Introducing classes:

Classes are the core of Java. A class 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.

General form of a class

When a class is defined, its exact form and nature are declared. The data that the class contains and the code that operates on the data have to be specified.

A class is declared by use of the **class** keyword.

```
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
}
}
```

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. Variables defined within a class are called instance variables.

A simple class

```
class Box {
double width;
double height;
double depth;
void volume() {
        System.out.print("Volume is ");
        System.out.println(width * height * depth);
}
}
class BoxDemo {
public static void main(String args[]) {
        Box mybox1 = new Box();
        mybox1.width = 10;
        mybox1.height = 20;
```

```
        mybox1.depth = 15;

        mybox1.volume();
        }
}
```

Creating instances of a class

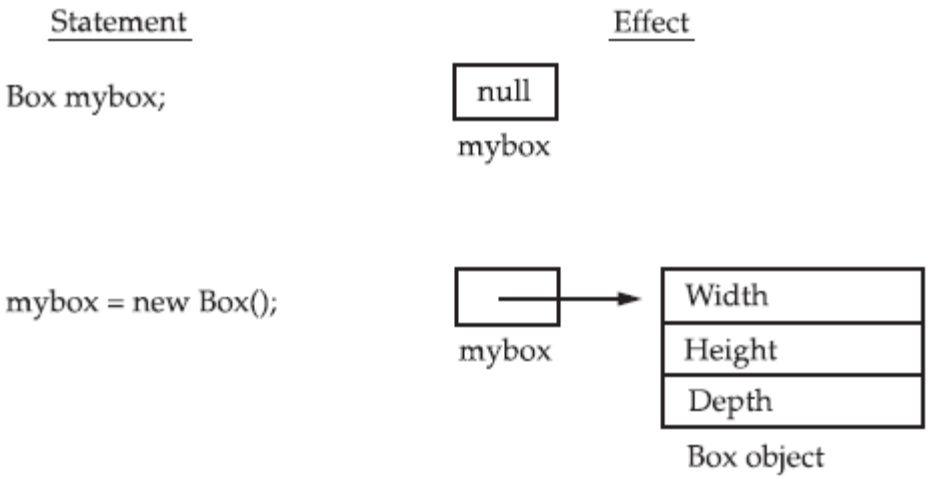
When a class is created, a new data type is created. This new data type can be used to declare objects of that type. Obtaining objects of a class is a 2 step process.

1. Declare a variable of the class type : This variable does not define an object but is a variable that can *refer* to an object.
2. Acquire an actual, physical copy of the object and assign it to that variable : This can be done using the **new** operator. The **new** operator dynamically allocates memory for an object and returns a reference to it.

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

The above statement combines the two steps. It can be rewritten like below to show each step more clearly:

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



Constructors

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, before the **new** operator completes. Constructors do not have a return type, not even **void**.

```
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 BoxDemo{
        public static void main(String args[]) {

        Box mybox1 = new Box();

        double vol;

        vol = mybox1.volume();
        System.out.println("Volume is " + vol);
        }
        }
```

Output

Constructing Box

Volume is 1000.0

The constructor in the above program contains no parameters. However a constructor can also have parameters and those are termed as parameterized constructors.

```
class Box {
        double width;
        double height;
        double depth;
// This is a parameterized constructor for Box.
        Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
        }

        double volume() {
        return width * height * depth;
        }
}

class BoxDemo {
        public static void main(String args[]) {

        Box mybox1 = new Box(10, 20, 15);
        double vol;

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


        }
```

```
}
```

Output

Volume is 3000.0

"this" keyword

"this" can be used inside any method to refer to the current object. It is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes. One can have local variables, including formal parameters to methods, which overlap with the names of the class' instance variables. Because **this** lets user to refer directly to the object, one can use it to resolve any namespace collisions that might occur between instance variables and local variables.

```
Box(double width, double height, double depth) {
        this.width = width;
        this.height = height;
        this.depth = depth;
}
```

Garbage Collection

Objects are dynamically allocated by using the **new** operator. Objects should be destroyed and their memory must be released for later reallocation. In C++, dynamically allocated objects must be manually released by use of a **delete** operator. Java handles deallocation automatically. The technique that accomplishes this is called *garbage collection*. 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. Different Java run-time implementations will take varying approaches to garbage collection.

The finalize() method

An object will need to perform some action before it is destroyed. If an object is holding some non-Java resource such as a file handle, it has to be freed before an object is destroyed. To handle such situations, Java provides a mechanism called *finalization*. By using finalization, one can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.

The finalize( ) method has this general form:

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

Stack Class – An example

```
class Stack {
        int stck[] = new int[10];
        int tos;

        // Initialize top-of-stack
        Stack() {
        tos = -1;
        }

        // Push an item onto the stack
        void push(int item) {
        if(tos==9)
        System.out.println("Stack is full.");
        else
        stck[++tos] = item;
        }

        // Pop an item from the stack
        int pop() {
        if(tos < 0) {
        System.out.println("Stack underflow.");
        return 0;
        }
        else
        return stck[tos--];
                }
        }
```

The Stack class defines two data items and three methods. The stack of integers is held by the array stck. This array is indexed by the variable tos, which always contains the index of the top of the stack. The Stack( ) constructor initializes tos to –1, which indicates an empty stack. The method push( ) puts an item on the stack. To retrieve an item, pop( ) is called.

The class TestStack, demonstrates the Stack class. It creates two integer stacks, pushes some values onto each, and then pops them off.

```
        class TestStack {
        public static void main(String args[]) {
        Stack mystack1 = new Stack();
        Stack mystack2 = new Stack();

        // push some numbers onto the stack
        for(int i=0; i<10; i++) mystack1.push(i);
        for(int i=10; i<20; i++) mystack2.push(i);

        // pop those numbers off the stack
        System.out.println("Stack in mystack1:");
        for(int i=0; i<10; i++)
        System.out.println(mystack1.pop());

        System.out.println("Stack in mystack2:");
        for(int i=0; i<10; i++)
        System.out.println(mystack2.pop());
```

```
}
}
```
This program generates the following output:
```
Stack in mystack1:
9
8
7
6
5
4
3
2
1
0
Stack in mystack2:
19
18
17
16
15
14
13
12
11
10
```
A Closer Look at Methods and Classes:

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. When this is the case, 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 supports polymorphism. When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide 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. While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method. 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.

Example Program
```
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);
      }
}
```

Output

```
No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625
```

Method **test( )** is overloaded four times. The first version takes no parameters, the second takes one integer parameter, the third takes two integer parameters, and the fourth takes one **double** parameter. The fourth version of **test( )** also returns a value is of no consequence relative to overloading, since return types do not play a role in overload resolution.

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.

```
// 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)
      }
}
```

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.

Overloading Constructors

In addition to overloading normal methods, constructor methods can also be overloaded. For most real-world classes that are created, overloaded constructors will be the norm.

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

Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}

double volume() {
return width * height * depth;
}
}
```

Box( ) constructor requires three parameters. All declarations of **Box** objects must pass three arguments to the **Box( )** constructor. The following statement is currently invalid:

```
Box ob = new Box();
```

Since **Box( )** requires three arguments, it's an error to call it without them. As a solution to this the **Box** constructor can be overloaded so that it handles the above situation.

```
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() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
```

```
depth = -1; // box
}
// constructor used when cube is created
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[]) {
// create boxes using the various constructors
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);
}
}
```

Output

```
Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0
Volume of mycube is 343.0
```

Using Objects as Parameters

Objects can be passed as parameters into methods or constructors.

Example Program.

```
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 o) {
if(o.a == a && o.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));
}
}
```

Output

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

The **equals( )** method inside **Test** compares two objects for equality and returns the result. 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**. The parameter **o** in **equals( ) s**pecifies **Test** as its type.

In order to construct a new object that it is initially the same as some existing object, a constructor that takes an object of its class as a parameter has to be defined. The example below shows how an object can be passed as a parameter to a constructor.

```
class Box {
double width;
double height;
double depth;
// Notice this constructor. It takes an object of type Box.
Box(Box ob) { // pass object to constructor
width = ob.width;
height = ob.height;
depth = ob.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() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
double volume() {
return width * height * depth;
}
}
class OverloadCons2 {
public static void main(String args[]) {
// create boxes using the various constructors
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box();
Box mycube = new Box(7);
Box myclone = new Box(mybox1); // create copy of mybox1
```

```
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 cube is " + vol);
// get volume of clone
vol = myclone.volume();
System.out.println("Volume of clone is " + vol);
}
}
```

Argument Passing

There are two ways in which an argument can be passed to a subroutine.

1. Call by value
2. Call by reference

The Call by value approach, copies the *value* of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument. In the second approach (call by reference), a reference to an argument is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. The changes made to the parameter will affect the argument used to call the subroutine.

Program that demonstrates pass by value

```
class Test {
void meth(int i, int j) {
i *= 2;
j /= 2;
      }
}
class CallByValue {
public static void main(String args[]) {
Test ob = new Test();
int a = 15, b = 20;
System.out.println("a and b before call: " + a + " " + b);
ob.meth(a, b);
System.out.println("a and b after call: " + a + " " + b);
      }
}
```

Output

```
a and b before call: 15 20
a and b after call: 15 20
```

The operations that occur inside **meth( )** have no effect on the values of **a** and **b** used in the call; their values here did not change to 30 and 10.

When an object is passed to a method, it is a call by reference variant. When a reference is passed to a method, the parameter that receives it will refer to the same object as that referred to by the argument. Changes to the object inside the method *do* affect the object used as an argument.

Program that demonstrate pass by reference

```
class Test {
int a, b;
Test(int i, int j) {
a = i;
b = j;
      }
void meth(Test o) {
o.a *= 2;
o.b /= 2;
      }
}
class CallByRef {
public static void main(String args[]) {
Test ob = new Test(15, 20);
System.out.println("ob.a and ob.b before call: " + ob.a + " " + ob.b);
ob.meth(ob);
System.out.println("ob.a and ob.b after call: " + ob.a + " " + ob.b);
      }
}
```

Output

```
ob.a and ob.b before call: 15 20
ob.a and ob.b after call: 30 10
```

The actions inside **meth( )** have affected the object used as an argument.

Returning objects

A method can return any type of data, including the class types that are created.

Program to demonstrate how objects are returned.

```
class Test {
int a;
Test(int i) {
a = i;
}
Test incrByTen() {
Test temp = new Test(a+10);
return temp;
      }
}
class RetOb {
public static void main(String args[]) {
Test ob1 = new Test(2);
Test ob2;
ob2 = ob1.incrByTen();
System.out.println("ob1.a: " + ob1.a);
System.out.println("ob2.a: " + ob2.a);
ob2 = ob2.incrByTen();
System.out.println("ob2.a after second increase: "+ ob2.a);
      }
}
```

Output

```
ob1.a: 2
ob2.a: 12
```

```
ob2.a after second increase: 22
```

Each time **incrByTen( )** is invoked, a new object is created, and a reference to it is returned to the calling routine.

Access Control

Encapsulation links data with the code that manipulates it. Encapsulation provides another important attribute: *access control.* Java's access specifiers are **public**, **private**, and **protected**. Java also defines a default access level. **protected** applies only when inheritance is involved. When a member of a class is modified by the **public** specifier, then that member can be accessed by any other code. When a member of a class is specified as **private**, then that member can only be accessed by other members of its class. **main( )** is always preceded by the **public** specifier. It is called by code that is outside the program—that is, by the Java run-time system. When no access specifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package. An access specifier precedes the rest of a member's type specification. Here is an example:

```
public int i;
private double j;
private int myMethod(int a, char b)
```

Program that demonstrates difference between public and private.

```
class Test {
int a; // default access
public int b; // public access
private int c; // private access
// methods to access c
void setc(int i) { // set c's value
c = i;
}
int getc() { // get c's value
return c;
      }
}
class AccessTest {
public static void main(String args[]) {
Test ob = new Test();
// These are OK, a and b may be accessed directly
ob.a = 10;
ob.b = 20;
// This is not OK and will cause an error
// ob.c = 100; // Error!
// You must access c through its methods
ob.setc(100); // OK
System.out.println("a, b, and c: " + ob.a + " " +
ob.b + " " + ob.getc());
      }
}
```

Inside the **Test** class, **a** uses default access, which for this example is the same as specifying **public**. **b** is explicitly specified as **public**. Member **c** is given private access. It cannot be accessed by code outside of its class. inside the **AccessTest** class, **c** cannot be used directly. It must be accessed through its public methods: **setc( )** and **getc( )**.

The Static keyword

When there is a need to define a class member that will be used independently of any object of that class static keyword should be used. It is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member, precede the member's declaration with the keyword **static**. When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object. Both methods and variables can be defined to be **static**.

Instance variables declared as **static** are, essentially, global variables. When objects of its class are declared, no copy of a **static** variable is made. Instead, all instances of the class share the same **static** variable.

Methods declared as **static** have several restrictions:
• They can only call other **static** methods.
• They must only access **static** data.

If any computation needs to be done in order to initialize static variables, a static block can be declared. This block will get executed only once, when the class is first loaded.

Program that demonstrates static variables, methods, and blocks.

```
class UseStatic {
static int a = 3;
static int b;
static void meth(int x) {
System.out.println("x = " + x);
System.out.println("a = " + a);
System.out.println("b = " + b);
}
static {
System.out.println("Static block initialized.");
b = a * 4;
}
public static void main(String args[]) {
meth(42);
}
}
```

As soon as the **UseStatic** class is loaded, all of the **static** statements are run. First, **a** is set to **3**, then the **static** block executes, which prints a message and then initializes **b** to **a * 4** or **12**. Then **main( )** is called, which calls **meth( )**, passing **42** to **x**. The three **println( )** statements refer to the two **static** variables **a** and **b**, as well as to the local variable **x**.

Output

```
Static block initialized.
x = 42
a = 3
b = 12
```

Outside of the class in which they are defined, **static** methods and variables can be used independently of any object. To do so, one needs to only specify the name of the class followed by the dot operator.

Program to demonstrate how static variables and methods can be accessed.

```
class StaticDemo {
static int a = 42;
static int b = 99;
static void callme() {
System.out.println("a = " + a);
}
}
class StaticByName {
public static void main(String args[]) {
StaticDemo.callme();
System.out.println("b = " + StaticDemo.b);
      }
}
```

Output

```
a = 42
b = 99
```

Final keyword

A variable can be declared as **final**. Doing so prevents its contents from being modified.
A **final** variable must be initialized when it is declared.

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
```

It is a common coding convention to choose all uppercase identifiers for **final** variables.

Command Line Arguments

If the user wishes to pass information into a program when he runs it, it can be done through command line arguments to main().A command-line argument is the information that directly follows the program's name on the command line when it is executed. The command line arguments are stored as strings in the **String** array passed to the args parameter of main().The first command-line argument is stored at **args[0]**, the second at **args[1]**, and so on.

Program that displays command line arguments.

```
class CommandLine {
public static void main(String args[]) {
for(int i=0; i<args.length; i++)
System.out.println("args[" + i + "]: " +
args[i]);
      }
}
```