volume inside the Box class where it belongs, it is not the best way to do it.
- For e.g. What if another part of your program wanted to know the volume of a box?
 - A better way to implement volume() is to have it compute the volume of the box and return the result to the caller.
- 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.

```
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();
        double vol;

        // assign values to mybox1's instance variables
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;

        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);
    }
}
```

- One more point:
 - The preceding program can be written a bit more efficiently because there is actually no need for the vol variable.

-
- The call to `volume()` could have been used in the `println()` statement directly, as shown here:
`System.out.println("Volume is " + mybox1.volume());`
 - In this case, when `println()` is executed, `mybox1.volume()` will be called automatically and its value will be passed to `println()`.

Adding a Method That Takes Parameters:

- While some methods don't need parameters, most do. Parameters allow a method to be generalized.
- That is, a parameterized method can operate on a variety of data and/or be used in a number of slightly different situations.
 - To illustrate this point, let's use a very simple example. Here is a method that returns the square of the number 10.
 - However, if you modify the method so that it takes a parameter, as shown next, then you can make `square()` much more useful.

```
int square()  
{  
    return 10 * 10;  
}
```

```
int square(int i)  
{  
    return i * i;  
}
```

- We can use a parameterized method to improve the **Box** class.
- In the preceding examples, the dimensions of each box had to be set separately by use of a sequence of statements, such as...
`mybox1.width = 10; mybox1.height = 20; mybox1.depth = 15;`
- While this code works, it is troubling for two reasons.
 - First, it is clumsy and error prone. For e.g., it would be easy to forget to set a dimension.
 - Second, in well-designed Java programs, instance variables should be accessed only through methods defined by their class.
- In the future, we can change the behavior of a method, but you can't change the behavior of an exposed instance variable.
- Thus, a better approach to setting the dimensions of a box is to create a method that takes the dimensions of a box in its parameters and sets each instance variable appropriately.

```
// 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);
    }
}
```

Constructors:

- It can be tedious to initialize all of the variables in a class each time an instance is created.
- Even when you add convenience functions like **setDim()**, it would be simpler and more concise to have all of the setup done at the time the object is first created.
- 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.
- 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.
- Once defined, the constructor is automatically called immediately after the object is created, before the **new** operator completes.
- Constructors look a little strange because they 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.
- We can rework the **Box** example so that the dimensions of a box are automatically initialized when an object is constructed. To do so, replace **setDim()** with a constructor.

```

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);
    }
}

```

- Before moving on, let's reexamine the **new** operator. As you know, when you allocate an object, you use the following general form...

classname Object_name = new classname();

- Now you can understand why the parentheses are needed after the class name. What is actually happening is that the constructor for the class is being called.
- Thus, in the line `Box mybox1 = new Box();` **new Box()** is calling the **Box()** constructor.
- When you do not explicitly define a constructor for a class, then Java creates a default constructor for the class.
- This is why the preceding line of code worked in earlier versions of **Box** that did not define a constructor.
- The default constructor automatically initializes all instance variables to zero.
- The default constructor is often sufficient for simple classes, but it usually won't do for more sophisticated ones.
- Once you define your own constructor, the default constructor is no longer used.

Parameterized Constructors:

- While the **Box()** constructor in the preceding example does initialize a **Box** object, it is not very useful—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 as shown in the following example...

```

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);
    }
}

```

The this Keyword:

- Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines this keyword, which can be used inside any method to refer to the current object.
- We can use this anywhere a reference to an object of the current class type is permitted.
- To better understand what this refers to, consider the following version of Box().

```

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

```

Instance Variable Hiding:

- As we know, it is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes.
- Interestingly, we can have local variables, including formal parameters to methods, which overlap with the names of the class instance variables.
- However, when a local variable has the same name as an instance variable, the local variable hides the instance variable.
 - This is why **width**, **height**, and **depth** were not used as the names of the parameters to the **Box()** constructor inside the **Box** class.

-
- If they had been, then **width** would have referred to the formal parameter, hiding the instance variable **width**.
 - 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.
 - For e.g., here is another version of **Box()**, which uses **width**, **height**, and **depth** for parameter names and then uses **this** to access the instance variables by the same name:

```
// Use this to resolve name-space collisions.
Box(double width, double height, double depth) {
    this.width = width;
    this.height = height;
    this.depth = depth;
}
```

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 occasionally (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.
- Furthermore, different Java run-time implementations will take varying approaches to garbage collection.

The finalize() Method:

- Sometimes an object will need to perform some action when it is destroyed.
 - For e.g., 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.

-
- To add a finalizer to a class, you simply define the **finalize()** method.
 - The Java run time calls finalize() method whenever it is about to recycle an object of that class.
 - The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects.
 - Right before an asset is freed, the Java run time calls the **finalize()** method on the object.
 - The **finalize()** method has this general form.....

```
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 we 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.

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.
- We will see, method overloading is one of Java's most exciting and useful features.
- 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.
- As you can see in following example... **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.

```
// 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
```

- 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.
- For example, consider the following program...
 - As you can see, this version of **OverloadDemo** does not define **test(int)**.
 - Therefore, when **test()** is called with an integer argument inside **Overload**, no matching method is found.
 - However, Java can automatically convert an integer into a **double**, and this conversion can be used to resolve the call.
 - Therefore, after **test(int)** is not found, Java elevates **i** to **double** and then calls **test(double)**.
 - Of course, if **test(int)** had been defined, it would have been called instead.
 - Java will employ its automatic type conversions only if no exact match is found.

```
// 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)
    }
}
```

- Method overloading supports polymorphism because it is one way that Java implements the “one interface, multiple methods” paradigm.
- To understand how, consider the following. 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. 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.

Overloading Constructors:

- In addition to overloading normal methods, we can also overload constructor methods.
- In fact, for most real-world classes that you create, overloaded constructors will be the normal.
To understand why, let's return to the **Box** class developed in the preceding.

```
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;
    }
}

/* 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);
    }
}
```

Using Objects as Parameters:

- So far, we have only been using simple types as parameters to methods. However, it is both correct and common to pass objects to methods.
- For example, consider the following short program.

```
// Objects may be passed to methods.
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
```

- As you can see, the **equals()** method inside **Test** compares two objects for equality and returns the result.
- That is, it compares the invoking object with the one that it is passed.

-
- If they contain the same values, then the method returns **true**. Otherwise, it returns **false**.
 - Notice that the parameter **o** in **equals()** specifies **Test** as its type.
 - Although **Test** is a class type created by the program, it is used in just the same way as Java's built-in types.
 - One of the most common uses of object parameters involves constructors.
 - Frequently, we will want to construct a new object so that it is initially the same as some existing object.
 - To do this, you must define a constructor that takes an object of its class as a parameter.

```
// Here, Box allows one object to initialize another.
```

```
class Box {
    double width;
    double height;
    double depth;

    // Notice this constructor. It takes an object of type Box.
    Box(Box ob) { // pass object to constructor
        width = ob.width;
        height = ob.height;
        depth = ob.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;
    }
}
```

```
class OverloadCons2 {
    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);

        Box myclone = new Box(mybox1); // create copy of mybox1

        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 cube is " + vol);

        // get volume of clone
        vol = myclone.volume();
        System.out.println("Volume of clone is " + vol);
    }
}
```

A Closer Look at Argument Passing:

- In general, there are two ways that a computer language can pass an argument to a subroutine.
- The first way is call-by-value.
 - This approach copies the value of an argument into the formal parameter of the subroutine.
 - Therefore, changes made to the parameter of the subroutine have no effect on the argument.
- The second way an argument can be passed is call-by-reference.
 - In this approach, a reference to an argument (not the value of the argument) is passed to the parameter.
 - Inside the subroutine, this reference is used to access the actual argument specified in the call.
 - This means that changes made to the parameter will affect the argument used to call the subroutine.
- As you will see, Java uses both approaches, depending upon what is passed.

-
- In Java, when you pass a primitive type to a method, it is passed by value.
 - Thus, what occurs to the parameter that receives the argument has no effect outside the method.
 - When you pass an object to a method, the situation changes dramatically, because objects are passed by what is effectively call-by-reference.
 - Keep in mind that when you create a variable of a class type, you are only creating a reference to an object.

```
// 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
```

- Thus, when you pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument.
- This effectively means that objects are passed to methods by use of call-by-reference. Changes to the object inside the method do affect the object used as an argument.

```
// 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
```

Returning Objects from methods:

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

<pre>// 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;</pre>	<pre> 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); } }</pre>
	<p>The output generated by this program is shown here:</p> <pre>ob1.a: 2 ob2.a: 12 ob2.a after second increase: 22</pre>

- The preceding program makes another important point:
 - Since all objects are dynamically allocated using **new**, you don't need to worry about an object going out-of-scope because the method in which it was created terminates.
 - The object will continue to exist as long as there is a reference to it somewhere in your program.
 - When there are no references to it, the object will be reclaimed the next time garbage collection takes place.

Introducing Access Control:

- As you know, encapsulation links data with the code that manipulates it. However, encapsulation provides another important attribute: access control.
- Through encapsulation, you can control what parts of a program can access the members of a class. By controlling access, you can prevent misuse.
- For e.g., allowing access to data only through a well-defined set of methods, you can prevent the misuse of that data.

-
- Thus, when correctly implemented, a class creates a “black box” which may be used, but the inner workings of which are not open to tampering.
 - However, the classes that were presented earlier do not completely meet this goal.
 - In this section, you will be introduced to the mechanism by which you can precisely control access to the various members of a class.
 - How a member can be accessed is determined by the access specifier that modifies its declaration. Java supplies a rich set of access specifiers.
 - Some aspects of access control are related mostly to inheritance or packages. These parts of Java’s access control mechanism will be discussed later.
 - Here, let’s begin by examining access control as it applies to a single class.
 - Java’s access specifiers are **public**, **private**, and **protected**. Java also defines a default access level.
 - **protected** applies only when inheritance is involved. The other access specifiers are described next.
 - Let’s begin by defining **public** and **private**.
 - When a member of a class is modified by the **public** specifier, then that member can be accessed by any other code.
 - When a member of a class is specified as **private**, then that member can only be accessed by other members of its class.
 - Now you can understand why **main()** has always been preceded by the **public** specifier. It is called by code that is outside the program—that is, by the Java run-time system.
 - When no access specifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package.
 - In the classes developed so far, all members of a class have used the default access mode, which is essentially public.
 - Usually, you will want to restrict access to the data members of a class—allowing access only through methods.
 - Also, there will be times when you will want to define methods that are private to a class.
 - An access specifier precedes the rest of a member’s type specification. That is, it must begin a member’s declaration statement.

```
public int i;  
private double j;  
  
private int myMethod(int a, char b) { // ...
```

-
- To understand the effects of public and private access, consider the following program.....

```
/* This program demonstrates the difference between
   public and private.
*/
class Test {
    int a; // default access
    public int b; // public access
    private int c; // private access

    // methods to access c
    void setc(int i) { // set c's value
        c = i;
    }
    int getc() { // get c's value
        return c;
    }
}

class AccessTest {
    public static void main(String args[]) {
        Test ob = new Test();

        // These are OK, a and b may be accessed directly
        ob.a = 10;
        ob.b = 20;

        // This is not OK and will cause an error
        // ob.c = 100; // Error!

        // You must access c through its methods
        ob.setc(100); // OK
        System.out.println("a, b, and c: " + ob.a + " " +
                           ob.b + " " + ob.getc());
    }
}
```

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.
- Normally, 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.
- We 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.)
- If you need to do computation in order to initialize your **static** variables, you can declare a **static** block that 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.
- 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, which prints a message and then initializes **b** 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**.
- Outside of the class in which they are defined, **static** methods and variables can be used independently of any object.
- To do so, we 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 methods and global variables.

```
// 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);
    }
}
```

Here is the output of the program:

```
Static block initialized.
x = 42
a = 3
b = 12
```

- Here is an example. Inside **main()**, the **static** method **callme()** and the **static** variable **b** are accessed through their class name **StaticDemo**.

```
class StaticDemo {
    static int a = 42;
    static int b = 99;

    static void callme() {
        System.out.println("a = " + a);
    }
}
```

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

Here is the output of this program:

```
a = 42
b = 99
```

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 does not exist independently 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.
 - 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.
 - 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 seldom used.
- 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. The following program illustrates how to define and use an inner class.
- In the following program,
 - An inner class named **Inner** is defined within the scope of class **Outer**. Therefore, any code in class **Inner** can directly access the variable **outer_x**.
 - An instance method named **display()** is defined inside **Inner**. This method displays **outer_x** on the standard output stream

```
// 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

- The **main()** method of **InnerClassDemo** creates an instance of class **Outer** and invokes its **test()** method.
- That method creates an instance of class **Inner** and the **display()** method is called.
- It is important to realize that an instance of **Inner** can be created only within the scope of class **Outer**.
- The Java compiler generates an error message if any code outside of class **Outer** attempts to instantiate class **Inner**.
- We can, however, create an instance of **Inner** outside of **Outer** by qualifying its name with **Outer**, as in **Outer.Inner**.
- As explained, an inner class has access to all of the members of its enclosing class, but the reverse is not true.
- Members of the inner class are known only within the scope of the inner class and may not be used by the outer class. For example.
 - Here, **y** is declared as an instance variable of **Inner**.
 - Thus, it is not known outside of that class and it cannot be used by **showy()**.

```
// This program will not compile.
class Outer {
    int outer_x = 100;

    void test() {
        Inner inner = new Inner();
        inner.display();
    }

    // this is an inner class
    class Inner {
        int y = 10; // y is local to Inner
        void display() {
            System.out.println("display: outer_x = " + outer_x);
        }
    }

    void showy() {
        System.out.println(y); // error, y not known here!
    }
}

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

- Although we have been focusing on inner classes declared as members within an outerclass scope, **it is possible to define inner classes within any block scope.**
- For example, you can define a nested class within the block defined by a method or even within the body of a **for** loop, as this next program shows...

```
// Define an inner class within a for loop.
class Outer {
    int outer_x = 100;

    void test() {
        for(int i=0; i<10; i++) {
            class Inner {
                void display() {
                    System.out.println("display: outer_x = " + outer_x);
                }
            }
            Inner inner = new Inner();
            inner.display();
        }
    }
}
```

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

The output from this version of the program is shown here.

display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
```

- **While nested classes are not applicable to all situations, they are particularly helpful when handling events.**
 - There you will see how inner classes can be used to simplify the code needed to handle certain types of events.
 - You will also learn about anonymous inner classes, which are inner classes that don't have a name.

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.

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 base type of the array.
- The base type determines the data type of each element that comprises the array.
- Thus, the base type for the array determines what type of data the array will hold.

-
- For example, the following declares an array named **month_days** with the type “array of int”...

int month_days[];

- Although this declaration establishes the fact that **month_days** is an array variable, no array actually exists.
- In fact, 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];

- Here,
 - type specifies the type of data being allocated,
 - size specifies the number of elements in the array, and
 - array_var is the array variable that is linked to the array.
- That is, to use **new** to allocate an array, you must specify the type and number of elements to allocate.
- **The elements in the array allocated by new will automatically be initialized to zero.**
- 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.
- Let's review: Obtaining an array is a two-step process.
 - First, you must declare a variable of the desired array type.
 - Second, you must allocate the memory that will hold the array, using **new**, and assign it to the array variable.

- Thus, in **Java all arrays are dynamically allocated.**
- Once you have allocated an array, you can access a specific element in the array by specifying its index within square brackets.
- 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;

- The next line displays the value stored at index 3.

System.out.println(month_days[3]);

- It is possible to combine the declaration of the array variable with the allocation of the array itself, as shown here.....

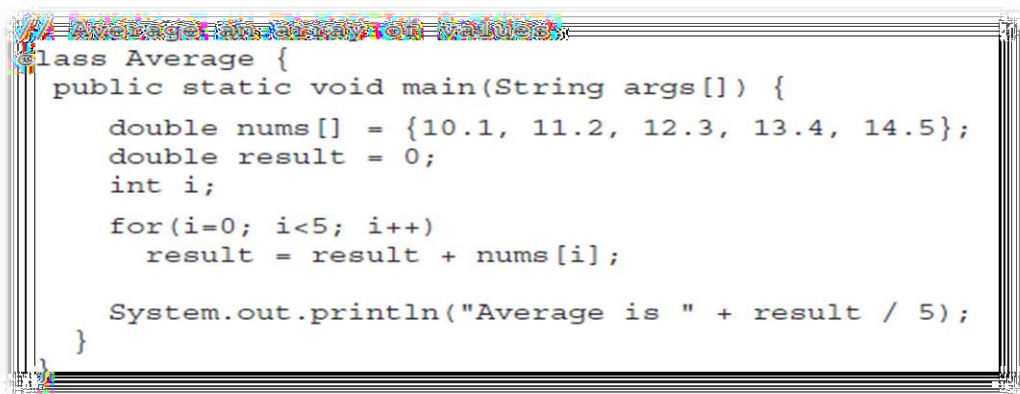
```
int month_days[] = new int[12];
```

- Arrays can be initialized when they are declared. The process is much the same as that used to initialize the simple types.
 - 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**.
- For example, to store the number of days in each month, the following code creates an initialized array of integers...

```
// 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.");
    }
}
```

- Java strictly checks to make sure you do not accidentally try to store or reference values outside of the range of the array. The Java run-time system will check to be sure that all array indexes are in the correct range.
- For e.g., the run-time system will check the value of each index into **month_days** to make sure that it is between 0 and 11 inclusive.
- If you try to access elements outside the range of the array (negative numbers or numbers greater than the length of the array), you will cause a run-time error. It is going to throw an ArrayIndexOutOfBoundsException.
- Here is one more example that uses a one-dimensional array. It finds the average of a set of numbers.



```
// Average an array of values
class Average {
    public static void main(String args[]) {
        double nums[] = {10.1, 11.2, 12.3, 13.4, 14.5};
        double result = 0;
        int i;
        for(i=0; i<5; i++)
            result = result + nums[i];

        System.out.println("Average is " + result / 5);
    }
}
```


- An important point can be made about arrays:
 - They are implemented as objects.
 - Because of this, there is a special array attribute that you will want to take advantage of.
 - Specifically, the size of an array—that is, the number of elements that an array can hold—is found in its **length** instance variable.
 - All arrays have this variable, and it will always hold the size of the array.

```
int ar[] = new int [10];
System.out.println("ar length is "+ ar.length);
```
 - Here is a program that demonstrates this property...

```
// This program demonstrates that all arrays have a length member.
class Length {
    public static void main(String args[]) {
        int a1[] = new int[10];
        int a2[] = {3, 5, 7, 1, 8, 99, 44, -10};
        int a3[] = {4, 3, 2, 1};

        System.out.println("length of a1 is " + a1.length);
        System.out.println("length of a2 is " + a2.length);
        System.out.println("length of a3 is " + a3.length);
    }
}
```

This program displays the following output:

```
length of a1 is 10
length of a2 is 8
length of a3 is 4
```

- As you can see, the size of each array is displayed. Keep in mind that the value of **length** has nothing to do with the number of elements that are actually in use.
- It only reflects the number of elements that the array is designed to hold. You can put the **length** member to good use in many situations.

```
// Improved Stack class that uses the length array member.
class Stack {
    private int stck[];
    private int tos;

    // allocate and initialize stack
    Stack(int size) {
        stck = new int[size];
        tos = -1;
    }

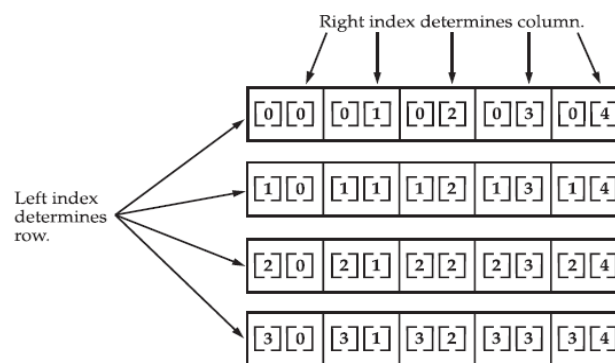
    // Push an item onto the stack
    void push(int item) {
        if(tos==stck.length-1) // use length member
            System.out.println("Stack is full.");
        else
            stck[++tos] = item;
    }
}
```

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 refined differences.
- To declare a multidimensional array variable, specify each additional index using another set of square brackets.
- For example, the following declares a two dimensional array variable called **twoD**.

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

- This allocates a 4 by 5 array and assigns it to **twoD**. Internally this matrix is implemented as an array of arrays of **int**. Conceptually, this array will look like the one shown in Figure 3-1.



Given: `int twoD [] [] = new int [4] [5];`

FIGURE 3-1 A conceptual view of a 4 by 5, two-dimensional array

- The following program numbers each element in the array from left to right, top to bottom, and then displays these values.

```
// Demonstrate a two-dimensional array.
class TwoDArray {
    public static void main(String args[]) {
        int twoD[ ][ ] = new int [4] [5];
        int i, j, k = 0;

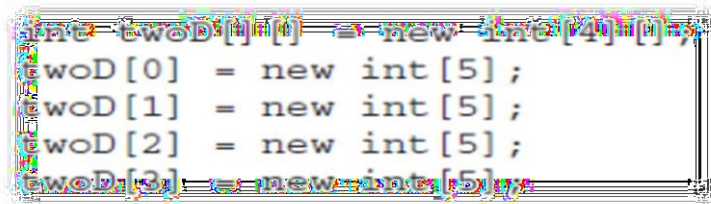
        for(i=0; i<4; i++)
            for(j=0; j<5; j++) {
                twoD[i][j] = k;
                k++;
            }

        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
```

- 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.
- For example, this following code allocates memory for the first dimension of **twoD** when it is declared. It allocates the second dimension manually.



```
int twoD[][] = new int[4][];
twoD[0] = new int[5];
twoD[1] = new int[5];
twoD[2] = new int[5];
twoD[3] = new int[5];
```

- While there is no advantage to individually allocating the second dimension arrays in this situation, there may be in others.
- For example, when you allocate dimensions manually, you do not need to allocate the same number of elements for each dimension.
- As stated earlier, since multidimensional arrays are actually arrays of arrays, the length of each array is under your control.
- For example, the following program creates a two-dimensional array in which the sizes of the second dimension are unequal.
- It is possible to initialize multidimensional arrays. To do so, simply enclose each dimension's initializer within its own set of curly braces.
- The following program creates a matrix where each element contains the product of the row and column indexes.

```
// 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();
    }
}

```

[0][0]			
[1][0]	[1][1]		
[2][0]	[2][1]	[2][2]	
[3][0]	[3][1]	[3][2]	[3][3]

This program generates the following output:

```

0
1 2
3 4 5
6 7 8 9

```

- Also notice that you can use expressions as well as literal values inside of array initializers.

```

// 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();
        }
    }
}

```

When you run this program,
you will get the following output:

```

0.0  0.0  0.0  0.0
0.0  1.0  2.0  3.0
0.0  2.0  4.0  6.0
0.0  3.0  6.0  9.0

```

Alternative Array Declaration Syntax:

- There is a second form that may be used to declare an array:

type[] var_name;

- Here, the square brackets follow the type specifier, and not the name of the array variable.
- For example, the following two declarations are equivalent:

int al[] = new int[3]; int[] a2 = new int[3];

char twod1[][] = new char[3][4]; is same as... char[][] twod2 = new char[3][4];

- This alternative declaration form offers convenience when declaring several arrays at the same time. For example,

int[] nums, nums2, nums3; // create three arrays creates three array variables of type **int**.

- The alternative declaration form is also useful when specifying an array as a return type for a method.

Array of Objects:

- As we have already said, **arrays are capable of storing objects also**. For example, we can create an array of Strings which is a reference type variable.
- However, using a String as a reference type to illustrate the concept of array of objects isn't too appropriate due to the immutability of String objects.
- Therefore, for this purpose, we will use a class Student containing a single instance variable marks. Following is the definition of this class.

```
class Student {  
    int marks;  
}
```

- An array of objects is created just like an array of primitive type data items in the following way.

Student[] studentArray = new Student[7];

- The above statement creates the array which can hold references to seven Student objects. It doesn't create the Student objects themselves.
- They have to be created separately using the constructor of the Student class.
- The studentArray contains seven memory spaces in which the address of seven Student objects may be stored.
- If we try to access the Student objects even before creating them, run time errors would occur. For instance, the following statement throws a NullPointerException during runtime.

studentArray[0].marks = 100;

- The Student objects have to be instantiated using the constructor of the Student class and their references should be assigned to the array elements in the following way...

studentArray[0] = new Student();

- If each of the Student objects have to be created using a different constructor, we use a statement similar to the above several times.
- However, in this particular case, we may use a for loop since all Student objects are created with the same default constructor.

```
for ( int i=0; i<studentArray.length; i++) {  
    studentArray[i]=new Student();  
}
```

- The above for loop creates seven Student objects and assigns their reference to the array elements. Now, a statement like the following would be valid.

studentArray[0].marks=100;

- Enhanced for loops find a better application here as we not only get the Student object but also we are capable of modifying it. This is because of the fact that Student is a reference type.
- Therefore the variable in the header of the enhanced for loop would be storing a reference to the Student object and not a copy of the Student object.

```
for ( Student x : studentArray ) {  
    x.marks = sc.nextInt();  
} // sc is a Scanner object
```

Problem:

1. Define a class student { regNo, Name, Marks of 3 subjects, Total and Avg } with following methods...
 - setValues(). // Use Scanner class inside a method.
 - Calculate total and avg.
 - Display the result (Distinction, Firstclass... etc.,)
 - Create an Array of Students in main class.

Using Command-Line Arguments:

- Sometimes we will want to pass information into a program when you run it. This is accomplished by passing command-line arguments to **main()**.
- 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.

-
- For example, the following program displays all of the command-line arguments that it is called with.

```
// 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
```

```
// Find the sum and average of all command-line arguments (integers)
class CommandLine {
    public static void main(String args[]) {

        int num, sum=0;
        float avg;

        for(int i=0; i<args.length; i++)
            sum = sum + Integer.parseInt(args[i]);

        avg = sum / args.length;

        System.out.println("Sum is " + sum);
        System.out.println("Average is " + avg);
    }
}
```

Self Study: Wrapper classes in Java

Some Problems using Command-Line Arguments:

1. To find the maximum and minimum element among all the command-line arguments passed.
2. To display all the elements in sorted order which passed as command-line arguments.

..... 😊

SAMPLE QUESTIONS:

1. Define a class. Explain the general syntax of defining a class with data fields and methods.
2. The new operator is used to instantiate the class object. Justify the effect of it on a reference variable through a example.
3. Explain the concept of formal arguments and actual arguments with an example.
4. What is instance variable hiding? Illustrate the same with an example and how to this pointer can be helpful to avoid instance variable hiding.
5. Define a class Product-pid, pname, price and methods to accept, display the product details along with a method which return amount for 'N' products. Use the same in main class.
6. Define a class Person - name, age, weight with methods to accept, display and return the BMI = Kg / m² in order to display the appropriate message in main class. criteria - < 18.5 underweight, 18.5 - 24.9 Normal, 25 - 29.9 Overweight and >= 30 Obese.
7. Define a Constructor. Explain its different types with an example.
8. Define a class Employee - Eid, Ename, Basic_Sal to illustrate the overloaded constructors and use the same in main class.
9. Explain how Constructor is different when compare to normal methods with an example.
10. Define a class Time - Hrs, Mins, Secs include appropriate methods to use the following statements in main method - Time t1 = new Time(2,45,55), t1.display(), t3.addTime(t1,t2)
11. Define a class Weight - kgs, gms with parameterized constructor, display method and methods to illustrate how an object can be passed and returned from a method - addWeight() method.
12. Explain the concept and advantages of method overloading with an example.
13. Define a class with overloaded methods to calculate area of Circle, rectangle and triangle and use the same in main class.
14. Non- static members can't be called in a static context. Illustrate with an example
15. Static members exists even before we instantiate the object. Justify it using an example.
16. Define a class NumberOp with static methods to calculate maximum digit, check prime and show how to use the same in main class.
17. Define an Array. Explain its advantages over normal variables.
18. Discuss the compile-time and runtime instantiation of arrays with an example.
19. Write a Java program to find the maximum element in a given array of 'N' elements.
20. Define a class ArrayOp-arr[] with appropriate constructors and methods to accept, display and implement Linear search.
21. Define a class Matrix - mat[][],row,col with appropriate constructors and methods to accept, display and add two matrices. Use the same in main class
22. Illustrate how for-each loop will be helpful while traversing arrays with an example.
