**Constructor:**

* Constructors are used **to construct new instance/object.**
* Purpose is **to initialize the object**.
* Constructor always have same **name as that of class**.

Example :

Queue q = new Queue (); //default cons

Queue q2=new Queue();

The new keyword says to create a new object. The instance variable q can be used

to refer to this object. Thus, the statement:

q.enque("one");

says to enque the string "one" in the Queue referred to by q.

It is often stated that Java does not have *pointers* and technically this is true.

However, class reference variables act like pointers (or really more like reference

variables in C++). Assigning class instance variables do not create new instances.

Constructor

It can be tedious to initialize all of the variables in a class each time an instance is created. Even when you add convenience functions like **setDim( )**, it would be simpler and more concise to have all of the setup done at the time the object is first created. Because the requirement for initialization is so common, **Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor.**

A *constructor* initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method. Once defined, the constructor is automatically called immediately after the object is created,Constructors look a little strange because they have no return type, not even **void**. This is because the implicit return type of a class' constructor is the class type itself. It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.

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

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

Now you can understand why the parentheses are needed after the class name. What is actually happening is that the constructor for the class is being called. Thus, in the line

Box mybox1 = new Box();

**new Box( )** is calling the **Box( )** constructor. When you do not explicitly define a

constructor for a class, then Java creates a default constructor for the class. 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.

/\* 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();//creatin obj

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

**Parameterized Constructors**

While the **Box( )** constructor in the preceding example does initialize a **Box** object, it is not very useful—all boxes have the same dimensions. What is needed is a way to construct **Box** objects of various dimensions. The easy solution is to add parameters to the constructor. As you can probably guess, this makes them much more useful. For example, the following version of **Box** defines a parameterized constructor which 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.

- 107 -

\*/

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

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.

**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.

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;

}

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.

If they had been, then **width** would have referred to the formal parameter, hiding the instance variable **width**. While it is usually easier to simply use different names, there is another way around this situation. Because **this** lets you refer directly to the object, you can use it to resolve any name space collisions that might occur between instance variables and local variables. For 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;

}