**Introducing Methods**

1. **General Form**

This is the general form of a method:

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

}

* 1. Here, type specifies the type of data returned by the method. This can be any valid type, including class types that you create.
  2. If the method does not return a value, its return type must be **void**.
  3. 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.
  4. The parameter-list is a sequence of type and identifier pairs separated by commas.
  5. Parameters are essentially variables that receive the value of the arguments passed to the method when it is called.
  6. If the method has no parameters, then the parameter list will be empty.

1. **Adding Method to the Box Class**
   1. Adding volume method to Box class instead of BoxDemo class

// display volume of a box

void volume() {

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

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

}

* 1. This program generates the following output, which is the same as the previous output.
  2. Lets closely look at the method calls
  3. When mybox1.volume( ) is executed, the Java run-time system transfers control to the code defined inside volume( ). After the statements inside volume( ) have executed, control is returned to the calling routine, and execution resumes with the line of code following the call. In the most general sense, a method is Java’s way of implementing subroutines.
  4. 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.**
  5. 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. This is easy to understand if you think about it. A method is always invoked relative to some object of its class. Once this invocation has occurred, the object is known. Thus, within a method, there is no need to specify the object a second time. This means that width, height, and depth inside volume( ) implicitly refer to the copies of those variables found in the object that invokes volume( ).
  6. **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. 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.

1. **Returning a value**
   1. 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.
   2. 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:

double volume() {

return width \* height \* depth;

}

* 1. 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.
  2. There are two important things to understand about returning values:
  3. 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.
  4. 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.
  5. **One more point**: The preceding program can be written a bit more efficiently because there is actually no need for the vol variable. The call to volume( ) could have been used in the println( ) statement directly, as shown here:

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

* 1. In this case, when println( ) is executed, mybox1.volume( ) will be called automatically and its value will be passed to println( ).

1. **Adding a Method That Takes Parameters**
   1. To compute 10 square, we can add a method:

int square()

{

return 10 \* 10;

}

* 1. This has limited use, a better one would be:

int square(int i)

{

return i \* i;

}

* 1. 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

* 1. It is important to keep the two terms parameter and argument straight. **A parameter** is a variable defined by a method that receives a value when the method is called. For example, in square( ), i is a parameter. **An argument is a value** that is passed to a method when it is invoked. For example, square(100) passes 100 as an argument. Inside square( ), the parameter i receives that value.
  2. You can use a parameterized method to improve the Box class. In the preceding examples, the dimensions of each box had to be set separately by use of a sequence of statements, such as:

mybox1.width = 10;

mybox1.height = 20;

mybox1.depth = 15;

* 1. We can forget to set height, so a better way of doing this would be:

class Box {

double width;

double height;

double depth;

// compute and return volume

double volume() {

return width \* height \* depth;

}

// sets dimensions of box

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

width = w;

height = h;

depth = d; }

}

class BoxDemo5 {

public static void main(String args[]) {

Box mybox1 = new Box();

Box mybox2 = new Box();

double vol;

// initialize each box

mybox1.setDim(10, 20, 15);

mybox2.setDim(3, 6, 9);

// get volume of first box

vol = mybox1.volume();

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

// get volume of second box

vol = mybox2.volume();

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

} }

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

mybox1.setDim(10, 20, 15);

* 1. 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.

**Constructors**

1. **Need for constructors**
   1. 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.
   2. Because the requirement for initialization is so common, Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor.
2. **What is a constructor?**
   1. 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.
   2. Once defined, the constructor is automatically called immediately after the object is created, before the new operator completes.
   3. 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.
   4. It is the constructor’s job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.
   5. Updating the Box Example

/\* Here, Box uses a constructor to initialize the

dimensions of a box.

\*/

class Box {

double width;

double height;

double depth;

// This is the constructor for Box.

Box() {

System.out.println("Constructing Box");

width = 10;

height = 10;

depth = 10;

}

// compute and return volume

double volume() {

return width \* height \* depth;

}

}

class BoxDemo6 {

public static void main(String args[]) {

// declare, allocate, and initialize Box objects

Box mybox1 = new Box();

Box mybox2 = new Box();

double vol;

// get volume of first box

vol = mybox1.volume();

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

// get volume of second box

vol = mybox2.volume();

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

} }

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

Constructing Box

Constructing Box

Volume is 1000.0

Volume is 1000.0

* 1. Now you can understand why the parentheses are needed after the class name. What is actually

happening is that the constructor for the class is being called. Thus, in the line

Box mybox1 = new Box();

* 1. **new Box( ) is calling the Box( ) constructor**. When you do not explicitly define a constructor for a class, then Java creates a **default constructor for the class**
  2. . This is why the preceding line of code worked in earlier versions of Box that did not define a constructor.
  3. **The default constructor automatically initializes all instance variables to zero**. The default constructor is often sufficient for simple classes, but it usually won’t do for more sophisticated ones. **Once you define your own constructor, the default constructor is no longer used.**

1. **Parameterized Constructors**
   1. 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.
   2. The easy solution is to add parameters to the constructor. As you can probably guess, this makes them 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.
   3. Example:

/\* 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

* 1. 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);

* 1. 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.

**This keyword**

1. **Why we need the this keyword?**
   1. Sometimes a method will need to refer to the object that invoked it.
   2. To allow this, Java defines the this keyword. this can be used inside any method to refer to the current object.
   3. 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.
   4. To better understand what this refers to, consider the following version of Box( ):

// A redundant use of this.

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

this.width = w;

this.height = h;

this.depth = d;

* 1. 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.

1. **Instance Variable Hiding**
   1. 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.
   2. However, when a local variable has the same name as an instance variable, the local variable hides the instance variable.
   3. This is why width, height, and depth were not used as the names of the parameters to the Box( ) constructor inside the Box class.
   4. If they had been, then width would have referred to the formal parameter, hiding the instance variable width.
   5. While it is usually easier to simply use different names, there is another way around this situation. Because this lets you refer directly to the object, you can use it to resolve any name space collisions that might occur between instance variables and local variables.
   6. 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;

}