Java Class

Introduction

- In object-oriented programming, a class is a construct that is used as a blueprint (or template) to create objects of that class.
  - **A *class* is the blueprint from which individual objects are created.**

- In the Java programming language, source files (.java files) are compiled into (virtual) machine-readable class files which have a .class extension.
- Since Java is a platform-independent language, source code is compiled into an output file known as bytecode, which it stores in a .class file.
- If a source file has more than one class, each class is compiled into a separate .class file.
- These .class files can be loaded by any Java Virtual Machine (JVM).

Simple Class

- Here is a class called Box that defines three member variables: width, height, and depth.

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

- To create a Box object, you will use a statement like the following:

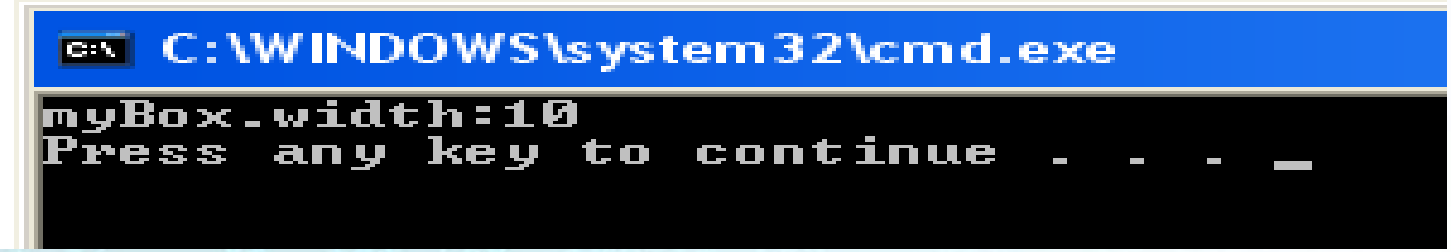
```
Box myBox = new Box();
```

- **myBox** is an instance of Box.
- mybox contains its own copy of each instance variable, width, height, and depth, defined by the class.
- To access these variables, you will use the **dot (.)** operator.

Simple program using class

Here is a complete program that uses the Box class:

```
class Box {
        int width;
        int height;
        int depth;
        }
public class SimpleclassMain {
  public static void main(String args[ ]) {
                Box myBox = new Box();
                myBox.width = 10;
                   System.out.println("myBox.width:"+myBox.width);
                                }
                }
```



Declaring objects

Object creation is a two-step process.

1. declare a variable of the class type.
2. acquire a physical copy of the object and assign it to that variable using the new operator.

The ***new operator*** dynamically allocates memory for an object and returns a reference to it.

The following lines are used to declare an object of type Box:

Box mybox;

declares mybox as a reference to an object of type Box. mybox is null now.

mybox = new Box(); // allocate a Box object

allocates an actual object and assigns a reference to it to mybox.

It can be rewritten to:

Box mybox = new Box();

- Any attempt to use a null mybox will result in a compile-time error.

Closer look @ new

The new operator has this general form:

classVariable = new classname( );

- 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.

Assigning Object Reference Variables:-

Consider the following code:

Box b1 = new Box();

Box b2 = b1;

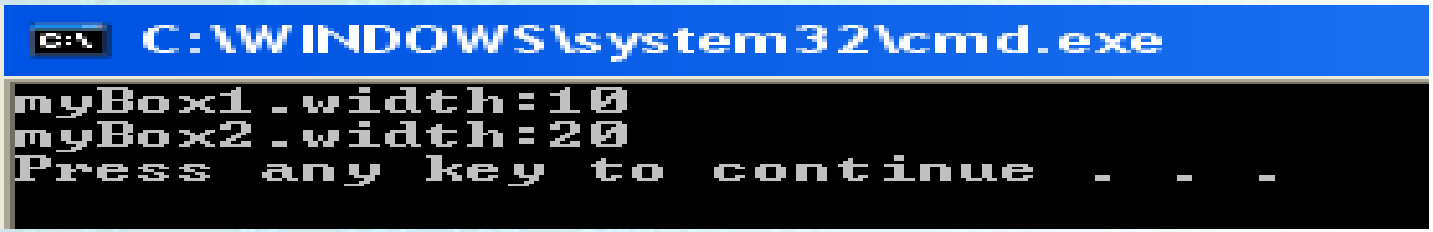
b1 and b2 will refer to the same object.

- ✓ Any changes made to the object through b2 will affect the object b1.
- ✓ If b1 is been set to null, b2 will still point to the original object.

Program(1)

The following program declares two Box objects:

```
class Box {
        int width;
        int height;
        int depth;
        }
public class InstanceMain {
  public static void main(String args[]) {
        Box myBox1 = new Box();
        Box myBox2 = new Box();
                myBox1.width = 10;
                myBox2.width = 20;
                        System.out.println("myBox1.width:" + myBox1.width);
                        System.out.println("myBox2.width:" + myBox2.width);
                   }                    }
```



main statement

public static void main(String args[ ])

- **public -** public everyone can access it.
- **static -** Java environment should be able to call this method without creating an instance of the class , so this method must be declared as static.
- **void -** the return type void means there's no return value.
- **main( ) -** the name of the method, main because it's the main method.
- **String args[ ] -** arguments to this method. This method must be given an array of Strings, and the array will be called 'args'.

method

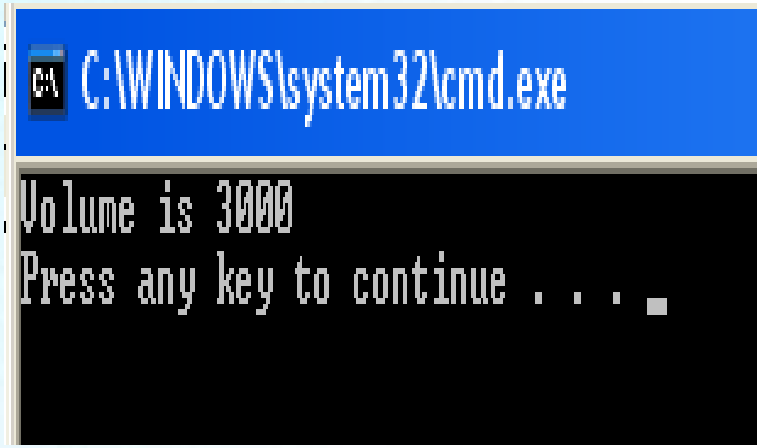
This is the general form of a method:

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

- type specifies the returned type.
- If the method does not return a value, its return type must be void.
- The parameter-list is a sequence of type and identifier pairs separated by commas.
- Parameters receive the arguments passed to the method.
- If the method has no parameters, the parameter list will be empty.

Adding a method to class

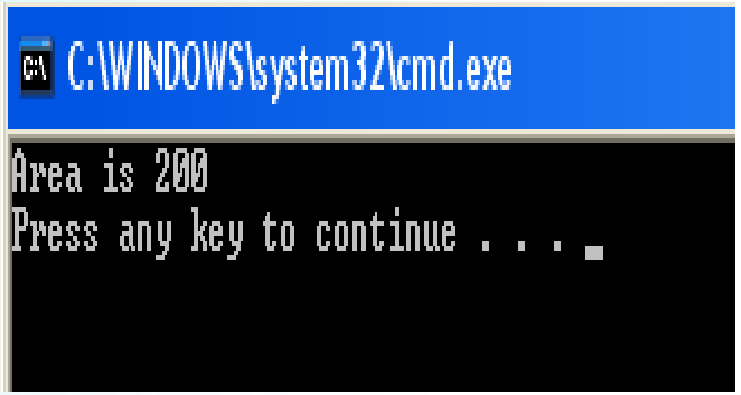
```
class Box {
  int width;
  int height;
  int depth;
            void calculateVolume() {
            System.out.print("Volume is ");
            System.out.println(width * height * depth);
                                    }
 }
public class MethodMain {
  public static void main(String args[]) {
            Box mybox1 = new Box();
            mybox1.width = 10;
            mybox1.height = 20;
            mybox1.depth = 15;
            mybox1.calculateVolume();
                                    }
                        }
```



Returning a value

We can use return statement to return a value to the callers.

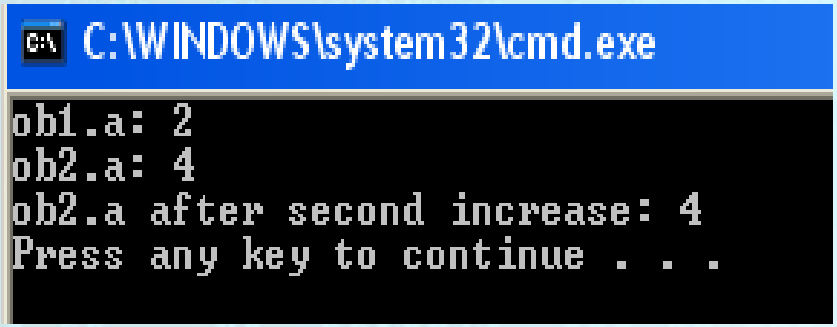
```
class Rectangle {
        int width;
        int height;
                int getArea() {
                        return width * height;
                        }
                }
public class ReturnMain {
public static void main(String args[]) {
        Rectangle mybox1 = new Rectangle();
        int area;
                mybox1.width = 10;
                mybox1.height = 20;
                area = mybox1.getArea();
                System.out.println("Area is " + area);

                }    }
```



Returning object

```
class MyClass {
  int myMemberValue = 2;

  MyClass() {
                }
  MyClass doubleValue() {
    MyClass temp = new MyClass();
    temp.myMemberValue = temp.myMemberValue*2;
    return temp;
  }
}
```

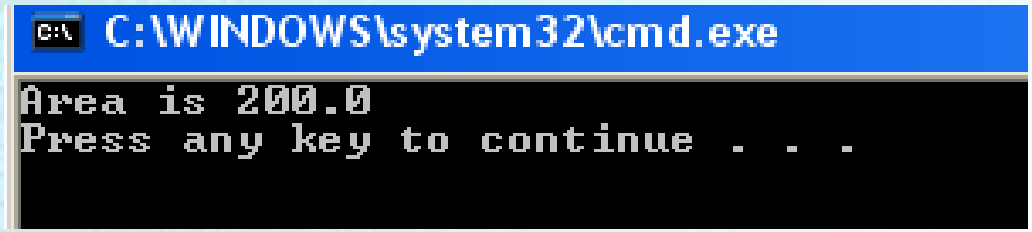
```
public class RetobjMain {
 public static void main(String args[]) {
 MyClass ob1 = new MyClass();
 ob1.myMemberValue =2;
 MyClass ob2;

ob2 = ob1.doubleValue();

System.out.println("ob1.a: " +
ob1.myMemberValue);

 System.out.println("ob2.a: " +
ob2.myMemberValue);

 ob2 = ob2.doubleValue();

 System.out.println("ob2.a after second
increase: " + ob2.myMemberValue);
  }
}
```



Method with parameters

```
class Rectangle {
  double width;
  double height;

  double area() {
    return width * height;
  }

  void setDim(double w, double h) {

    width = w;
    height = h;

  }
}
```

```
public class WithParaMain {

  public static void main(String args[]) {
    Rectangle mybox1 = new Rectangle();
    double vol;
    mybox1.setDim(10, 20);

    vol = mybox1.area();
    System.out.println("Area is " + vol);

  }
}
```

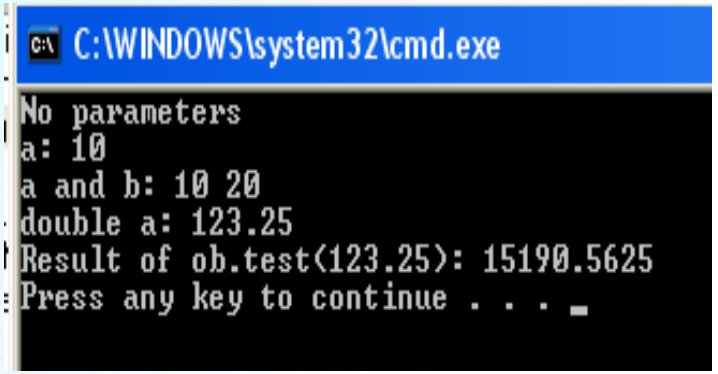


Method overloading

- Overloaded methods have the same name but different parameters.
- Overloaded methods must differ in the type and/or number of their parameters.
- Overloaded methods may have different return types.
- return type alone is insufficient to distinguish two methods.

```
class OverloadDemo {
  void test() {
    System.out.println("No parameters");
  }
  void test(int a) {
    System.out.println("a: " + a);
  }
  void test(int a, int b) {
    System.out.println("a and b: " + a + " " + b);
  }
  double test(double a) {
    System.out.println("double a: " + a);
    return a * a;
  }
}
```

```
public class MeOverloadingMain {
  public static void main(String args[]) {
    OverloadDemo ob = new OverloadDemo();
    ob.test();
    ob.test(10);
    ob.test(10, 20);
    double result = ob.test(123.25);
    System.out.println("Result of ob.test(123.25): " +
result);
  }
}
```

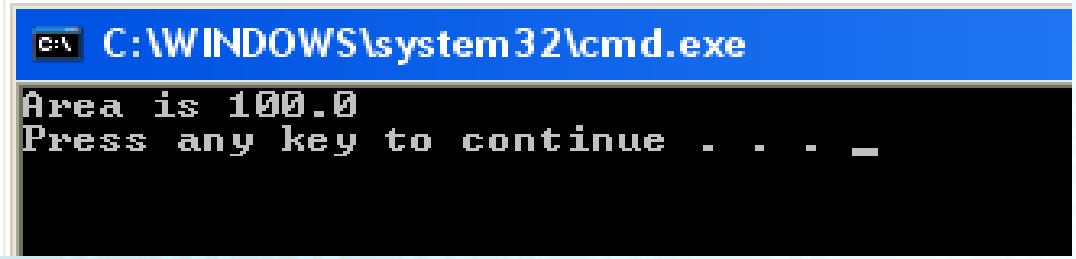


Constructors

- A constructor initializes an object during creation.
- It has the same name as the class.
- Constructors have no return type (not even void).

```
class Rectangle {
                    double width;
                    double height;
          Rectangle() {
                    width = 10;
                    height = 10;
                     }
 double area() {
          return width * height;
             }
           }
```

```
public class ConstructorsMain {
  public static void main(String args[]) {
    Rectangle mybox1 = new Rectangle();
    double area;
    area = mybox1.area();
    System.out.println("Area is " + area);

  }
}
```

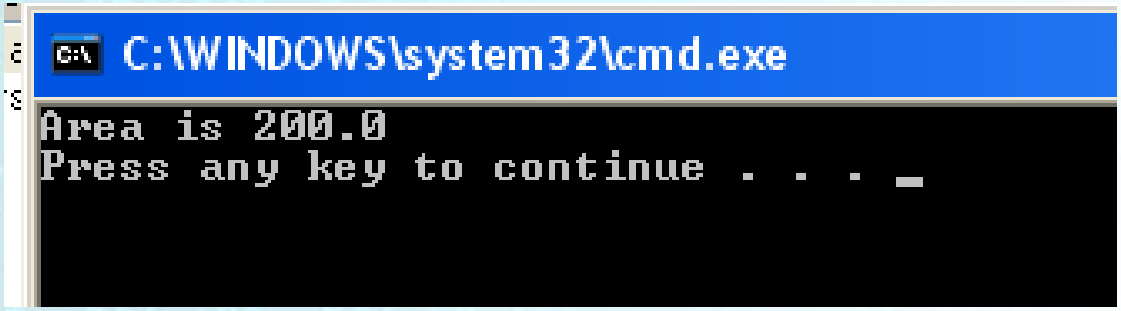


Constructors with Parameters

- The constructors can also have parameters.
- The parameters are used to set the initial states of the object.

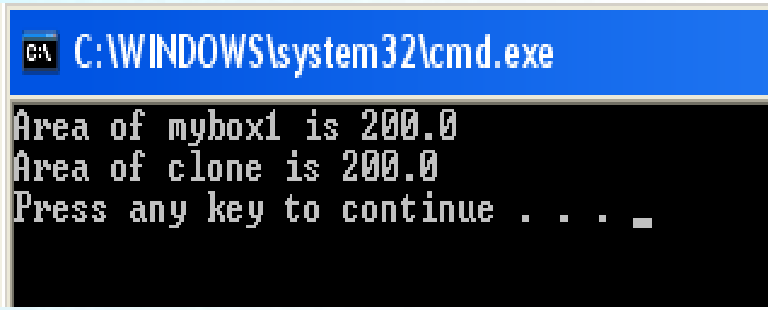
```
class Rectangle {
  double width;
  double height;
  Rectangle(double w, double h) {
                                  width = w;
                                  height = h;
                                  }
  double area() {
              return width * height;
                  }
}
```

```
public class ConstParaMain {
  public static void main(String args[]) {
    Rectangle mybox1 = new Rectangle(10, 20);
    double area;
    area = mybox1.area();
    System.out.println("Area is " + area);
  }
}
```



Constructors with object parameters

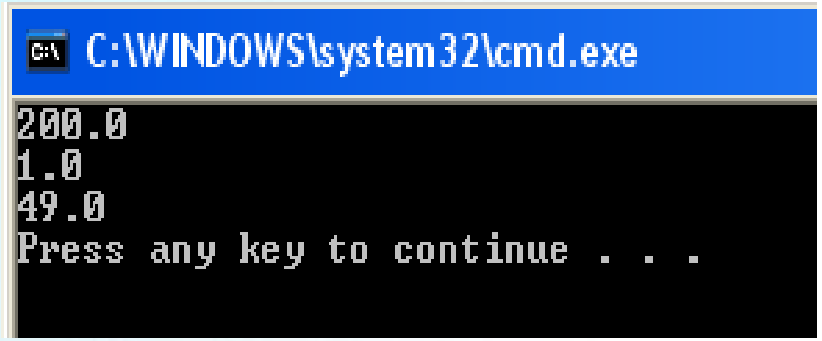
```
class Rectangle {
  double width;
  double height;
  Rectangle(Rectangle ob) {
                        width = ob.width;
                         height = ob.height;
                            }
  Rectangle(double w, double h) {
                                  width = w;
                                  height = h;
                                  }
    Rectangle() {
                        width = -1;
                        height = -1;
                  }
Rectangle(double len) {
                        width = height = len;
                        }
double area() {
                              return width * height;
                     }   }
  public class ConsobjMain {
  public static void main(String args[]) {
    Rectangle mybox1 = new Rectangle(10, 20);
    Rectangle myclone = new Rectangle(mybox1);
    double area;
    area = mybox1.area();
    System.out.println("Area of mybox1 is " +
area);
    area = myclone.area();
    System.out.println("Area of clone is " + area);
  }     }
```



Overloading Constructors

```
class Rectangle {
 double width;
 double height;
 Rectangle(double w, double h) {
                                    width = w;
                                    height = h;
                                    }
 Rectangle() {
              width = -1;
              height = -1;
               }
 Rectangle(double len) {
                         width = height = len;
                         }
 double area() {
                  return width * height;
                  }
}
```

```
public class ConsOverloadingMain {
   public static void main(String args[]) {
   Rectangle mybox1 = new Rectangle(10, 20);
   Rectangle mybox2 = new Rectangle();
   Rectangle mycube = new Rectangle(7);
  double area = mybox1.area();
   System.out.println(area);
   area = mybox2.area();
   System.out.println(area);
   area = mycube.area();
   System.out.println(area);
 }   }
```



this keyword

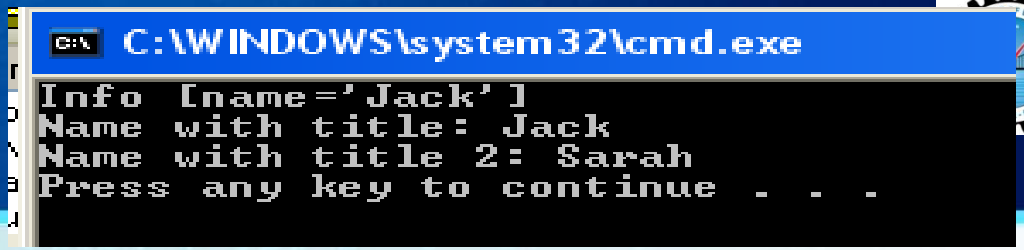
- this refers to the current object.

```
Rectangle(double w, double h) {
        this.width = w;
        this.height = h;
        }
```

- Use *this* to reference the hidden instance variables
- Member variables and method parameters may have the same name.

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

Program(2)




```
class Person{
  private String name;
  private String title;
  private String address;
  public Person() {        }
  public Person(String name) {
    this.name = name;          }
  public String getName() {
    return name;             }
  public void setName(String name) {
    this.name = name;                  }
  public String getNameWithTitle() {
    String nameTitle;
    if (title != null) {
      nameTitle = name + ", " + title;
    }
else {     nameTitle = name;          }
    return nameTitle;
  }
```

```
@Override
  public String toString() {
    return "Info [" +
        "name='" + name + '\'' +
        ']';       }    }
public class ThisMain{
  public static void main(String[] args) {
    Person person = new Person();
    person.setName("Jack");
   System.out.println(person);
  String nameTitle1 = person.getNameWithTitle();
System.out.println("Name with title: " + nameTitle1);

    Person person2 = new Person("Sarah");
String nameTitle2 = person2.getNameWithTitle();
System.out.println("Name with title 2: " + nameTitle2);
  }  }
```

Garbage collection

- Java handles deallocation automatically.
- The automated deallocation is called garbage collection.
- When no references to an object exist, and that object is eligible for garbage collection.

The finalize( ) Method

- The Java calls finalize( ) method whenever it is about to recycle an object.
- finalize() is called just prior to garbage collection.

The finalize( ) method has this general form:

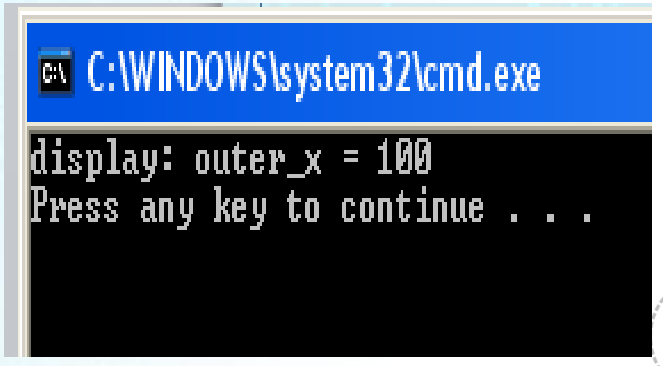
```
protected void finalize( )
{
   // finalization code here
}
```

The keyword protected prevents access to finalize( ) by code defined outside its class.

Inner Classes

- **A class within another class is called nested classes.**

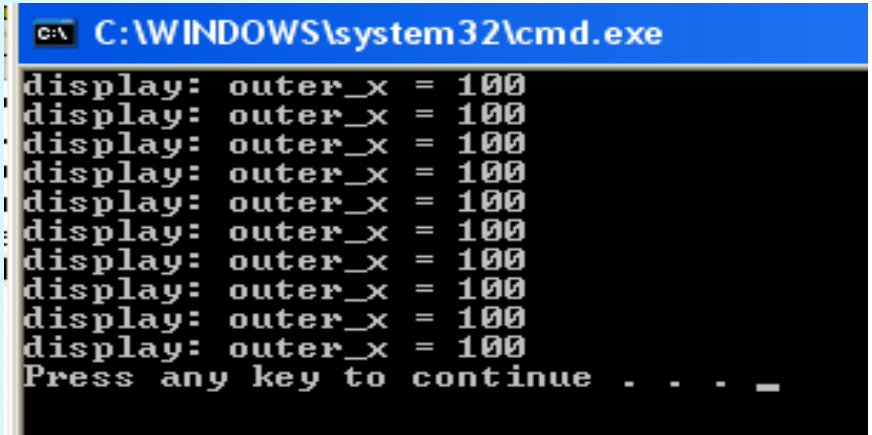
```
class Outer {
        int outer_x = 100;
                void test() {
                        Inner inner = new Inner();
                        inner.display();
                       }
                class Inner {
                        void display() {
                                System.out.println("display: outer_x = " + outer_x);
                                  }
                       }
        }
public class InnerMain {
 public static void main(String args[]) {
                        Outer outer = new Outer();
                        outer.test();
                          }
                }
```



Nested Classes

- We can define a nested class within the block defined by a method.

```
public class NestedMain {
                int outer_x = 100;
                        void test() {
                                for (int i = 0; i < 10; i++) {
                        class Inner {
                                void display() {
                                        System.out.println("display: outer_x = " + outer_x);
                                                }
                                }
                                Inner inner = new Inner();
                                inner.display();
                                }
        }
 public static void main(String args[]) {
NestedMain outer = new NestedMain();
  outer.test();
 }
}
```



Argument Passing

There are two ways that a computer language can pass an argument to subroutine.

The first way is *call-by-value*.

- This method 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.

The second way an argument can be passed is *call-by-reference*.

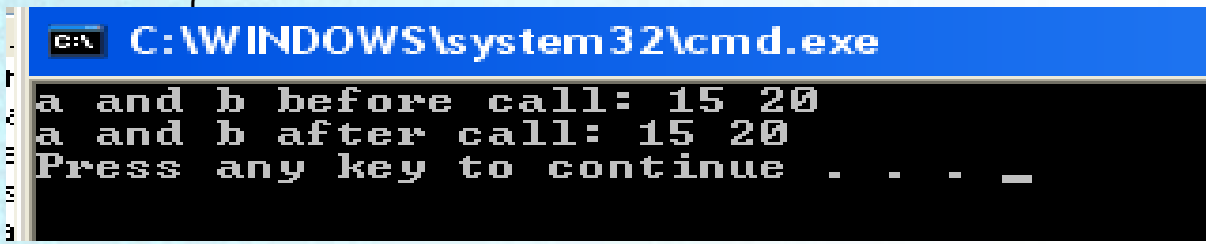
- In this method, a reference to an argument (not the value of the argument) is passed to the parameter.
- This reference is used to access the actual argument specified in the call.
- This means that changes made to the parameter will affect the argument used to call the subroutine.

Java uses both approaches, depending upon what is passed.

1. In Java, when you pass a simple type to a method, it is passed by value.

Program on pass by value

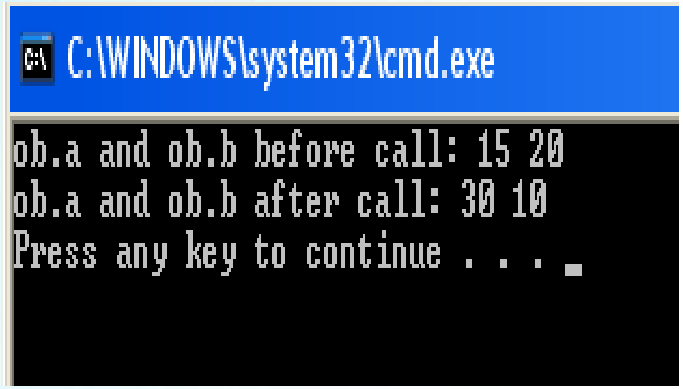
```
class Test {
        void meth(int i, int j) {
                i *= 2;
                j /= 2;
                }
        }
class CallByValue {
public static void main(String args[]) {
                Test ob = new Test();
                int a = 15, b = 20;
                System.out.println("a and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



- ✓ **When you pass an object to a method, the situation changes dramatically, because objects are passed by reference.**

Program on pass by reference

```
class Test {
        int a, b;
        Test(int i, int j) {
                a = i;
                b = j;
                }
// pass an object
        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);
}   }
```

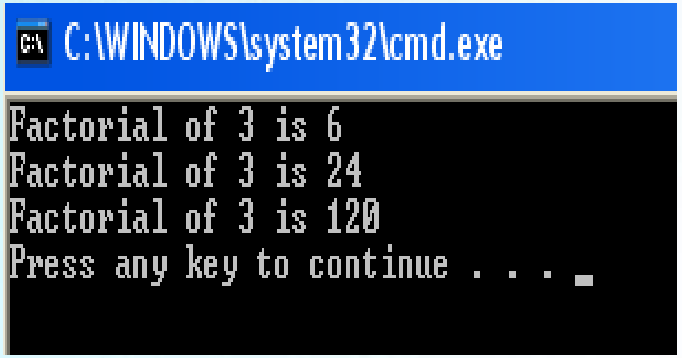


Recursion

❖**Java supports recursion.**

- Recursion is the process of defining something in terms of itself.
- As it relates to java programming, recursion is the attribute that allows a method to call itself.
- *A method that calls itself is said to be recursive*.

```
class Factorial {
    int fact(int n) {
                    int result;
                if ( n ==1) return 1;
                result = fact (n-1) * n;
                return result;
                   }
            }
class Recursion {
    public static void main (String args[]) {
        Factorial f =new Factorial();
System.out.println("Factorial of 3 is " + f.fact(3));
System.out.println("Factorial of 4 is " + f.fact(4));
System.out.println("Factorial of 5 is " + f.fact(5));
}       }
```



Access control

❖ **Java's access specifiers are public, private, protected and a default access level.**

1. A *public* class member can be accessed by any other code.
2. A *private* class member can only be accessed within its class.

3. Default (without an access modifier).

- A class's fields, methods and the class itself may be default.
- A class's default features are accessible to any class in the same package.
- A default method may be overridden by any subclass that is in the same package as the superclass.

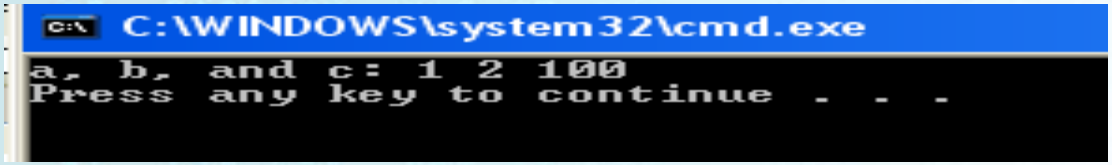
4. *Protected*

- protected features are more accessible than default features.
- Only variables and methods may be declared protected.
- A protected feature of a class is available to all classes in the same package(like a default).
- protected feature of a class can be available to its subclasses.

Program for access control

```
class Test {
  int a;       // default access
  public int b; // public access
  private int c; // private access

  // methods to access c
  void setc(int i) {
          c = i;
  }
  int getc() {
          return c;
  }
}
```

```
public class AccessMain {
  public static void main(String args[]) {
  Test ob = new Test();
    ob.a = 1;
    ob.b = 2;

  // 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());
  }
}
```



What is 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.

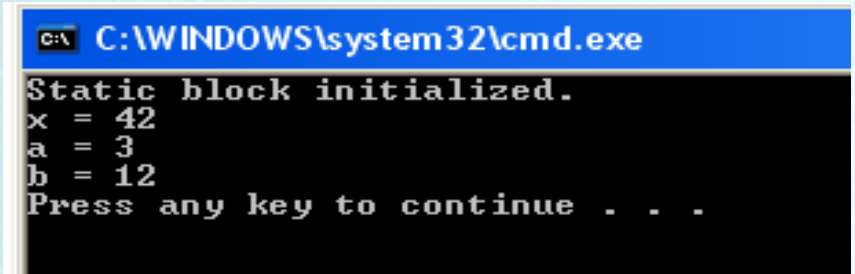
- You can declare both methods and variables to be static.
- The most common example of a static member is main( ). main( ) is declared as static because it must be called before any objects exist.
- Instance variables declared as static are, essentially, global variables.
- When objects of its class 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.
- They cannot refer to this or super in any way.

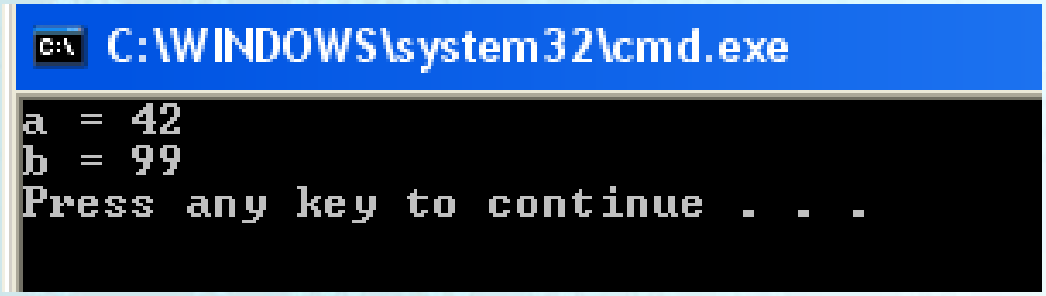
Program on static variables, methods, and block

```
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);
                            }
}
```



program on static method and the static variable are accessed outside of their class.

```
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);
                                }
}
```



final

- A variable can be declared as final.
- Doing so prevents its contents from being modified.
- This means that you must initialize a final variable when it is declared.

For example:

```
1. final int FILE_NEW = 1;
2. final int FILE_OPEN = 2;
3. final int FILE_SAVE = 3;
4. final int FILE_SAVEAS = 4;
5. final int FILE_QUIT = 5;
```

- It is a common coding convention to choose all uppercase identifiers for final variables.
- Variables declared as final do not occupy memory on a per-instance basis.
- a final variable is essentially a constant.
- The keyword final can also be applied to methods, but its meaning is substantially different than when it is applied to variables.

String class

- String is probably the most commonly used class in Java's class library.
- The reason for this is that strings are a very important part of programming.
- The first thing to understand about strings is that every string you create is actually an object of type String.
- Even string constants are actually String objects.

For example, in the statement

System.out.println("This is a String, too");

the string "**This is a String, too**" is a **String constant.**

- objects of type String are immutable; once a String object is created, its contents cannot be altered.

While this may seem like a serious restriction, it is not, for two reasons:

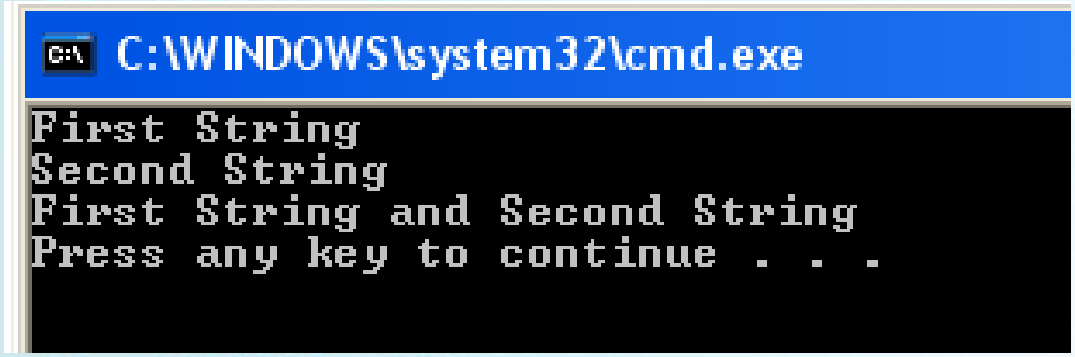
- If you need to change a string, you can always create a new one that contains the modifications.
- Java defines a peer class of String, called StringBuffer, which allows strings to be altered, so all of the normal string manipulations are still available in Java.

Strings can be constructed a variety of ways. The easiest is to use a statement like this:

```
String myString = "this is a test";
```

Program on String class

```
class StringDemo {
public static void main(String args[]) {
String strOb1 = "First String";
String strOb2 = "Second String";
String strOb3 = strOb1 + " and " + strOb2;
System.out.println(strOb1);
System.out.println(strOb2);
System.out.println(strOb3);
}
}
```



String methods

```
class StringDemo2 {
public static void main(String args[]) {
        String strOb1 = "First String";
        String strOb2 = "Second String";
        String strOb3 = strOb1;
        System.out.println("Length of strOb1: " + strOb1.length());
        System.out.println("Char at index 3 in strOb1: " + strOb1.charAt(3));
        if(strOb1.equals(strOb2))
        System.out.println("strOb1 == strOb2");
        else
        System.out.println("strOb1 != strOb2");
        if(strOb1.equals(strOb3))
        System.out.println("strOb1 == strOb3");
        else
        System.out.println("strOb1 != strOb3");
        }
        }
```