# Introducing classes

## 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.
- 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 you define a class, you declare its exact form and nature. You 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. As you will see, a class' code defines the interface to its data.
- A class is declared by use of the class keyword
- 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 } }
```

Java

- 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

- 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.
- All methods have the same general form as main( ), which we have been using thus far. However, most methods will not be specified as **static** or **public**.
- Notice that the general form of a class does not specify a main( ) method. Java classes do not need to have a main( ) method.
- You only specify one if that class is the starting point for your program.

## 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)

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

Java

- As stated, a class defines a new type of data. 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.
- To actually create a Box object, you will use a statement like the following:

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

Java

- 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, 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:

```java
mybox.width = 100;
```

- In general, you use the dot operator to access both the instance variables and the methods within an object

```java
/* 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
- To run this program, you must execute BoxDemo.class. When you do, you will see the following output:

```java
Volume is 3000.0
```

- As stated earlier, each object has its own copies of the instance variables. This means that if you have two Box objects, each has its own copy of depth, width, and height. It is important to understand that changes to the instance variables of one object have no effect on the instance variables of another. For example, the following program declares two Box objects:

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

Java

- The output produced by this program is shown here:

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

Java

# Declaring Objects

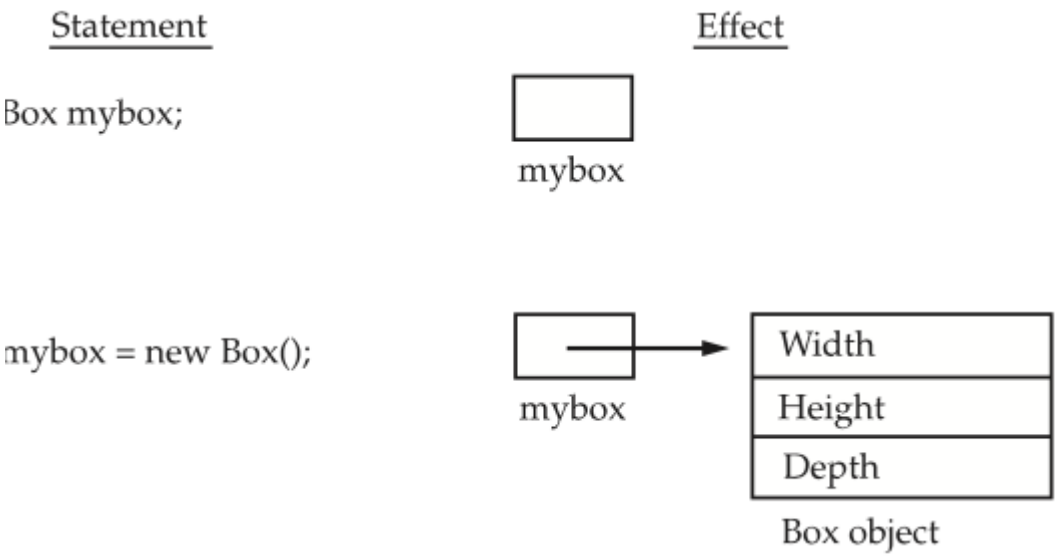
- Obtaining objects of a class is a two-step process. First, you must declare a variable of the class type.
- This variable does not define an object. Instead, it is simply a variable that can refer to an object
- Second, you must acquire an actual, physical copy of the object and assign it to that variable. 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 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

```java
Box mybox = new Box();
```

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

```java
Box mybox; // declare reference to object mybox = new Box(); //
allocate a Box object
```




## A Closer Look at new

- As just explained, the new operator dynamically allocates memory for an object. It has this general form:

```java
class-var = new classname();
```

- 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.
- At this point, you might be wondering why you do not need to use new for such things as integers or characters.
- The answer is that Java's primitive types are not implemented as objects. Rather, they are implemented as "normal" variables. This is done in the interest of efficiency.
- As you will see, objects have many features and attributes that require Java to treat them differently than it treats the primitive types. By not applying the same overhead to the primitive types that applies to objects, Java can implement the primitive types more efficiently.
- Later, you will see object versions of the primitive types that are available for your use in those situations in which complete objects of these types are needed.
- It is important to understand that new allocates memory for an object during run time. The advantage of this approach is that your program can create as many or as few objects as it needs during the execution of your program.
- However, since memory is finite, it is possible that new will not be able to allocate memory for an object because insufficient memory exists.
- If this happens, a run-time exception will occur ( you will learn about this later )

## Assigning Object Reference Variables

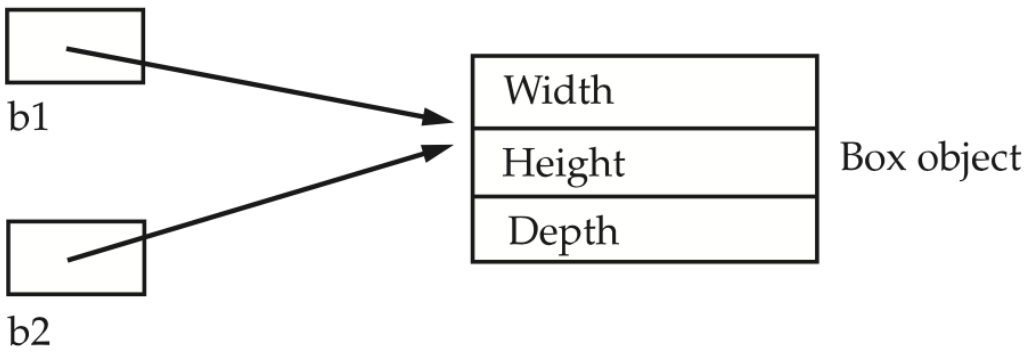
- Object reference variables act differently than you might expect when an assignment takes place. For example, what do you think the following fragment does?

```java
Box b1 = new Box(); Box b2 = b1;
```

Java

- You might think that b2 is being assigned a reference to a copy of the object referred to by b1. That is, you might think that b1 and b2 refer to separate and distinct objects. However, this would be wrong.

- Instead, after this fragment executes, b1 and b2 will both refer to the same object
- Thus, any changes made to the object through b2 will affect the object to which b1 is referring, since they are the same object.




```java
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

- The topic of methods is a large one because Java gives them so much power and flexibility.
- This is the general form of a method:

```java
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:

```java
return value;
```

Java

## Adding a method to the Box class

- Methods define the interface to most classes
- In addition to defining methods that provide access to data, you can also define methods that are used internally by the class itself.

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

Java

- There is something very important to notice inside the volume( ) method: the instance variables width, height, and depth are referred to directly, without preceding them with an object name or the dot operator.
- When a method uses an instance variable that is defined by its class, it does so directly, without explicit reference to an object and without use of the dot operator.

### 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); } }
```

Java

### Adding a method that takes parameters

```
// This program uses a parameterized method. class Box { double
width; double height; double depth; // compute and return volume
double volume() { return width * height * depth; } // sets dimensions
of box void setDim(double w, double h, double d) { width = w; height
= h; depth = d; } } class BoxDemo5 { public static void main(String
args[]) { Box mybox1 = new Box(); Box mybox2 = new Box(); double vol;
// initialize each box mybox1.setDim(10, 20, 15); mybox2.setDim(3, 6,
9); // get volume of first box vol = mybox1.volume();
System.out.println("Volume is " + vol); // get volume of second box
vol = mybox2.volume(); System.out.println("Volume is " + vol); } }
```

Java

## 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.

- 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 when the object is created, before the new operator completes.
- Constructors look a little strange because they have no return type, not even void.

```java
/* Here, Box uses a constructor to initialize the dimensions of a
box. */ class Box { double width; double height; double depth; //
This is the constructor for Box. Box() {
System.out.println("Constructing Box"); width = 10; height = 10;
depth = 10; } // compute and return volume double volume() { return
width * height * depth; } } class BoxDemo6 { public static void
main(String args[]) { // declare, allocate, and initialize Box
objects Box mybox1 = new Box(); Box mybox2 = new Box(); double vol;
// get volume of first box vol = mybox1.volume();
System.out.println("Volume is " + vol); // get volume of second box
vol = mybox2.volume(); System.out.println("Volume is " + vol); } }
```

Java

```
Constructing Box Constructing Box Volume is 1000.0 Volume is 1000.0
```

Java

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

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

Java

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

Java

## The this Keyword

- Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the this keyword.
- this can be used inside any method to refer to the current object.
- That is, this is always a reference to the object on which the method was invoked

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

Java

- This version of Box( ) operates exactly like the earlier version. The use of this is redundant, but perfectly correct. Inside Box( ), this will always refer to the invoking object.
- While it is, redundant in this case, **this** is useful in other contexts, one of which is explained in the next section.

### Instance Variable hiding

- As you know, it is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes. Interestingly, you 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.
- Because this lets you refer directly to the object, you can use it to resolve any namespace collisions that might occur between instance variables and local variables

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

Java

## Garbage Collection

- Since objects are dynamically allocated by using the new operator, you might be wondering how such objects are destroyed and their memory released for later reallocation
- In some languages, such as C++, dynamically allocated objects must be manually released by use of a delete operator.
- Java takes a different approach; it handles deallocation for you automatically. The technique that accomplishes this is called garbage collection
- It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. There is no explicit need to destroy objects as in C++.

- Garbage collection only occurs sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used. Furthermore, different Java run-time implementations will take varying approaches to garbage collection, but for the most part, you should not have to think about it while writing your programs.

## 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 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 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.
- Right before an asset is freed, the Java run time calls the finalize( ) method on the object.

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

Java

- Here, the keyword protected is a specifier that limits access to finalize( ). Will be discussed more in detail later

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