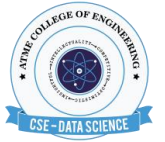




**A T M E**  
College of Engineering



# Introducing Classes

## Module 2- Chapt 1

**Mrs. Madhu Nagaraj**  
**Assistant Professor**  
**Dept of CSE-Data Science**  
**ATMECE**

# Class Fundamentals

The classes created in the preceding chapters primarily exist simply to encapsulate the `main( )` method, which has been used to demonstrate the basics of the Java syntax.

You will now see, classes are more powerful than the limited ones presented so far.

class 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

```
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  
    }  
}
```

**Instance variables:** The data, or variables, defined within a class

**Methods:** The code is contained within methods.

**Members:** Collectively, the methods and variables defined within a class are called members of the class.

Variables defined within a class are called **instance variables** because each instance 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 object**.

Java classes do not need to have a main( ) method.

You only specify one if that class is the starting point for your program.

Further, some kinds of Java applications, such as applets, sevlets don't require a main( ) method at all.

# A Simple Class

Here is a class called Box that defines three instance variables:  
width, height, and depth

Currently, Box does not contain any methods (but some will be added soon).

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

In this case, the new data type is called Box.

You 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.

Thus, the preceding code does not cause any objects of type Box to come into existence.

# Create a Box object

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

After this statement executes, mybox will be an instance of Box.

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, you 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;`

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

You should call the file that contains this program BoxDemo.java, because the main() method is in the class called BoxDemo, not the class called Box.

When you compile this program, you will find that two .class files have been created, one for Box and one for BoxDemo.

The Java compiler automatically puts each class into its own .class file.

It is not necessary for both the Box and the BoxDemo class to actually be in the same source file.

You could put each class in its own file, called Box.java and BoxDemo.java, respectively.

Volume is 3000.0



# Example program that declares two Box objects.



Volume is 3000.0  
Volume is 162.0

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

# Declaring Objects

when you create a class, you are creating a new data type.

We can use this type to declare objects of that type.

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

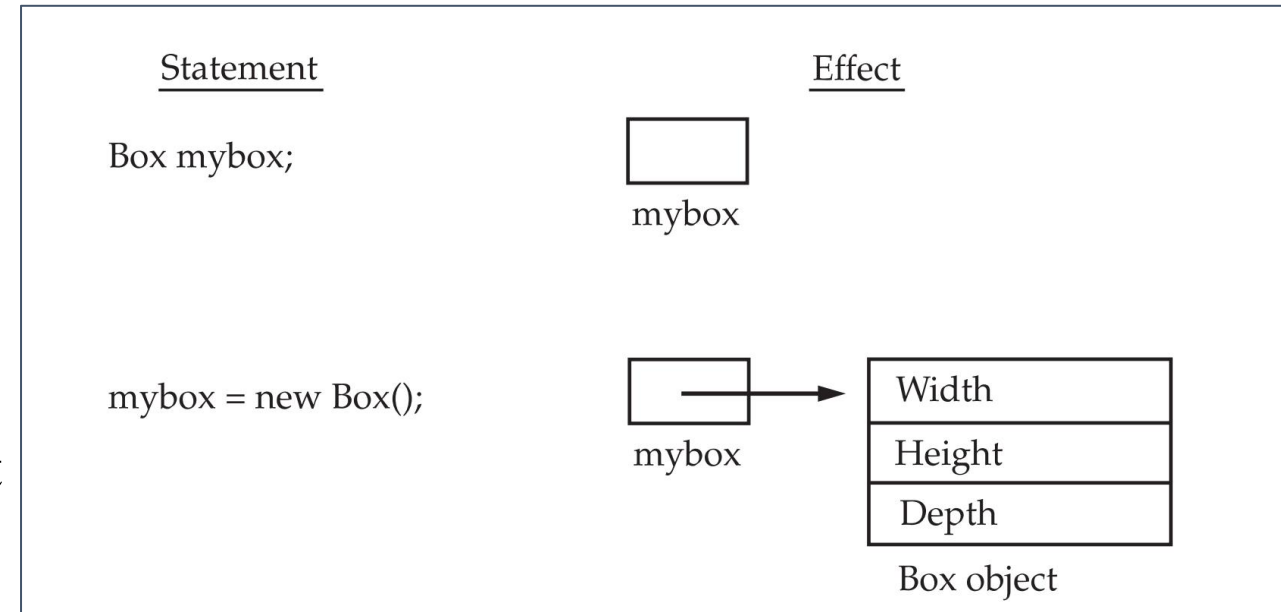
- First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can refer to an object.
- Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the **new operator**.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

`Box mybox = new Box();` this can be written in two steps as

`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. At this point, mybox does not yet refer to an actual object.

The Second line allocates an object and assigns a reference to it to mybox. After the second line executes, you can use mybox as if it were a Box object.



# A Closer Look at new keyword

**new operator** dynamically allocates memory for an object.

It has this general form:

**class-var = new classname ( );**

class-var is a variable of the class type being created. The classname is the name of the class that is being instantiated. The class name followed by parentheses specifies the constructor for the class. A constructor defines what occurs when an object of a class is created.

Constructors are an important part of all classes and have many significant attributes. Most real-world classes explicitly define their own constructors within their class definition.

However, if no explicit constructor is specified, then Java will automatically supply a default constructor. This is the case with Box. For now, we will use the default constructor. Soon, you will see how to define your own constructors.

- new operator is not required for allocating the memory for the variables of type integer or characters.
- Because Java's primitive types are not implemented as objects. Rather, they are implemented as "normal" variables.
- objects have many features and attributes that require Java to treat them differently than it treats the primitive types.
- 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.
- But, 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. Later you will learn how to handle the run-time exception.

# Assigning Object Reference Variables

`Box b1 = new Box();`

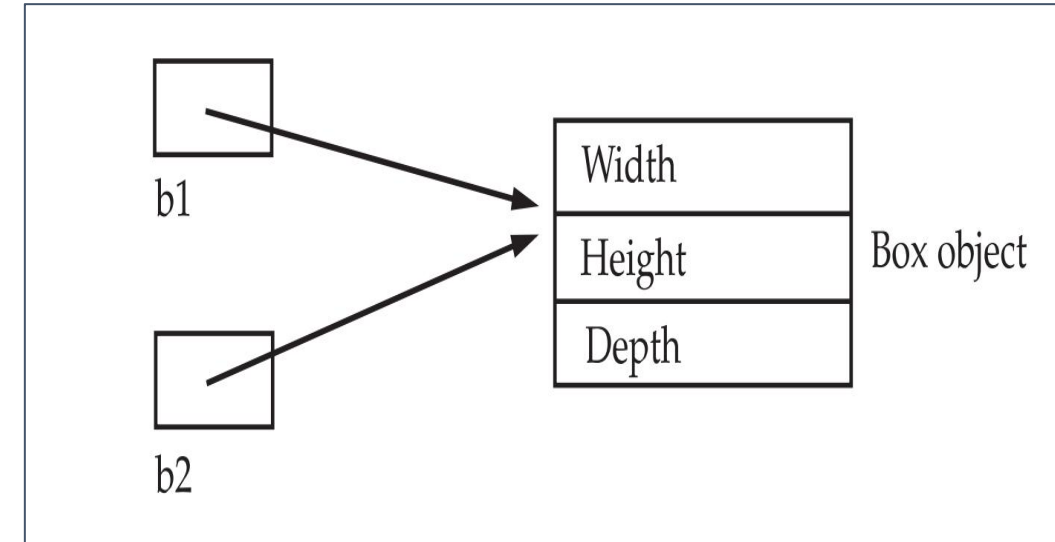
`Box b2 = b1;`

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.

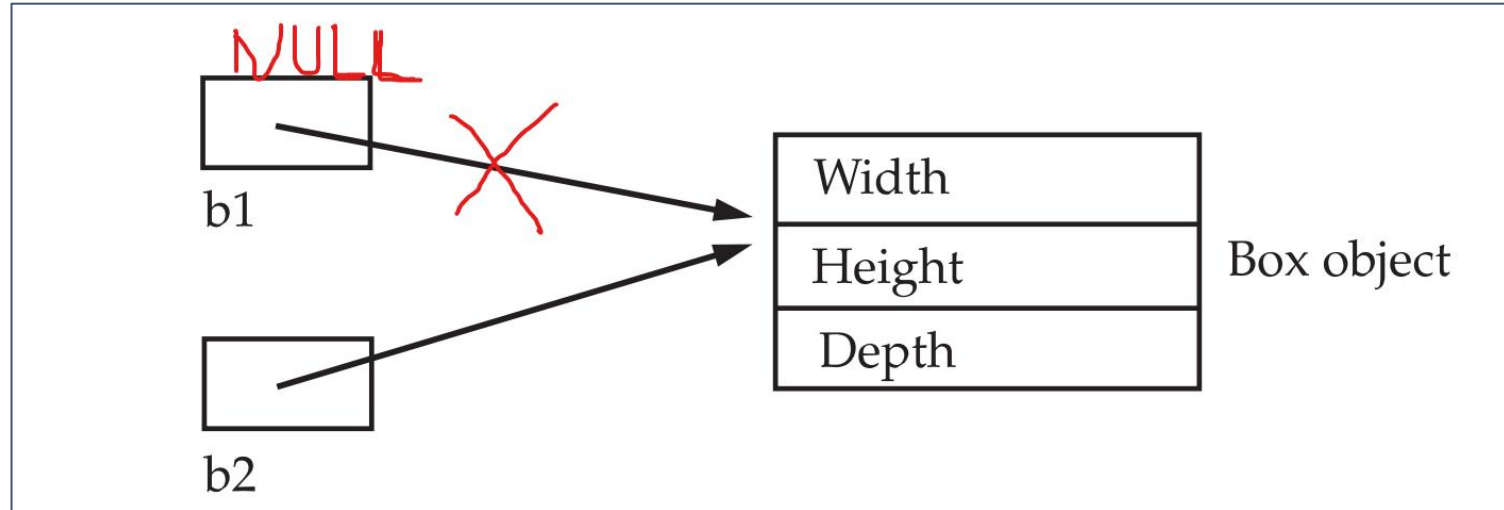
It simply makes b2 refer to the same object as does b1.

Thus, any changes made to the object through b2 will affect the object to which b1 is referring, since they are the same object.



# Assigning Object Reference Variables

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



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

# Introducing Methods

classes usually consist of two things:  
**instance variables and**  
**methods**

This is the general form of a method:

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



# 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();  
    }  
}
```

```
Volume is 3000.0  
Volume is 162.0
```

There is something very important to notice inside the volume( ) method:

i,e the instance variables **width, height, and depth** are referred to directly, without preceding them with an **object name or the dot operator**.

**Volume() method is invoked on the instance.**

**This means that width, height, and depth inside volume( ) implicitly refer to the copies of those**

**variables found in the object that invokes volume( )**

# Let's review:

- 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. (BoxDemo2 Example)
- However, when an instance variable is accessed by code that is part of the same class as the instance variable, that variable can be referred to directly. The same thing applies to methods. (BoxDemo3 Example)



# ATME

College of Engineering

## 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);  
    }  
}
```

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.

**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:

```
System.out.println("Volume is" + mybox1.volume());
```

# Adding a Method That Takes Parameters

- 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.

Example:

Here is a method that returns the square of the number 10:

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

This method's use is very limited.

If you modify the method so that it takes a parameter, as below, then you can make square( ) much more useful.

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

```
int x, y;  
x = square(5); // x equals 25  
x = square(9); // x equals 81
```

```
y = 2;  
x = square(y); // x equals 4
```

# Example that uses a parameterized method

```
// 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 is tedious and difficult to initialize all of the variables in a class each time an instance is created.

Even when you add convenience functions like **setDim( )** as shown in the previous program.

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.

The constructor is automatically called when the object is created,

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**.

```
/* 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;
    }
}
```

Example that  
uses the  
constructor to  
initialize the  
object

```
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);  
    }  
}
```



When this program is run, it generates the following results:

Constructing Box

Constructing Box

Volume is 1000.0

Volume is 1000.0

When you do not explicitly define a constructor for a class, then Java creates a default constructor for the class.

This is why in the preceding example the code worked even though ( that is earlier versions of Box program) that did not define a constructor.

The default constructor automatically initializes all instance variables to their default values, which are **zero, null, and false**, for **numeric types, reference types, and boolean**, respectively.

The default constructor is often sufficient for simple classes, but complicated classes needs the your own constructor to be defined.

Once you define your own constructor, the default constructor is no longer used.

# Parameterized Constructors

The 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. Constructors with the parameters are called as **Parameterized Constructors**

# Example on parameterized constructor

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



The output from this program is shown here:

```
Volume is 3000.0  
Volume is 162.0
```

# 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** keyword is always a reference to the object on which the method was invoked.

To better understand what this refers to, consider the following version of Box( ) constructor:

// A redundant use of this.

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

```
    this.width = w;
```

```
    this.height = h;
```

```
    this.depth = d;
```

```
}
```

This version of `Box( )` operates exactly like the earlier version.

The use of `this` here is redundant, but the constructor is perfectly correct.

Inside `Box( )`, `this` will always refer to the invoking object. While it is redundant in this case, this is useful in other contexts where the local variables takes the same name as the instance variables.

**// Use this to resolve name-space collisions.**

**`Box(double width, double height, double depth) {`**

**`this.width = width;`**

**`this.height = height;`**

**`this.depth = depth;}`**

`Box()` uses `width`, `height`, and `depth` as local parameter names and then uses **this keyword** to access the instance variables by the same name:

# Garbage Collection

Memory to 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.

# The finalize( ) Method

- Sometimes an object will need to perform some action when it is destroyed.
- For example,
  - if an object is holding some non-Java resource such as a file handler, database connections, 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 that method whenever it is about to recycle an object of that class.
- Inside the `finalize( )` method, you will specify those actions that must be performed before an object is destroyed.

The **`finalize( )`** method has this general form:

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



- finalize( ) is only called just prior to garbage collection.
- It is not called when an object goes out-of-scope.
- This means that you cannot know when—or even if—finalize( ) will be executed.
- Therefore, your program should provide other means of releasing system resources, etc., used by the object.
- It must not rely on finalize( ) for normal program operation.



**A T M E**  
College of Engineering



# Methods and Classes

## Module 2- Chapt 2

**Mrs. Madhu Nagaraj**  
**Assistant Professor**  
**Dept of CSE-Data Science**  
**ATMECE**



## Topics discussed

- overloading,
- parameter passing
- recursion
- access control
- the use of the keyword static, and
- use of Java's most important built-in classes: String.



# 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.
- In this case, the methods are said to be **overloaded**, and the process is referred to as **method overloading**.
- Method overloading is one of the ways that Java supports **polymorphism**.
- When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call.

# Example on method overloading

```
// 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
```

# Java's automatic type conversions in method overloading

Java's automatic type conversions can play a role in overload resolution.

```
/*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
```



- OverloadDemo class 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.

- 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,  
**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 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.

# Overloading Constructors

```
/* 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;
    }
}
```

In addition to overloading normal methods, you can also **overload constructor methods.**



```
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
```

## Using Objects as Parameters

- So far, we have only been using simple types as parameters to methods.
- We can also pass the objects as parameters to the methods.
- For example, consider the following short program:

This program generates the following output:

```
ob1 == ob2: true
ob1 == ob3: false
```

```
// 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 equalTo(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.equalTo(ob2));
        System.out.println("ob1 == ob3: " + ob1.equalTo(ob3));
    }
}
```

- One of the most common uses of passing object parameters is in constructors.
- If you 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.
- For example, the following version of 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;
}

// compute and return volume
double volume() {
    return width * height * depth;
}
}
```

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

# Types of argument passing (A Closer Look at Argument Passing)

1. Call by value
2. Call by reference

## **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.

By default Primitive types are passed by value.

## 2. Call by reference:

- In this approach, a reference to an argument (not the value of the argument) is passed to the formal 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.
- Object are passed by reference.

# Primitive types are passed by value.

// Primitive types are passed by value.

```
class Test {  
    void meth(int i, int j) {  
        i *= 2;  
        j /= 2;  
    }  
}  
  
class CallByValue {  
    public static void main(String args[]) {  
        Test ob = new Test();  
        int a = 15, b = 20;  
        System.out.println("a and b before call: " + a + " " + b);  
        ob.meth(a, b);  
        System.out.println("a and b after call: " + a + " " + b);  
    }  
}
```

The output from this program is shown here:

```
a and b before call: 15 20  
a and b after call: 15 20
```

# Objects are passed through their references.

```
// Objects are passed through their references.

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 PassObjRef {
    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

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

```
// Returning an object.
class Test {
    int a;

    Test(int i) {
        a = i;
    }

    Test incrByTen() {
        Test temp = new Test(a+10);
        return temp;
    }
}

class RetOb {
    public static void main(String args[]) {
        Test ob1 = new Test(2);
        Test ob2;

        ob2 = ob1.incrByTen();
        System.out.println("ob1.a: " + ob1.a);
        System.out.println("ob2.a: " + ob2.a);
    }
}
```

```
        ob2 = ob2.incrByTen();
        System.out.println("ob2.a after second increase: "
                           + ob2.a);
    }
}
```

The output generated by this program is shown here:

```
ob1.a: 2
ob2.a: 12
ob2.a after second increase: 22
```

# Recursion

- A method that calls itself is said to be recursive.
- The classic example of recursion is the computation of the factorial of a number.
- The factorial of a number N is the product of all the whole numbers between 1 and N.
- For example,  
3 factorial is  $1 \times 2 \times 3 = 6$

```
// A simple example of recursion.
class Factorial {
    // this is a recursive method
    int fact(int n) {
        int result;

        if(n==1) return 1;
        result = fact(n-1) * n;
        return result;
    }
}

class Recursion {
    public static void main(String args[]) {
        Factorial f = new Factorial();

        System.out.println("Factorial of 3 is " + f.fact(3));
        System.out.println("Factorial of 4 is " + f.fact(4));
        System.out.println("Factorial of 5 is " + f.fact(5));
    }
}
```

# A simple example of recursion

The output from this program is shown here:

```
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
```

```
class RecTest {  
    int values[];  
    RecTest(int i) {  
        values = new int[i];  
    }  
    // display array -- recursively  
    void printArray(int i) {  
        if(i==0) return;  
        else print Array(i-1);  
        System.out.println "[" + (i-1) + "]" + values[i-1];  
    }  
}  
class Recursion2 {  
    public static void main(String args[]) {  
        RecTest ob = new RecTest(10);  
        int i;  
        for(i=0; i<10; i++) ob.values[i] = i;  
        ob.printArray(10);  
    }  
}
```

## Another example that uses recursion.

This program generates the following output:

```
[0] 0  
[1] 1  
[2] 2  
[3] 3  
[4] 4  
[5] 5  
[6] 6  
[7] 7  
[8] 8  
[9] 9
```

# Introducing Access Control

- Encapsulation links data with the code that manipulates it.
- Encapsulation also 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 example, 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.



- How a member can be accessed is determined by the access modifier attached to its declaration.
- Java supplies a rich set of access modifiers.
- Java's access modifiers are **public, private, and protected**. Java also defines a **default access level**.
- **protected** applies only when inheritance is involved.
- **Public access** modifier says member can be accessed by any other code.
- **Private access** modifier says member can be accessed only by other members of its class.
- When no access modifier is used, then by default the member of a class is **public within its own package, but cannot be accessed outside of its package**.
- **Package:** A package is, essentially, a grouping of classes.

# Understanding the effects of public and private access.

```
/* 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;
    }
}
```

- An access modifier precedes the rest of a member's type specification. That is, it must begin a member's declaration statement.
- Here is an example:

```
public int i;  
private double j;
```

```
private int myMethod(int a, char b) { //...
```



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

```
// This class defines an integer stack that can hold 10 values.
class Stack {
    /* Now, both stck and tos are private. This means
       that they cannot be accidentally or maliciously
       altered in a way that would be harmful to the stack.
    */
    private int stck[] = new int[10];
    private int tos;

    // Initialize top-of-stack
    Stack() {
        tos = -1;
    }

    // Push an item onto the stack
    void push(int item) {
        if(tos==9)
            System.out.println("Stack is full.");
        else
            stck[++tos] = item;
    }
}
```

} → Marked  
private

} default  
package  
private

## Access Control in Stack Program

```
// Pop an item from the stack
int pop() {
    if(tos < 0) {
        System.out.println("Stack underflow.");
        return 0;
    }
    else
        return stck[tos--];
}
```

De fault  
package  
private

As you can see, now both stck, which holds the stack, and tos, which is the index of the top of the stack, are **specified as private**.

This means that they cannot be accessed or altered except through **push( ) and pop( )**.

```
class TestStack {  
    public static void main(String args[]) {  
        Stack mystack1 = new Stack();  
        Stack mystack2 = new Stack();  
  
        // push some numbers onto the stack  
        for(int i=0; i<10; i++) mystack1.push(i);  
        for(int i=10; i<20; i++) mystack2.push(i);  
  
        // pop those numbers off the stack  
        System.out.println("Stack in mystack1:");  
        for(int i=0; i<10; i++)  
            System.out.println(mystack1.pop());  
  
        System.out.println("Stack in mystack2:");  
  
        for(int i=0; i<10; i++)  
            System.out.println(mystack2.pop());  
  
        // these statements are not legal  
        // mystack1.tos = -2;  
        // mystack2.stck[3] = 100;  
    }  
}
```

} NOT  
allowed

# Understanding static

- Sometimes it is necessary 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.
- But, it is possible to create a member that can be used by itself, without reference to a specific instance.
- To create such a member, precede its declaration with the keyword **static**.
- When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object.
- You can declare both methods and variables to be static.
- The most common example of a static member is `main( )`.
- `main( )` is declared as static because it must be called before any objects exist.

- Instance variables declared as static are, essentially, **global variables**.
- When objects of its class are declared, **no copy of a static variable is made**.
- Instead, all instances of the class share the same static variable.

Methods declared as static have several restrictions:

- **They can only directly call other static methods.**
- **They can only directly access static data.**
- **They cannot refer to this or super in any way.** (The keyword super relates to inheritance and is described later.)

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

*Executes exactly once when the class is loaded.*

- 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.

Here is the output of the program:

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

# Calling the static method from outside of its class

- 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**.



```
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
```

Inside main( ), the static method callme( ) and the static variable b are accessed through their class name StaticDemo.

# Introducing final

- A field can be declared as **final**.
- Doing so prevents its contents from being modified, making it, essentially, a constant.
- This means that you must initialize a final field when it is declared.
- You can do this in one of two ways:

First, you can give it a value when it is declared.

Second, you can assign it a value within a constructor.

- The first approach is the most common.

# Example

```
final int FILE_NEW = 1;
```

```
final int FILE_OPEN = 2;
```

```
final int FILE_SAVE = 3;
```

```
final int FILE_SAVEAS = 4;
```

```
final int FILE_QUIT = 5;
```

- It is a common coding convention to choose all uppercase identifiers for final fields, as this example shows.
- In addition to fields, both method parameters and local variables can be declared final.
- Declaring a parameter final prevents it from being changed within the method. Declaring a local variable final prevents it from being assigned a value more than once.
- The keyword final can also be applied to methods, but its meaning is substantially different than when it is applied to variables. We will discuss later how to use final for methods

# 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.

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

## **Static Nested Classes:**

A static nested class is a class that is defined within another class and marked as static.

It is associated with the outer class, but it does not have access to the instance variables and methods of the outer class directly.

It can have static members (variables and methods) of its own.

An instance of a static nested class can be created without creating an instance of the outer class.

# Example on static nested class

```
public class OuterClass {  
    // Outer class members...  
  
    static class StaticNestedClass {  
        // Static nested class members...  
    }  
}
```

A static nested class is one that has the static modifier applied.

Because it is static, it must access the non-static members of its enclosing class through an object.

That is, it cannot refer to non-static members of its enclosing class directly.

Because of this restriction, **static nested classes are seldom used.**

# Example on a non-static inner class

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

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.

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.

**An instance of Inner can be created only in the context of class Outer. The Java compiler generates an error message otherwise.**

Output from this application is shown here:

```
display: outer_x = 100
```

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

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

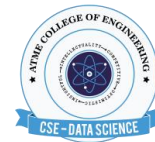
    // this is an inner class
    class Inner {
        int y = 10; // y is local to Inner

        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();
    }
}
```

trying to  
access y.



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,

## Defining nested class within the block [loops]

We have been focusing on inner classes declared as members within an outer class 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/while/do while or any block in general.

# Example on nested Class in for loop

```
// 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:

[illegible]