Classes

“**Class** is a template that describes the kinds of state and behavior that objects of its type support”. It is the logical construct upon which the entire Java language is built because it defines the shape and nature of an object. As such, the class forms the basis for object-oriented programming in Java.

The general form of a class:
When you define a class, you declare its exact form and nature. While very simple classes may contain only code or only data, most real-world classes contain both. The 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
}
}//class body closed
```

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 *embers* of the class. In most classes, the instance variables are acted upon and accessed by the methods defined for that class.

NOTE: 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, applets don't require a **main( )** method at all.

Object

A Class creates a new data type that can be used to create objects. That is, a class creates a logical framework that defines the relationship between its members. When you declare an object of a class, you are creating an instance of that class. Thus, a class is a logical construct. An object has physical reality. (i.e., an object occupies space in memory.)

Object: At runtime, when the Java Virtual Machine (JVM) encounters the new keyword, it will use the appropriate class to make an object which is an instance of that class. That object will have its own state, and access to all of the behaviors defined by its class.

State (**instance variables**)**:** Each object (instance of a class) will have its own unique set of instance variables as defined in the class. Collectively, the values assigned to an object's instance variables make up the object's state.
Behavior (**methods**): When a programmer creates a class, she creates methods for that class. Methods are where the class' logic is stored. Methods are where the real work gets done. They are where algorithms get executed, and data gets manipulated.

The new operator:

We can create an object using **new** operator. The **new** operator dynamically allocates memory for an object during run time. It has this general form:

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

Here,

- *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.

Note:

1. The new operator returns a reference to a new instance of the ***classname*** class. This reference can be assigned to a reference variable of the appropriate class.
2. Each object has a unique identity and has its own copy of the fields declared in the class definition. The Advantage of this approach is that your program can create as many or as few object as it needs during the execution of your program.
3.

Types of Objects:

1. Instance Object: It is having it's own memory space and always assign a new memory space.
Example: **Box mybox;**
mybox=new Box ();

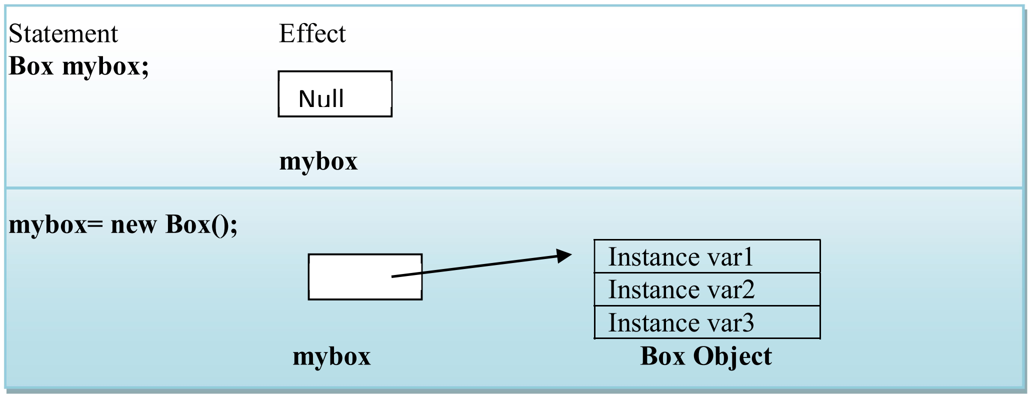
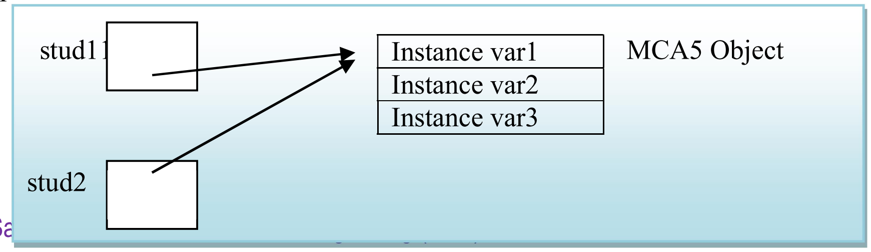


Figure: Declaring an object of type Box

2. Reference Object: For the reference object no memory space will be given. It always pointing to other memory space assigning already existing memory space. Example:
Box b1 = new Box();
Box b2 = b1;

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.

This situation is depicted here:



Although **b1** and **b2** both refer to the same object, they are not linked in any other way.
For example, a subsequent assignment to **b1** will simply *unhook* **b1** from the original object without affecting the object or affecting **b2**. For example:

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

Here, **b1** has been set to **null**, but **b2** still points to the original object.
Note: 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.

Bundling code into individual software objects provides a number of benefits, including:

1. Modularity: The source code for an object can be written and maintained independently of the source code for other objects. Once created, an object can be easily passed around inside the system.
2. Information-hiding: By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.
3. Code re-use: If an object already exists (perhaps written by another software developer), you can use that object in your program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.
4. Pluggability and debugging ease: If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is analogous to fixing mechanical problems in the real world. If a bolt breaks, you replace *it*, not the entire machine.

Constructor

"Constructors are the code that runs whenever you use the keyword new." Every class, *including abstract classes*, MUST have a constructor.

Rules for Constructor:

- Constructors can use any access modifier, including private. (A private constructor means only code within the class itself can instantiate an object of that type)
- A *constructor* initializes an object immediately upon creation.
- The constructor name must match the name of the class.
- Constructors must not have a return type.
- Once defined, the constructor is automatically called immediately after the object is created, before the **new** operator completes.
- Constructors must not have a return type, not even **void**. This is because the implicit return type of a class' constructor is the class type itself.
- It's legal (but stupid) to have a method with the same name as the class, but that doesn't make it a constructor. If you see a return type, it's a method rather than a constructor. In fact, you could have both a method and a constructor with the same name—the name of the class—in the same class, and that's not a problem for Java. Be careful not to mistake a method for a constructor—be sure to look for a return type.
- 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.
- Abstract classes have constructors, and those constructors are always called when a concrete subclass is instantiated.
- Interfaces do not have constructors. Interfaces are not part of an object's inheritance tree.

- The only way a constructor can be invoked is from within another constructor. In other words, you can't write code that actually calls a constructor as follows:

```
class Horse {
        Horse() { } // constructor
        void doStuff() {
        Horse();// calling the constructor - illegal!
        }}
```

Default Constructor

If you want a no-arg constructor and you've typed any other constructor(s) into your class code, the compiler won't provide the no-arg constructor (or any other constructor). A default Constructor does not accept any parameter's values from the object when it gets created.

Note:

- A call to super() can be either a no-arg call or can include arguments passed to the super constructor.
- A no-arg constructor is not necessarily the default (i.e., compiler-supplied) constructor, although the default constructor is always a no-arg constructor.
- The default constructor is the one the compiler provides! While the default constructor is always a no-arg constructor, you're free to put in your own noarg constructor.
- You cannot make a call to an instance method, or access an instance variable, until after the super constructor runs.
- Only static variables and methods can be accessed as part of the call to `super()` or `this()`.

For Example:
Box mybox1 = new Box();
Where: 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. This is why the preceding line of code worked in earlier versions of **Box** that did not define a constructor. The default constructor automatically initializes all instance variables to zero.

Parameterized Constructor:

These Constructors have arguments. The values to these arguments are passed from the object when it is created. For example:

```
class MyClass{
int c;
public MyClass(int num){
c=num;
}
}
```

The this Keyword

While a method need to refer to the object that invoked it. To allow this, Java defines **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.

Note:

1. **this** keyword is used as reference object to current class in which it is used.
2. **this** can be used for passing reference of the current class to a method call.
3. Like **super** keyword, **this** is also not accessible in the main() method.

The use of **this** is redundant, but perfectly correct. There are two uses of this keyword:

1. **this** will always refer to the invoking object. To better understand what **this** refers to, consider the following version of **Box( )**:

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

2. Another use of **this** is to call a constructor of a class in another constructor into the same class. **this()** always means a call to another constructor in the same class. But what happens after the call to this()? A call to this() just means you're delaying the inevitable.

```
Class A{
int a;
A(){
a=10;
}
A(int b){
this();
A=a+b;
}
public static void main(String args[]){
A ob=new A(5);
}}
```

Key Rule: The first line in a constructor must be a call to this().

Methods

Classes usually consist of two things: instance variables and methods. The topic of methods is a large one because Java gives them so much power and flexibility. 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
}

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:

return *value*;

Here, *value* is the value returned.

Instance Variable Hiding:

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;
}
```

That it is a good convention to use the same names for clarity, and use **this** to overcome the instance variable hiding. It is a matter of taste which approach you adopt. Although **this** is of no significant value in the examples just shown, it is very useful in certain situations.

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". 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 implements polymorphism. If you have never used a language that allows the overloading of methods, then the concept may seem strange at first. Method overloading is one of Java's most exciting and useful features.

The rules are simple:

- Overloaded methods MUST change the argument list.
- Overloaded methods CAN change the return type.
- Overloaded methods CAN change the access modifier.
- Overloaded methods CAN declare new or broader checked exceptions.

Here is a simple example that illustrates 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
```

Note:

- When an overloaded method is invoked, Java uses the type and/or number of arguments to determine which version of the overloaded method to actually call.
- Thus, overloaded methods must differ in the type and/or number of their parameters.
- When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.
- A method can be overloaded in the *same* class or in a *subclass*.

When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters. However, this match need not always be exact. In some cases Java's automatic type conversions can play a role in overload resolution. For example, consider the following program:

```
// 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
```

This version of **OverloadDemo** 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.

Method overloading supports polymorphism because it is one way that Java implements the "one interface, multiple methods" paradigm.

Overloading Constructors

Overloading a constructor means typing in multiple versions of the constructor, each having a different argument list, like the following examples:

```
class Foo {
Foo() { }
Foo(String s) { }
}
```

In fact, for most real-world classes that you create, overloaded constructors will be the norm, not the exception. Following is the latest version of **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;
}}
```

The **Box( )** constructor requires three parameters. This means that all declarations of **Box** objects must pass three arguments to the **Box( )** constructor. For example, the following statement is currently invalid:
Box ob = new Box();

Since **Box( )** requires three arguments, it's an error to call it without them. This raises some important questions. What if you simply wanted a box and did not care (or know) what its initial dimensions were? Or, what if you want to be able to initialize a cube by specifying only one value that would be used for all three dimensions? As the **Box** class is currently written, these other options are not available to you.

Fortunately, the solution to these problems is quite easy: simply overload the **Box** constructor so that it handles the situations just described. 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() { }
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}}
class OverloadCons {
public static void main(String args[]) {
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
```

The proper overloaded constructor is called based upon the parameters specified when **new** is executed.

Using Objects as Parameters

So far we have only been using simple types as parameters to methods. However, it is both correct and common to pass objects to methods. For example, consider the following short program:
// 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 equals(Test ob) {
if(ob.a == a && ob.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.equals(ob2));
System.out.println("ob1 == ob3: " +
ob1.equals(ob3));
}
}
```

This program generates the following output:

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

The **equals( )** method inside **Test** compares two objects for equality and returns the result. That is, it compares the invoking object with the one that it is passed. If they contain the same values, then the method returns **true**. Otherwise, it returns **false**. Notice that the parameter **o** in **equals( )** specifies **Test** as its type.

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

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 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.

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

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

Here, the keyword **protected** is a specifier that prevents access to **finalize ( )** by code defined outside its class.

- 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.