# 5.1: Introducing classes and objects

IT1406 - Introduction to Programming

Level I - Semester 1

# Introducing classes and objects

- What is a Class?
  - A class is a template/blueprint for an object
  - Describes the behavior/state that the object of its type support

- What is an Object?
  - An object is an instance of a class
  - Objects have states and behaviors.
    - Example: A dog has states - color, name, breed as well as behaviors – wagging the tail, barking, eating

# Introducing classes and objects

- A class is declared by use of the class keyword
- A simplified general form of a **class** definition

```
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
}
}
```

# Introducing classes and objects

- The data, or 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.
- The code is contained within *methods.*
- The methods and variables defined within a class are called *members of the class.*
- 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. Further,  some kinds of Java applications, such as applets, don’t require a **main( )** method at all.

# Introducing classes and objects

- A Simple Class

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

- This class called **Box** that defines three instance variables: **width**, **height**, and **depth**. Currently, **Box** does not contain any methods
- Class defines a new type of data. In this case, the new data type is called **Box**
- A class declaration only creates a template; it does not create an actual object

# Introducing classes and objects

- Statement for creating an object:

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

- 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;
```

- The complete program that uses the Box class:

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

# Introducing classes and objects

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

```
Volume is 3000.0
```

# Introducing classes and objects

- 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:

```
// 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;
```

- Assigning values to instance variables and compute them

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

# Introducing classes and objects

- The output produced by this program is shown here:

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

- As you can see, **mybox1**'s data is completely separate from the data contained in **mybox2**.

# Introducing classes and objects

## 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.

# Introducing classes and objects

## Declaring Objects

- 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
```

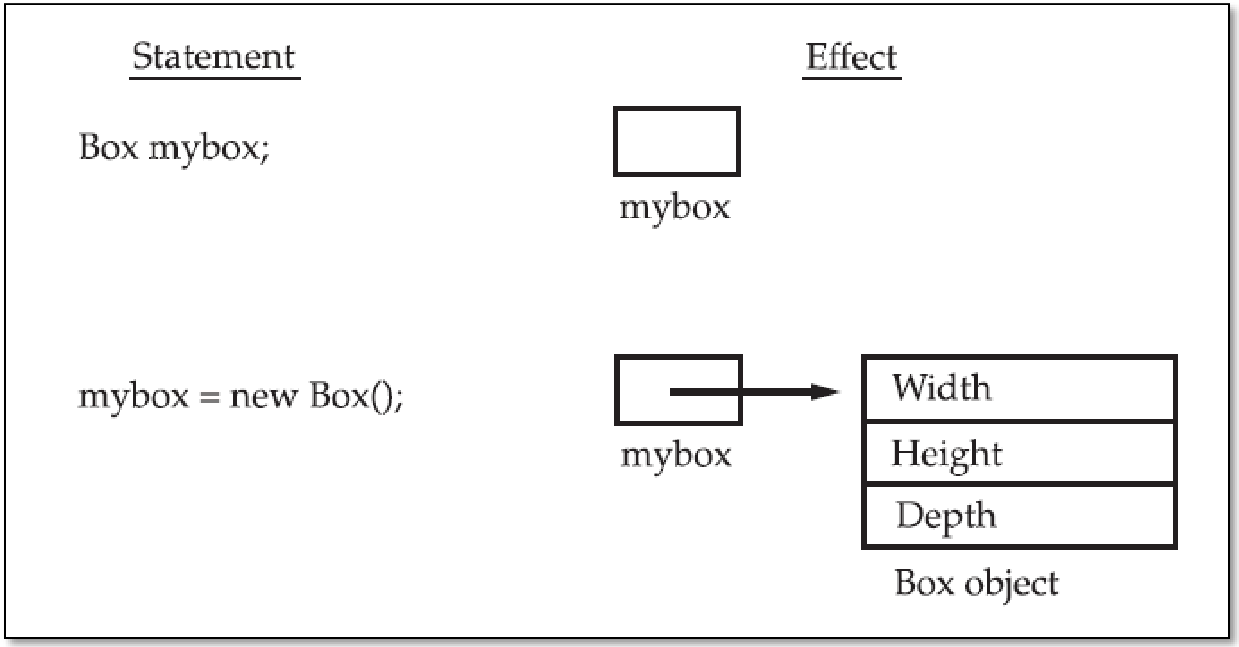
# Introducing classes and objects

## A Closer Look at new

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

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

- Declaring an object of type Box



©2020 e-Learning Centre, UCSC

# Introducing classes and objects

## A Closer Look at new

- **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
- 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.

# Introducing classes and objects

## A Closer Look at new

- 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.
- When you declare an object of a class, you are creating an instance of that class.

# Introducing classes and objects

**Assigning Object Reference Variables**

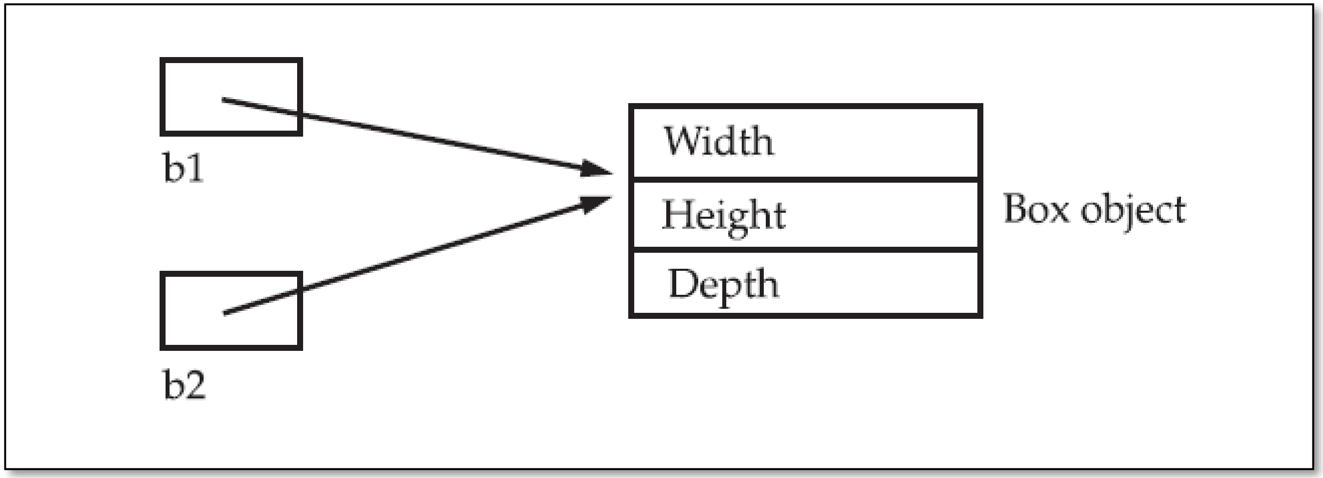
Object reference variables act differently than you might expect when an assignment takes place. For example, what do you think the following fragment does?

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

You might think that 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. 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.

# Introducing classes and objects

## Assigning Object Reference Variables



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

# Introducing classes and objects

## Introducing Methods

- This is the general form of a method:

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

- Here, *type* specifies the type of data returned by the method. This can be any valid type, including class types that you create. If the method does not return a value, its return type must be **void**. The name of the method is specified by *name*. This can be any legal identifier 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.

# Introducing classes and objects

## Introducing Methods

- 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.

# Introducing classes and objects

## Adding a Method to the Box Class

- Methods define the interface to most classes. This allows the class implementer to hide the specific layout of internal data structures behind cleaner method abstractions.

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

# Introducing classes and objects

- This program generates the following output, which is the same as the previous version.

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

- Look closely at the following two lines of code:

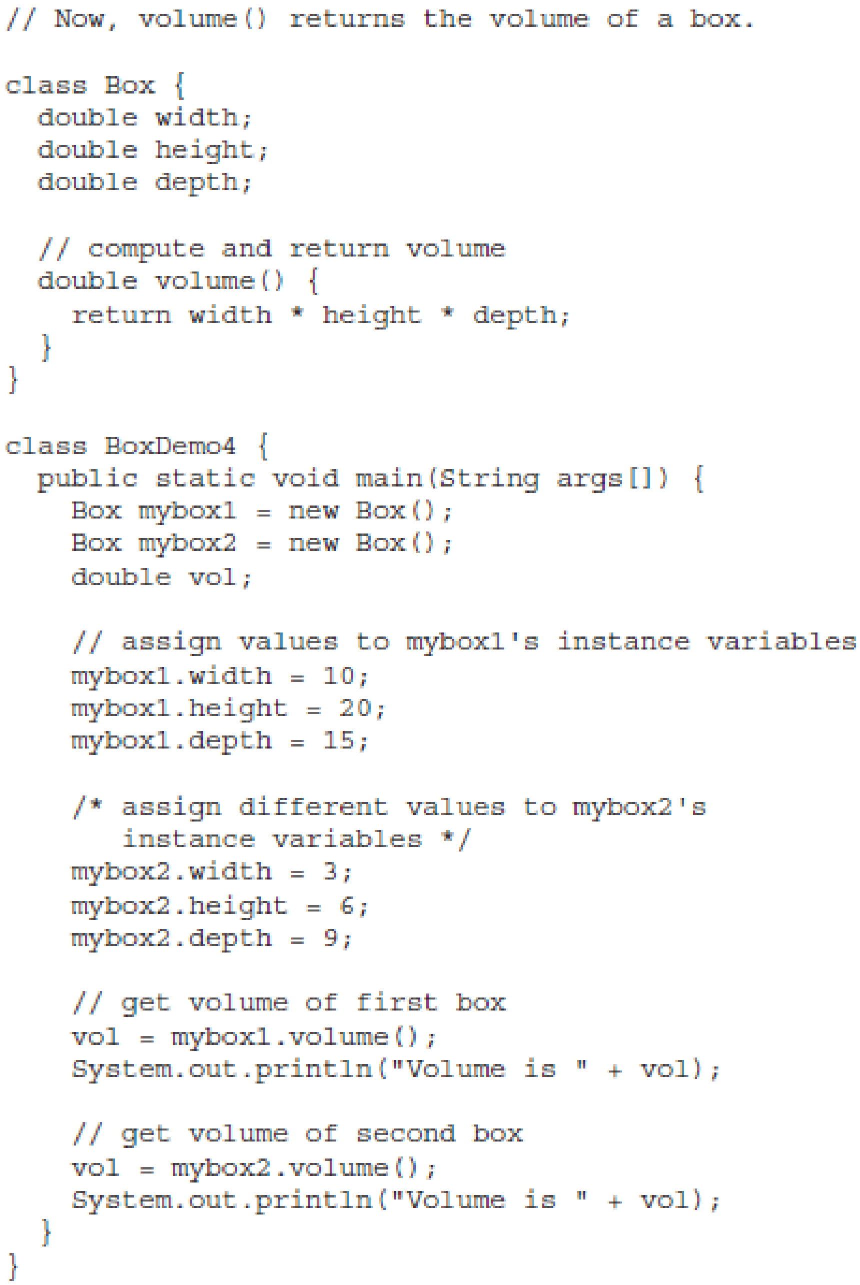
```
mybox1.volume();
mybox2.volume();
```

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

# Introducing classes and objects

## Returning a Value

- 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 example, what if another part of your program wanted to know the volume of a box, but not display its value? A better way to implement **volume( )** is to have it compute the volume of the box and return the result to the caller. The following example, an improved version of the preceding program, does just that:



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

# Introducing classes and objects

- As you can see, when **volume( )** is called, it is put on the right side of an assignment statement. On the left is a variable, in this case **vol**, that will receive the value returned by **volume( )**. Thus, after

```
vol = mybox1.volume();
```

  executes, the value of **mybox1.volume( )** is 3,000 and this value then is stored in **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.

# Introducing classes and objects

## 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:

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

# Introducing classes and objects

- While this method does, indeed, return the value of 10 squared, its use is very limited. 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(int i)
{
return i * i;
}
```

- Now, **square( )** will return the square of whatever value it is called with. That is, **square( )** is now a general-purpose method that can compute the square of any integer value, rather than just 10.
- Here is an example:

```
int x, y;
x = square(5); // x equals 25
x = square(9); // x equals 81
y = 2;
x = square(y); // x equals 4
```

# Introducing classes and objects

- 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 concept is implemented by the following program:

```
// 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;
```

# Introducing classes and objects

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

# Introducing classes and objects

- As you can see, the **setDim( )** method is used to set the dimensions of each box. For example, when

```
mybox1.setDim(10, 20, 15);
```

is executed, 10 is copied into parameter **w**, 20 is copied into **h**, and 15 is copied into **d**. Inside **setDim( )** the values of **w**, **h**, and **d** are then assigned to **width**, **height**, and **depth**, respectively.

# Introducing classes and objects

## 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**. This is because the implicit return type of a class' constructor is the class type itself.

# Introducing classes and objects

## Constructors

- You 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.
- Let’s begin by defining a simple constructor that simply sets the dimensions of each box to the same values. This version is shown here:

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

# Introducing classes and objects

## Constructors

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

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

- As you can see, both **mybox1** and **mybox2** were initialized by the **Box( )** constructor when they were created. Since the constructor gives all boxes the same dimensions, 10 by 10 by 10, both **mybox1** and **mybox2** will have the same volume. The **println( )** statement inside **Box( )** is for the sake of illustration only. Most constructors will not display anything. They will simply initialize an object.

# Introducing classes and objects

## Parameterized Constructors

The easy solution is to add parameters to the constructor. As you can probably guess, this makes it much more useful. For example, the following version of **Box** defines a parameterized constructor that sets the dimensions of a box as specified by those parameters. Pay special attention to how **Box** objects are created.

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

# Introducing classes and objects

## Parameterized Constructors

- The output from this program is shown here:

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

- As you can see, each object is initialized as specified in the parameters to its constructor. For example, in the following line,

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

the values 10, 20, and 15 are passed to the **Box( )** constructor when **new** creates the object.

- Thus, **mybox1**'s copy of **width**, **height**, and **depth** will contain the values 10, 20, and 15, respectively.

# Introducing classes and objects

## The ‘this’ Keyword

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

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

# Introducing classes and objects

## The ‘this’ Keyword

- 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.

# Introducing classes and objects

## 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. If they had been, then **width,** for example, 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 namespace collisions that might occur between instance variables and local variables.

# Introducing classes and objects

## Instance Variable Hiding

- For example, 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;
}
```

# Introducing classes and objects

## 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.

# Introducing classes and objects

## The finalize( ) Method

- 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 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.

# Introducing classes and objects

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

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

- Here, the keyword **protected** is a specifier that limits access to **finalize( )**. 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.