UNIT - II

Introducing Classes

and

String Handling

Introducing Classes

- Class Fundamentals
- Declaring objects
- Introducing Methods
- Constructors
- *this* keyword
- Use of objects as parameter & Methods returning objects
- Call by value & Call by reference
- Static variables & methods
- Garbage collection
- Nested & Inner classes.

Introducing Classes

Class Fundamentals:

- Class is the logical construct upon which the entire Java language is built because *it defines the shape and nature of an object*. It defines a new data type.
- The class forms the basis for OOP in Java. Any concept you wish to implement in a Java program must be encapsulated within a class.
- 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.
- The two words ***object*** and ***instance*** used

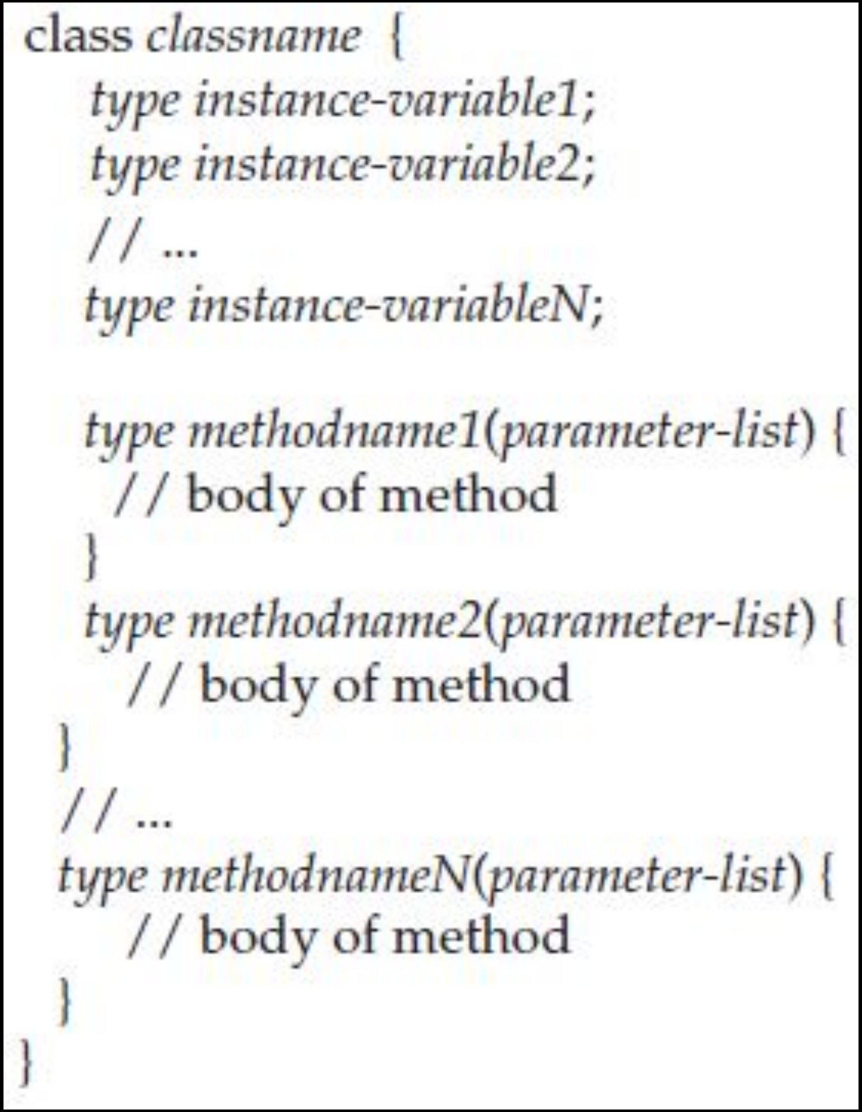
Introducing Classes

Class Fundamentals: ...

- When you define a class, you declare its exact form and nature. You do this by specifying the data that it contains and the code that operates on that data. While very simple classes may contain only code or only data, ***most real-world classes contain both***.
- A class is declared by use of the ***class*** keyword.

Introducing Classes

Class Fundamentals: … *A simplified general form*



```
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. In most classes, the instance variables are acted upon and accessed* by the methods defined for that class.
- Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another.

Introducing Classes

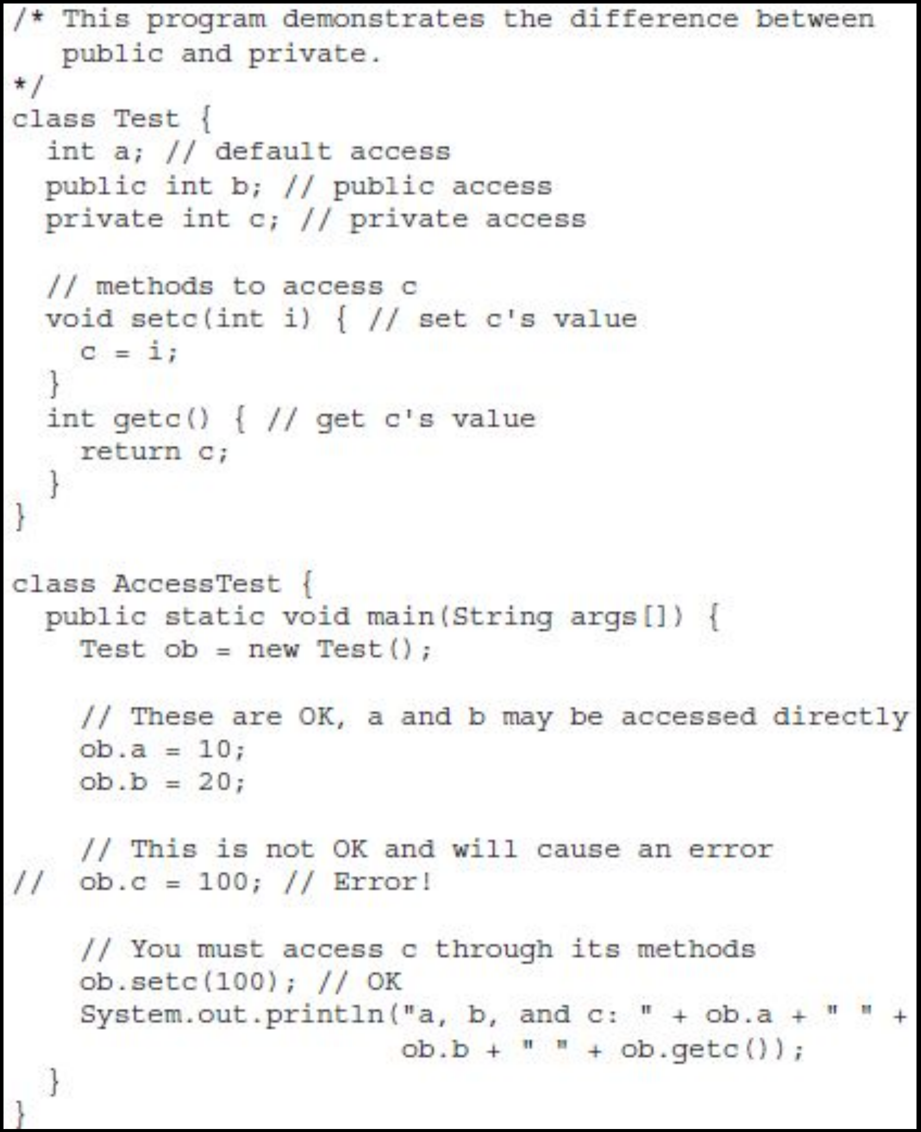
Class Fundamentals: ...

NOTE:

- *C++ programmers will notice that the class declaration and the implementation of the methods are stored in the same place and not defined separately. This sometimes makes for very large .java files, since any class must be entirely defined in a single source file.*
- *This design feature was built into Java because it was felt that in the long run, having specification, declaration, and implementation all in one place makes for code that is easier to maintain.*

Introducing Classes

Class Fundamentals: … *Introducing Access Control*



```
/* This program demonstrates the 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());
  }
}
```

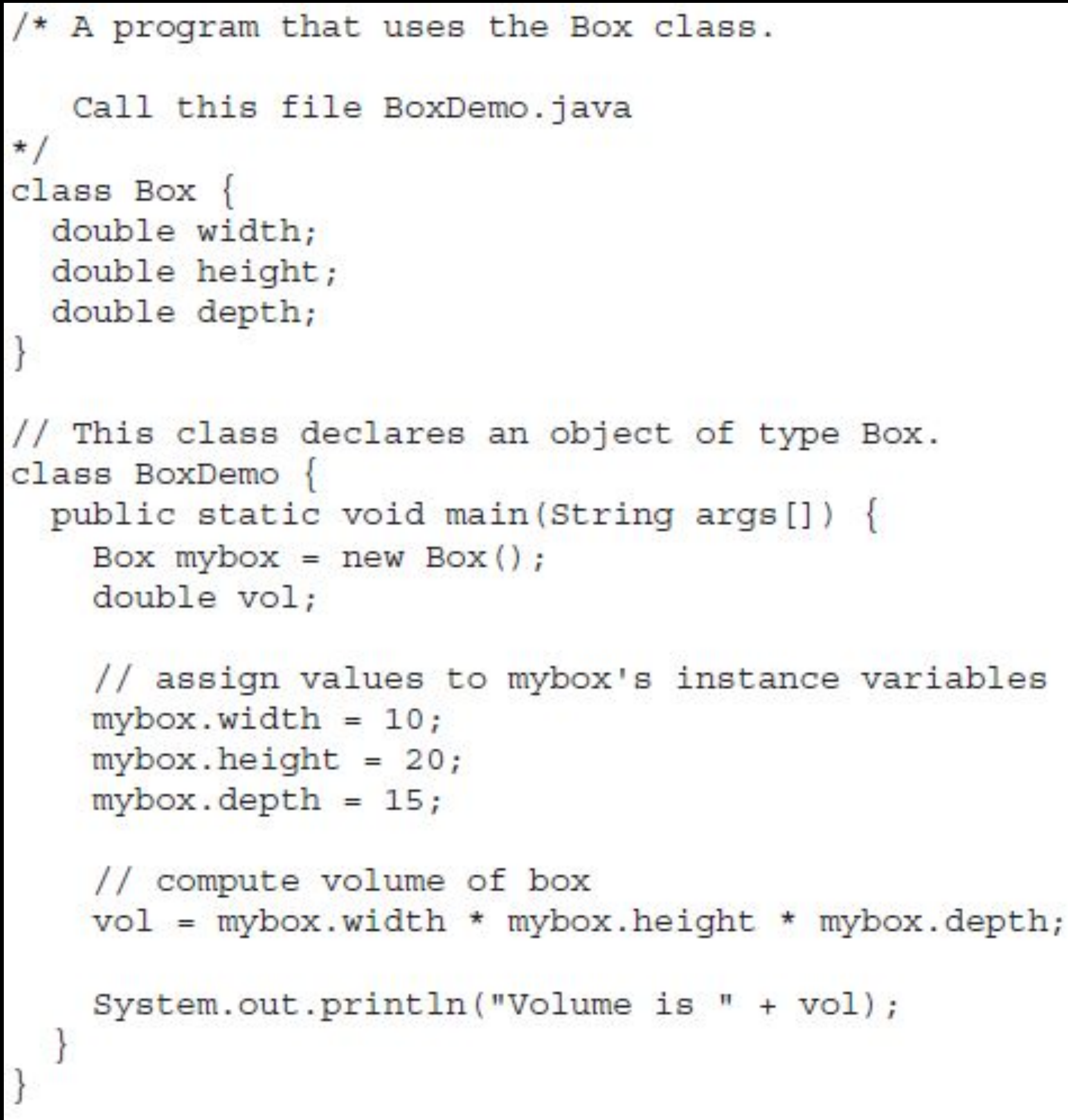
- Java's access specifiers are **public, private,** and **protected.** Java also defines a default access level.
- **protected** applies only when inheritance is involved.

Introducing Classes

Class Fundamentals:

Simple Class: ...

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

Box mybox = new Box();

mybox.width = 100;
```

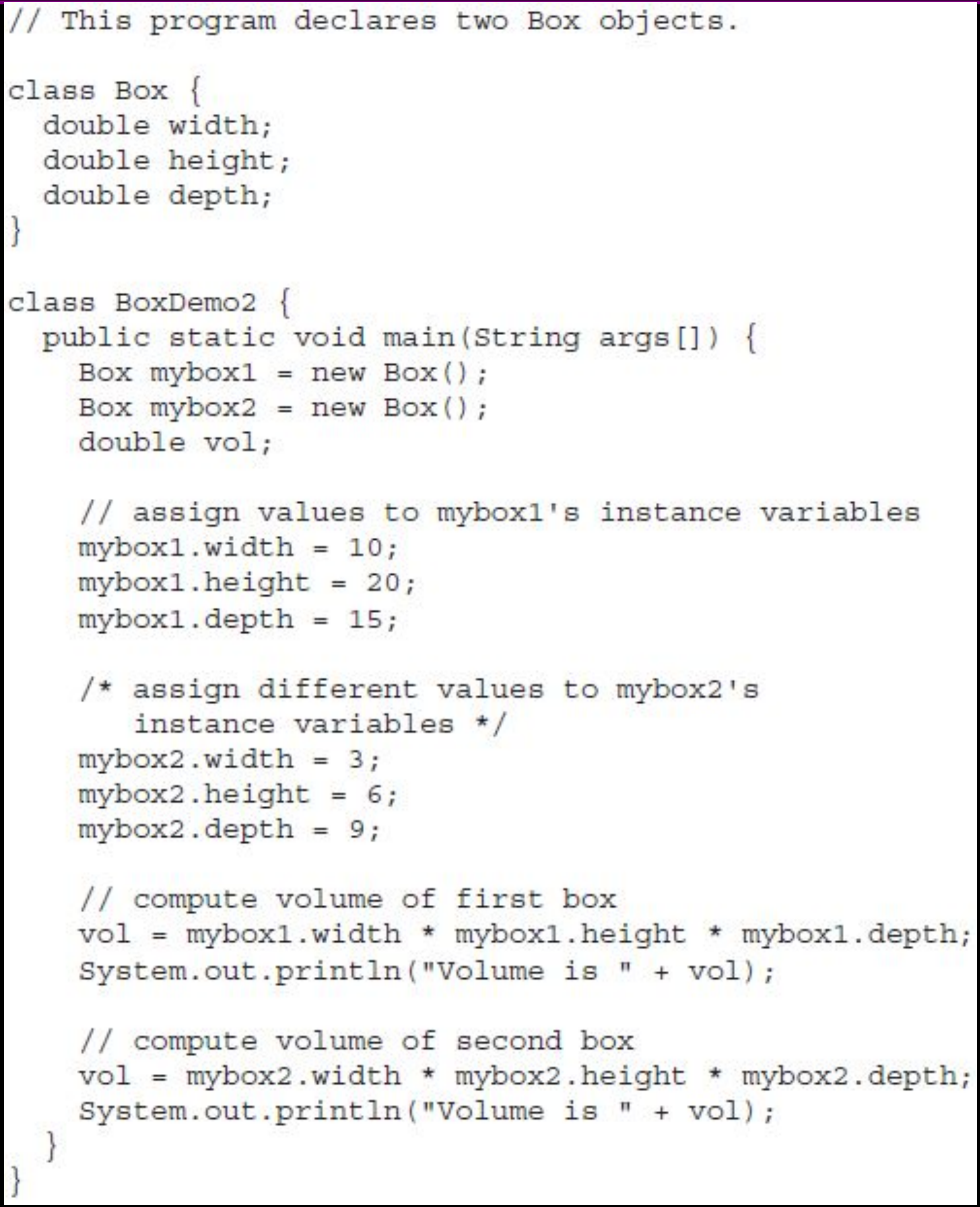


```
/* A program that uses the Box class.

   Call this file BoxDemo.java
*/
class Box {
  double width;
  double height;
  double depth;
}

// This class declares an object of type Box.
class BoxDemo {
  public static void main(String args[]) {
    Box mybox = new Box();
    double vol;

    // assign values to mybox's instance variables
    mybox.width = 10;
    mybox.height = 20;
    mybox.depth = 15;

    // compute volume of box
    vol = mybox.width * mybox.height * mybox.depth;

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

Introducing Classes

Class Fundamentals: ...

Simple Class: ...



```
// This program declares two Box objects.

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

class BoxDemo2 {
  public static void main(String args[]) {
    Box mybox1 = new Box();
    Box mybox2 = new Box();
    double vol;

    // assign values to mybox1's instance variables
    mybox1.width = 10;
    mybox1.height = 20;
    mybox1.depth = 15;

    /* assign different values to mybox2's
       instance variables */
    mybox2.width = 3;
    mybox2.height = 6;
    mybox2.depth = 9;

    // compute volume of first box
    vol = mybox1.width * mybox1.height * mybox1.depth;
    System.out.println("Volume is " + vol);

    // compute volume of second box
    vol = mybox2.width * mybox2.height * mybox2.depth;
    System.out.println("Volume is " + vol);
  }
}
```

Introducing Classes

Declaring objects:

- Obtaining objects of a class is a two-step process.
  - Declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can *refer to an object.*
  - *Acquire an actual,* physical copy of the object and assign it to that variable. (using the **new** operator)
- The **new** operator dynamically allocates (allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by **new.**
- This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated.

1. Box mybox; // declare reference to object

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

ClassName Object_Name = new ClassName(); OR

ClassName class-var = new ClassName();

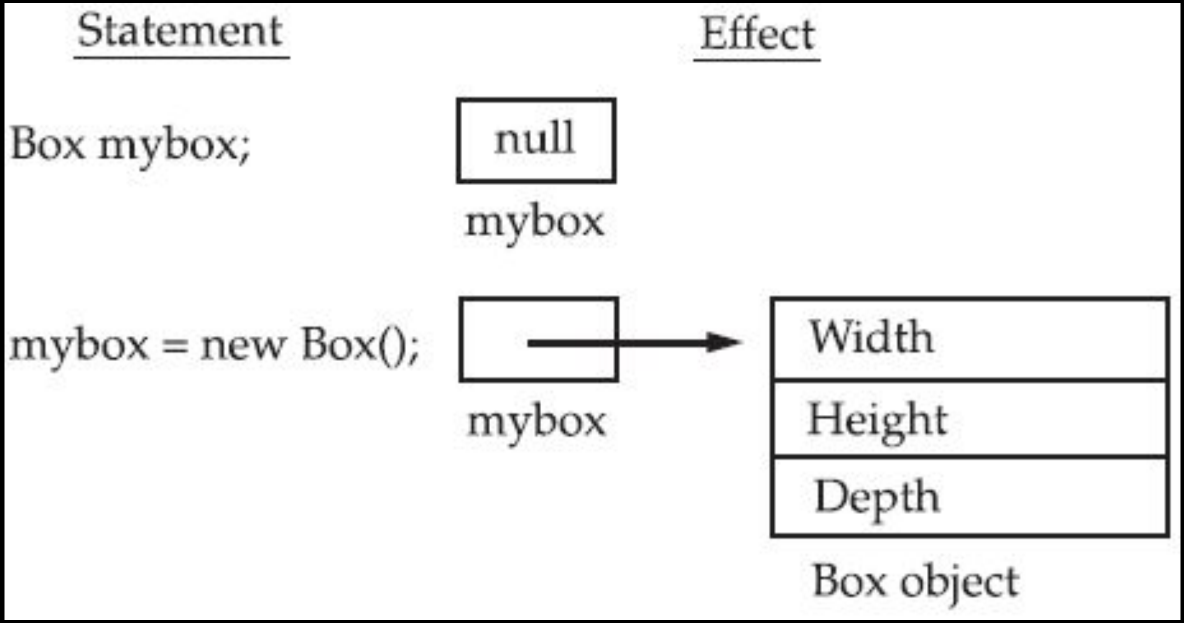
Box mybox = new Box();

Introducing Classes

Declaring objects: ...

A Closer Look at ***new* :** operator dynamically allocates memory for an object.

General form: ***class-var = new classname( );***



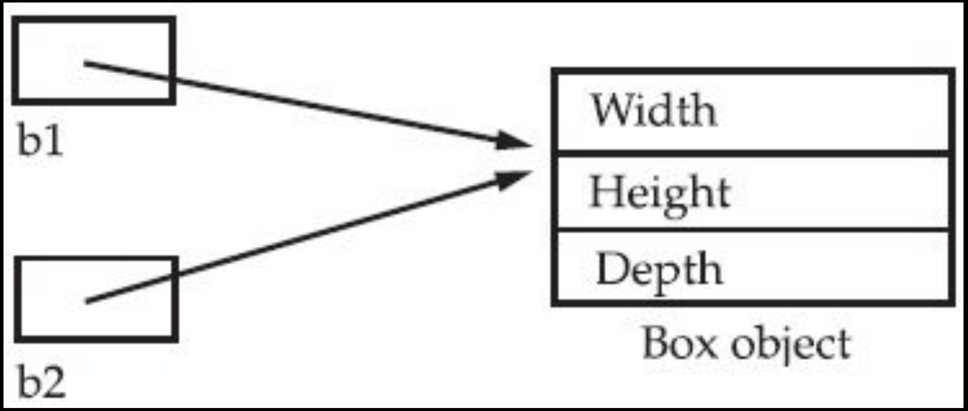
Note: *An object reference is similar to a memory pointer. The main difference—and the key to Java's safety—is that you cannot manipulate references as you can actual pointers. Thus, you cannot cause an object reference to point to an arbitrary memory location or manipulate it like an integer.*

Introducing Classes

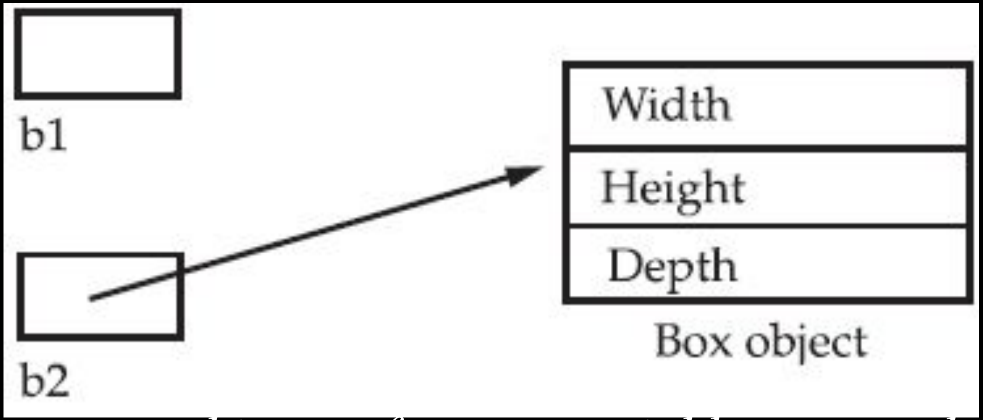
Declaring objects: ...

Assigning Object Reference Variables:

Box b1 = new Box();
Box b2 = b1;



Box b1 = new Box();
Box b2 = b1;
// …
b1 = null;



REMEMBER *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.*

Introducing Classes

Introducing Methods:

- Classes consist of two things: *instance variables and methods.*

General form:

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

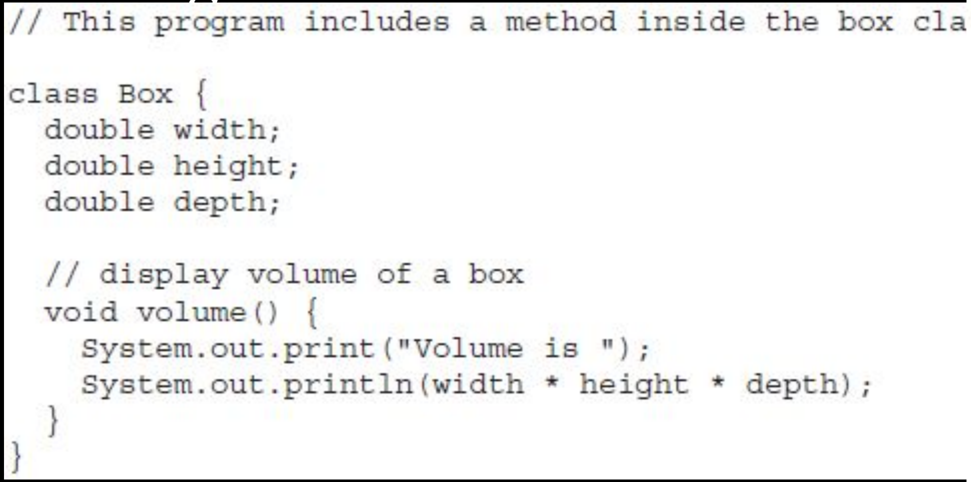
- Here, *type specifies the type of data returned by the method. This can be any valid type,* including class types that you create. If the method does not return a value, its return type must be **void.** The name of the method is specified by *name. This can be any legal identifier* other than those 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;*

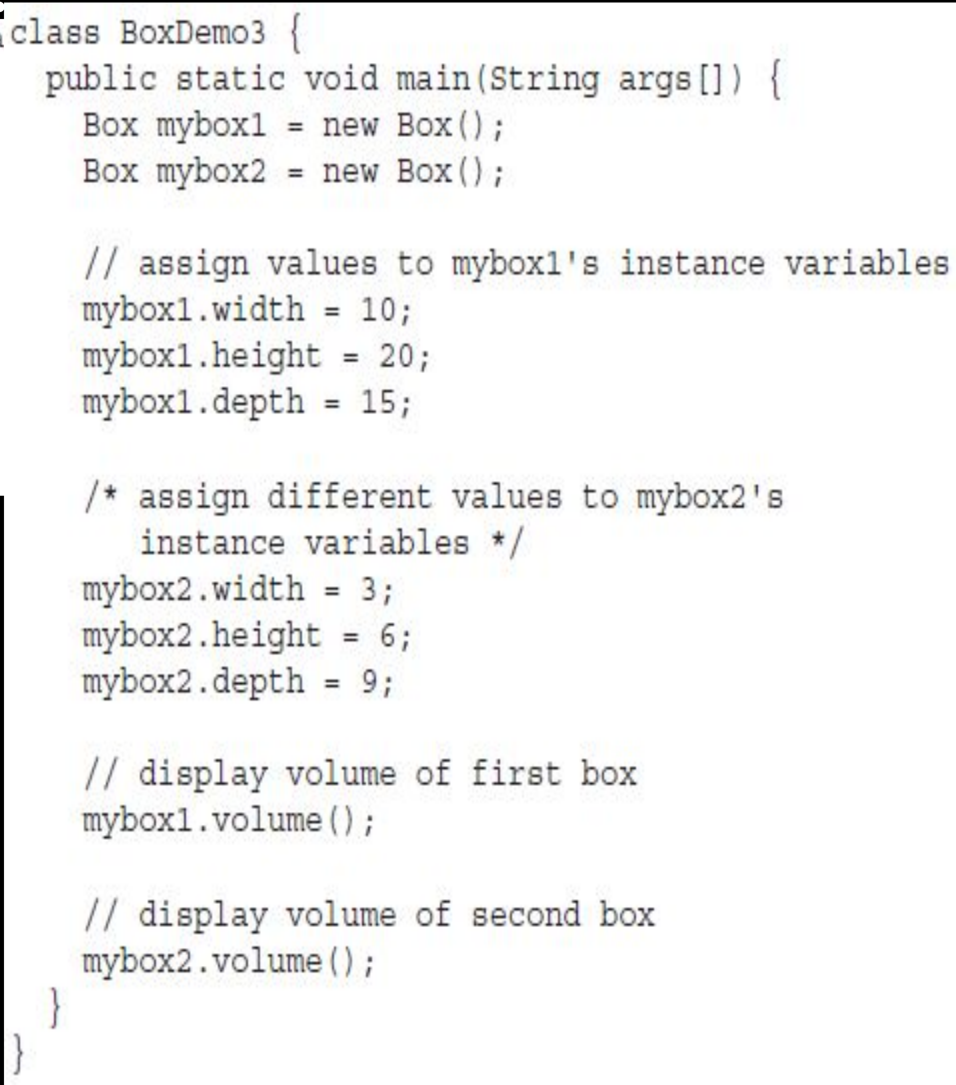
Introducing Classes

Introducing Methods: ...

Adding a Method to the Box Class:



```
// This program includes a method inside the box cla

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

  // display volume of a box
  void volume() {
    System.out.print("Volume is ");
    System.out.println(width * height * depth);
  }
}
```

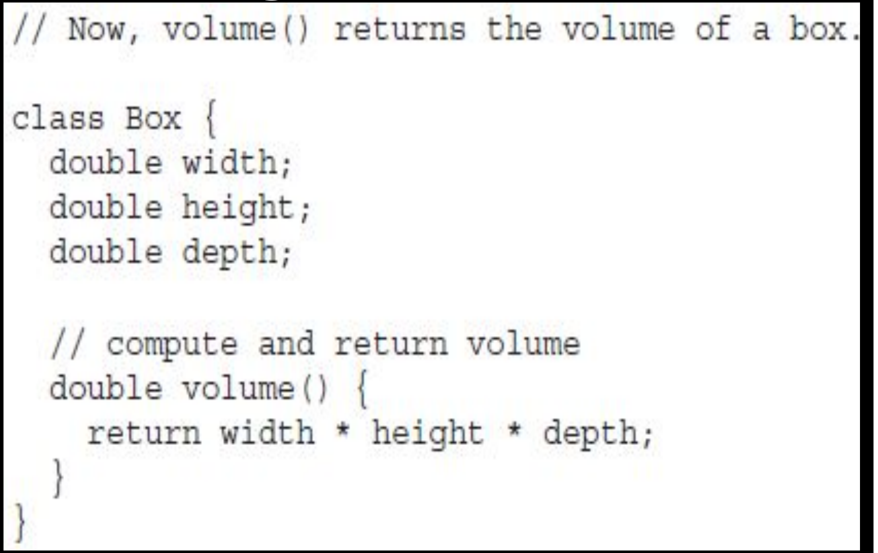


```
class BoxDemo3 {
  public static void main(String args[]) {
    Box mybox1 = new Box();
    Box mybox2 = new Box();

    // assign values to mybox1's instance variables
    mybox1.width = 10;
    mybox1.height = 20;
    mybox1.depth = 15;

    /* assign different values to mybox2's
       instance variables */
    mybox2.width = 3;
    mybox2.height = 6;
    mybox2.depth = 9;

    // display volume of first box
    mybox1.volume();

    // display volume of second box
    mybox2.volume();
  }
}
```

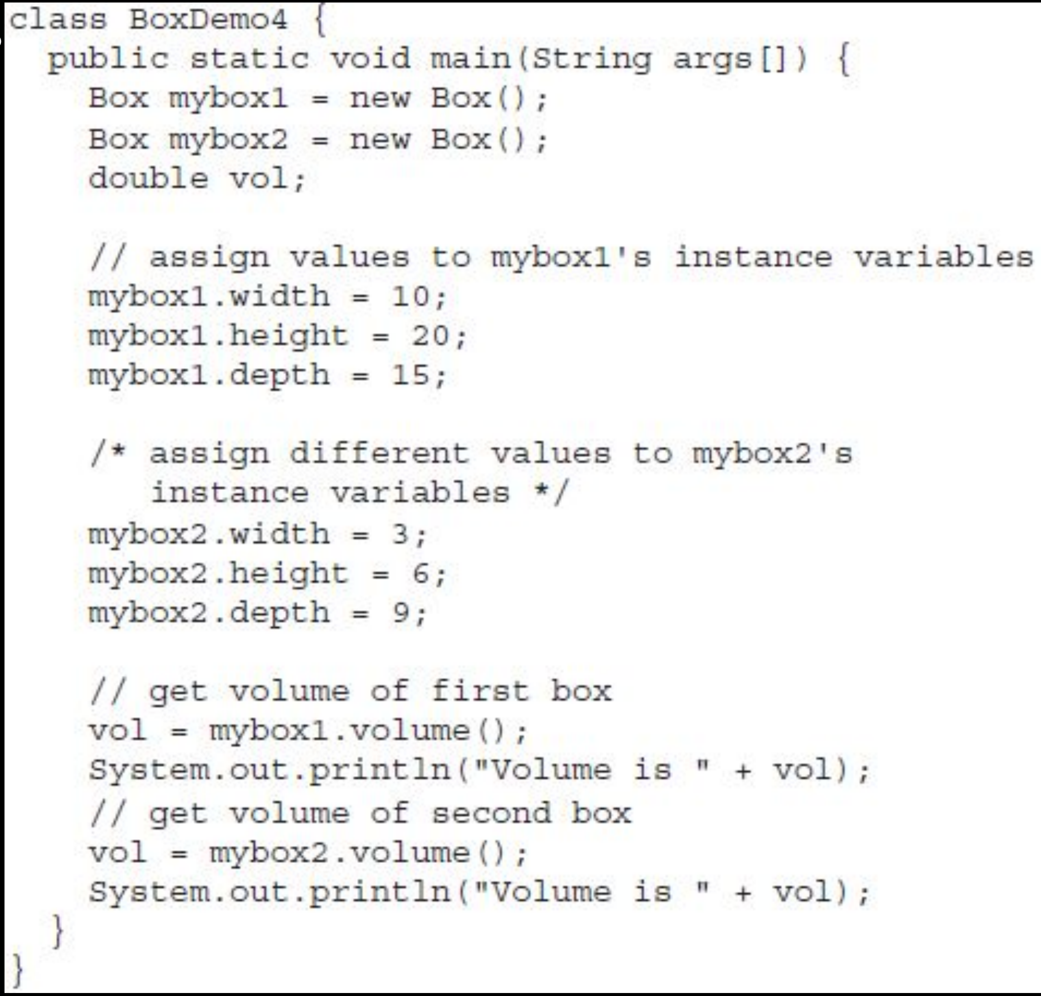
Introducing Classes

Introducing Methods:...

Returning a Value:



```
// Now, volume() returns the volume of a box.

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

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



```
class BoxDemo4 {
  public static void main(String args[]) {
    Box mybox1 = new Box();
    Box mybox2 = new Box();
    double vol;

    // assign values to mybox1's instance variables
    mybox1.width = 10;
    mybox1.height = 20;
    mybox1.depth = 15;

    /* assign different values to mybox2's
       instance variables */
    mybox2.width = 3;
    mybox2.height = 6;
    mybox2.depth = 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);
  }
}
```

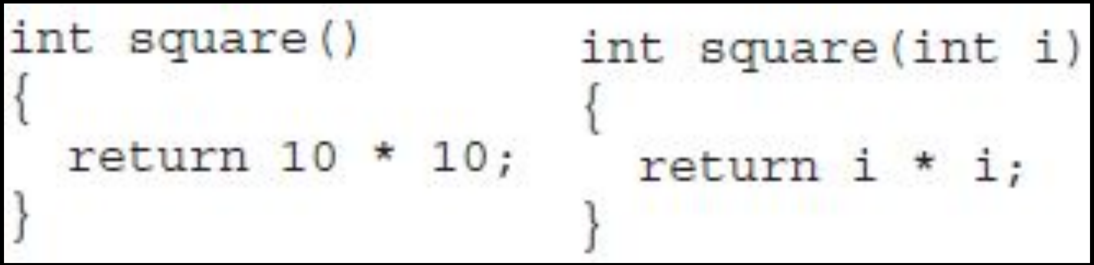
vol = mybox1.volume(); can be replaced by

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

Introducing Classes

Introducing Methods: ...

- **Adding a Method That Takes Parameters:** Parameters allow a method to be generalized. A parameterized method can operate on a variety of data and/or be used in a number of slightly different situations.



```
int square()
{
  return 10 * 10;
}
```

```
int square(int i)
{
  return i * i;
}
```

Example:

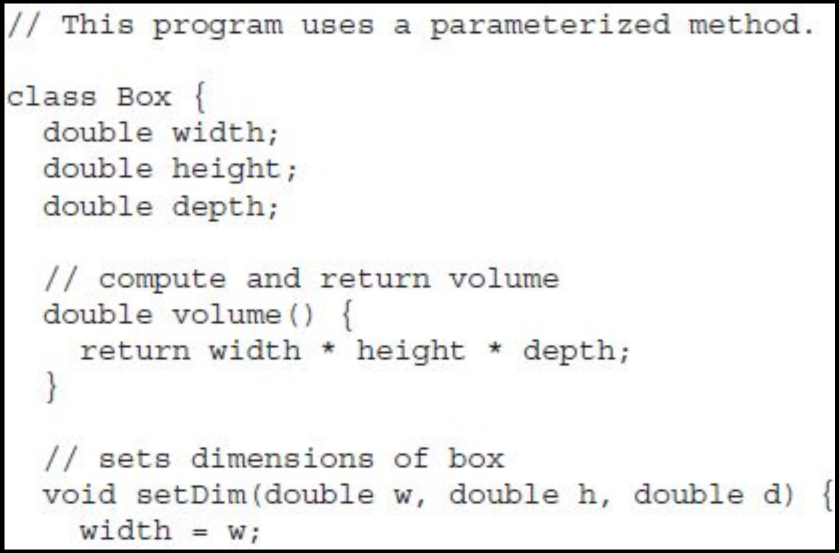
int x, y;

x = square(5); // x equals 25

x = square(9); // x equals 81

y = 2;

x = square(y); // x equals 4

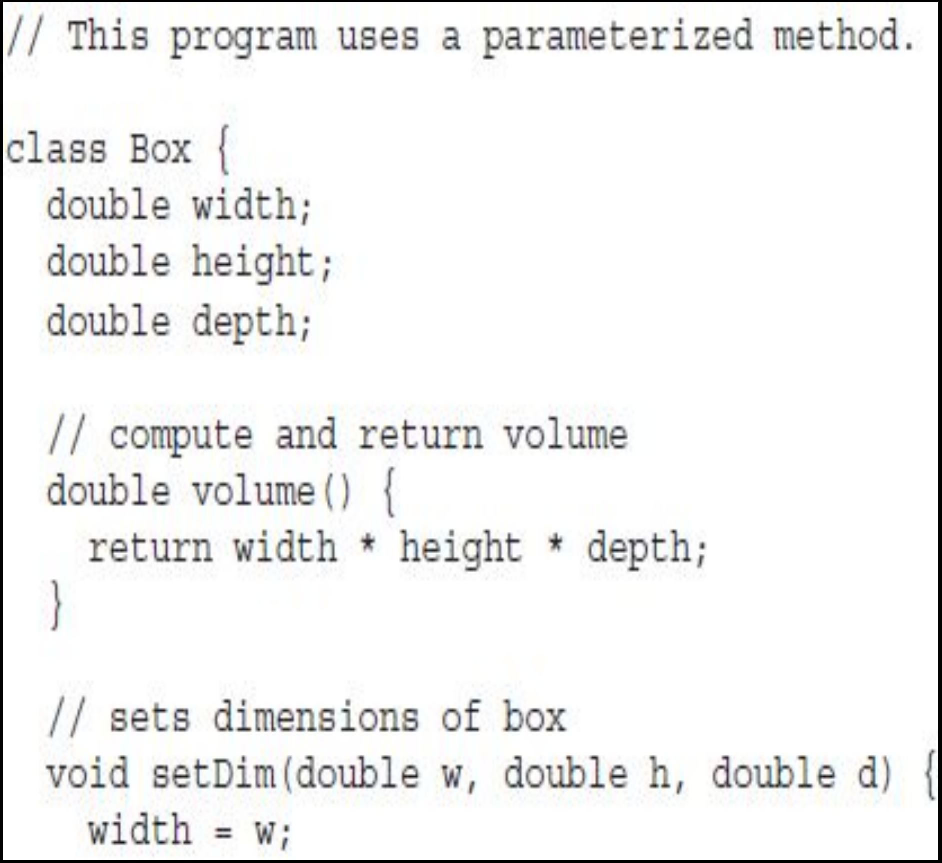


```
// This program uses a parameterized method.

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;
```

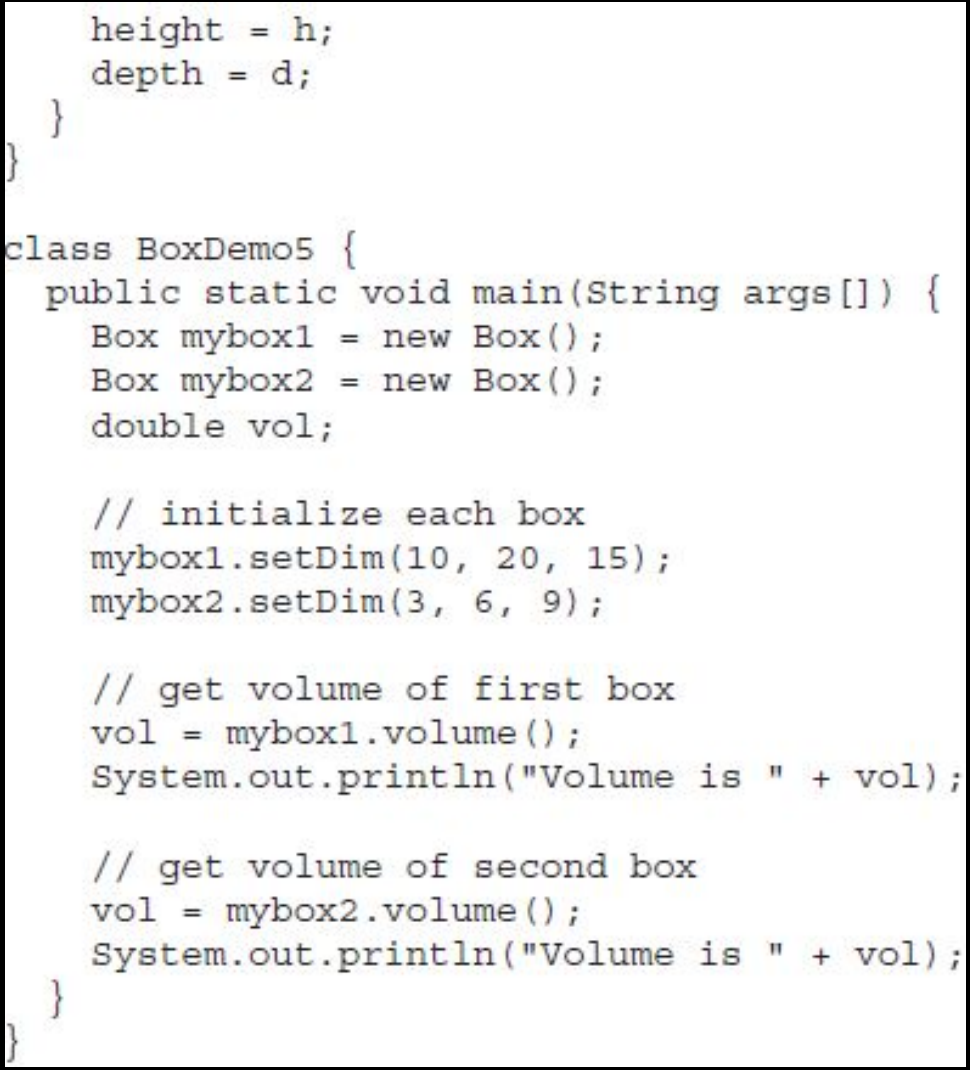
Introducing Classes

Introducing Methods: ...

- Adding a Method That Takes Parameters: ...



```
// This program uses a parameterized method.

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;
```



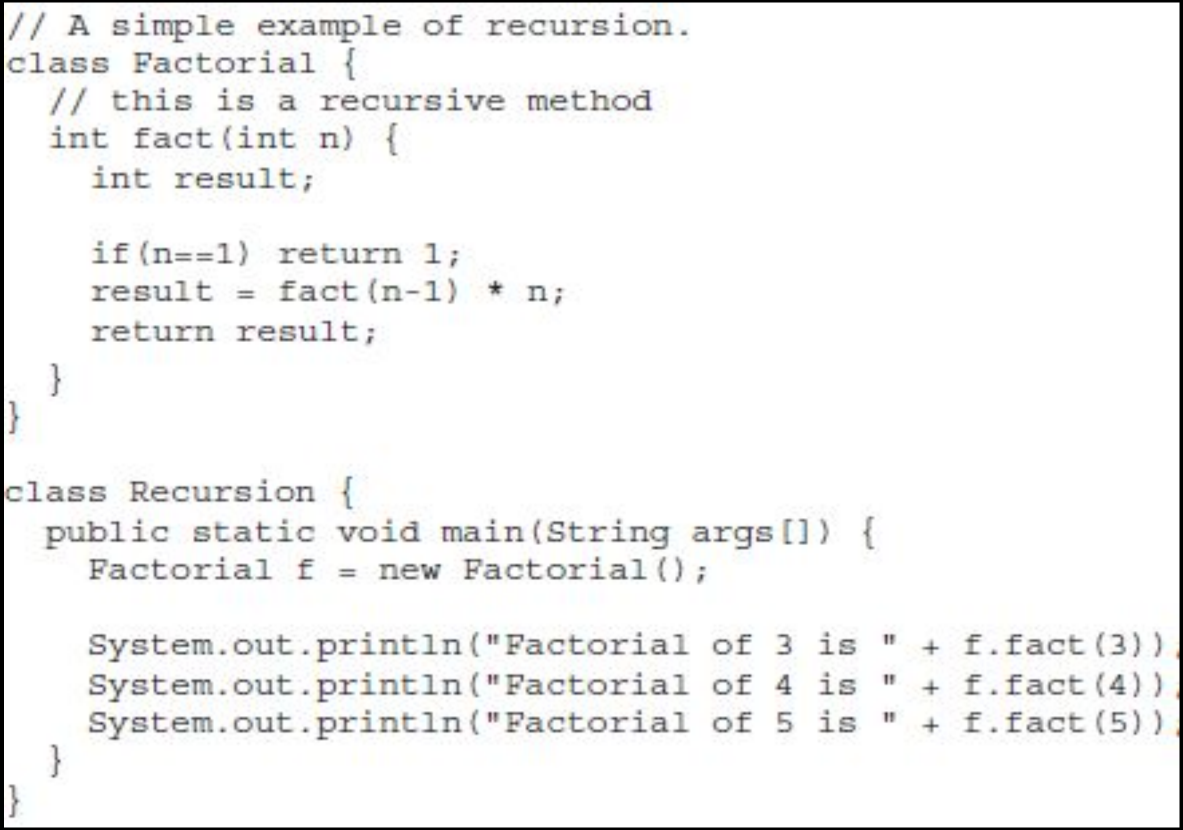
```
    height = h;
    depth = d;
  }
}

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);
  }
}
```

Note: ***The concepts of the method invocation, parameters, and return values are fundamental to Java programming.***

Introducing Classes

Introducing Methods: ... 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*.



```
// A simple example of recursion.
class Factorial {
  // this is a recursive method
  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))
  }
}
```

Introducing Classes

Introducing Methods: ... *Overloading*

- 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.
- Method overloading is one of the ways that Java supports polymorphism.
- Method overloading is one of Java's most exciting and useful features.
- 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.

Introducing Classes

Introducing Methods: … *Overloading* …

Method Overloading by changing the no. of arguments

```
class Calculation{

    void sum(int a, int b){
        System.out.println(a+b);
    }

    void sum(int a, int b, int c){
        System.out.println(a+b+c);
    }
  public static void main(String args[]){
        Calculation obj = new Calculation();
        obj.sum(10,10,10);
        obj.sum(20,20);
  }
}
```

Introducing Classes

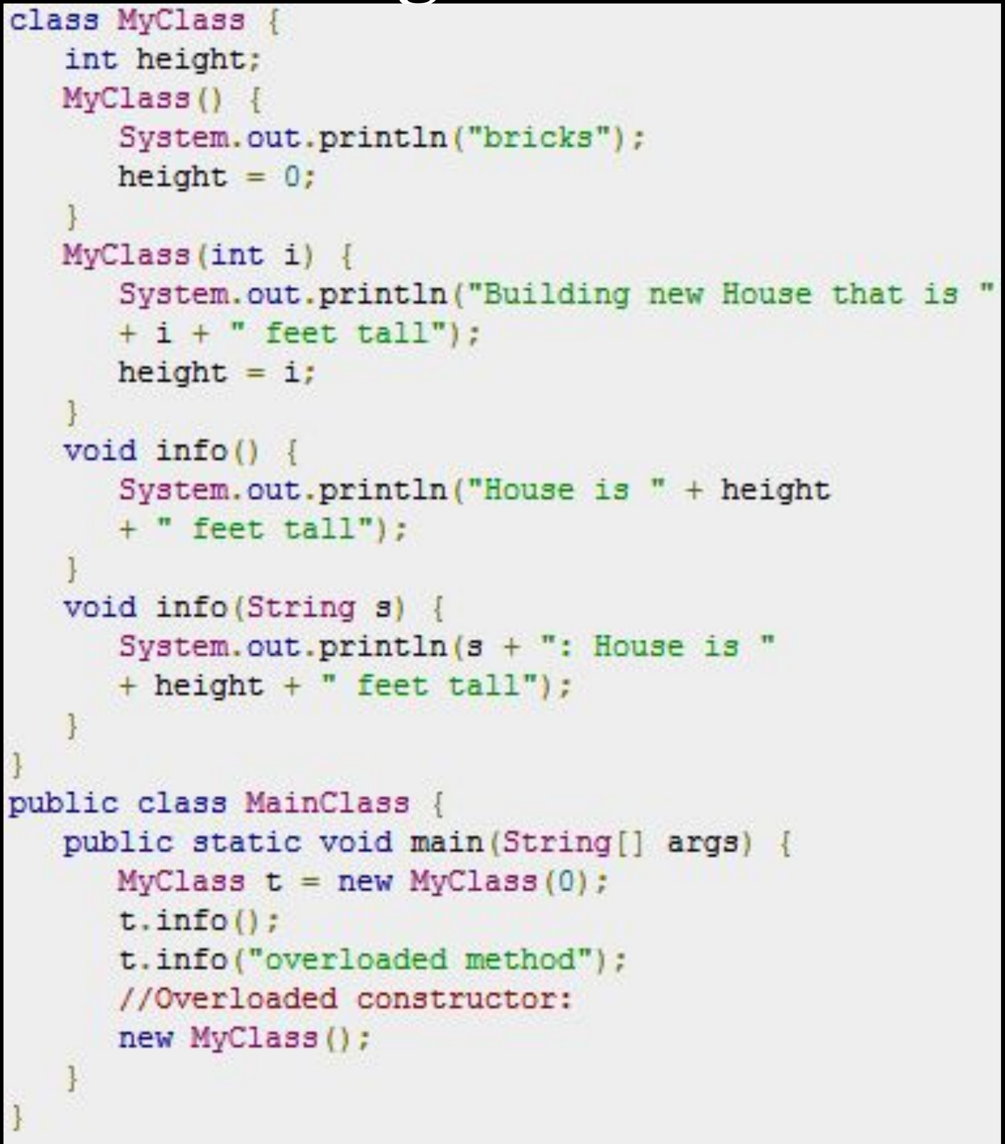
Introducing Methods: ... *Overloading* ...

Method Overloading by changing data type of argument

```
class Calculation2{
 void sum(int a, int b){
     System.out.println(a + b);
 }

 void sum(double a, double b){
     System.out.println(a + b);
 }

 public static void main(String args[]){
       Calculation2 obj = new Calculation2();
       obj.sum(10.5,10.5);
       obj.sum(20,20);
  }
}
```

Introducing Classes

Introducing Methods: ... *Overloading* ...



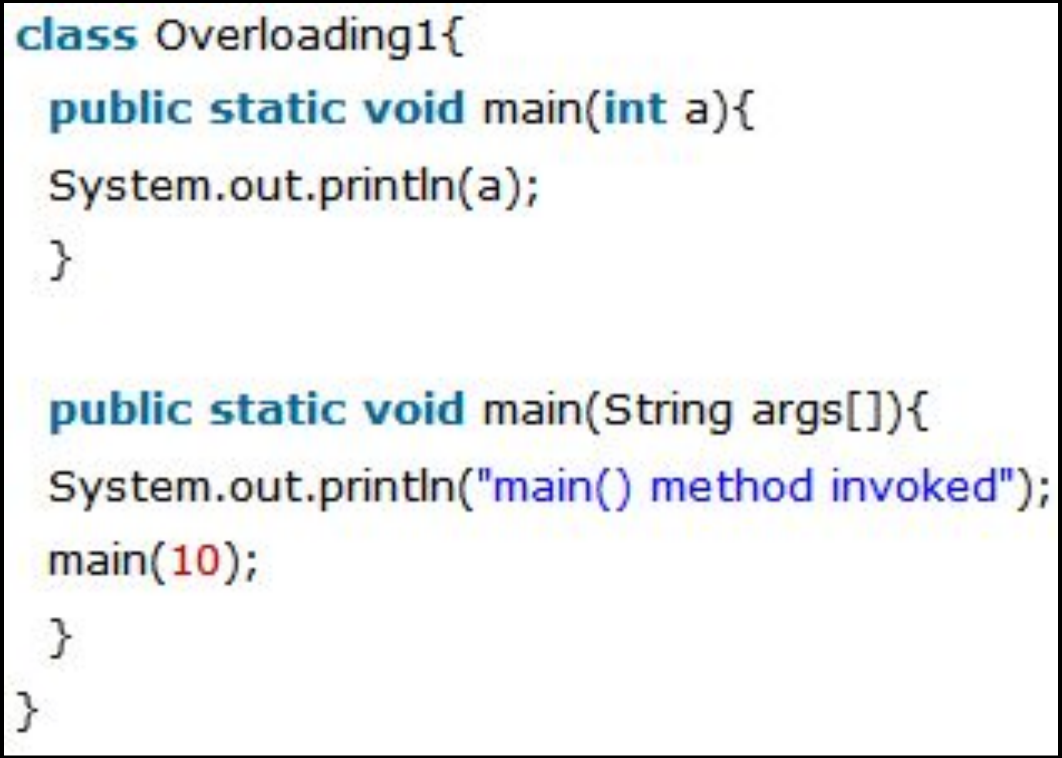
```
class MyClass {
   int height;
   MyClass() {
      System.out.println("bricks");
      height = 0;
   }
   MyClass(int i) {
      System.out.println("Building new House that is "
      + i + " feet tall");
      height = i;
   }
   void info() {
      System.out.println("House is " + height
      + " feet tall");
   }
   void info(String s) {
      System.out.println(s + ": House is "
      + height + " feet tall");
   }
}
public class MainClass {
   public static void main(String[] args) {
      MyClass t = new MyClass(0);
      t.info();
      t.info("overloaded method");
      //Overloaded constructor:
      new MyClass();
   }
}
```

Method/Constructor Overloading by changing number of argument

Introducing Classes

Introducing Methods: ... *Overloading* ...

Can we overload main() method?

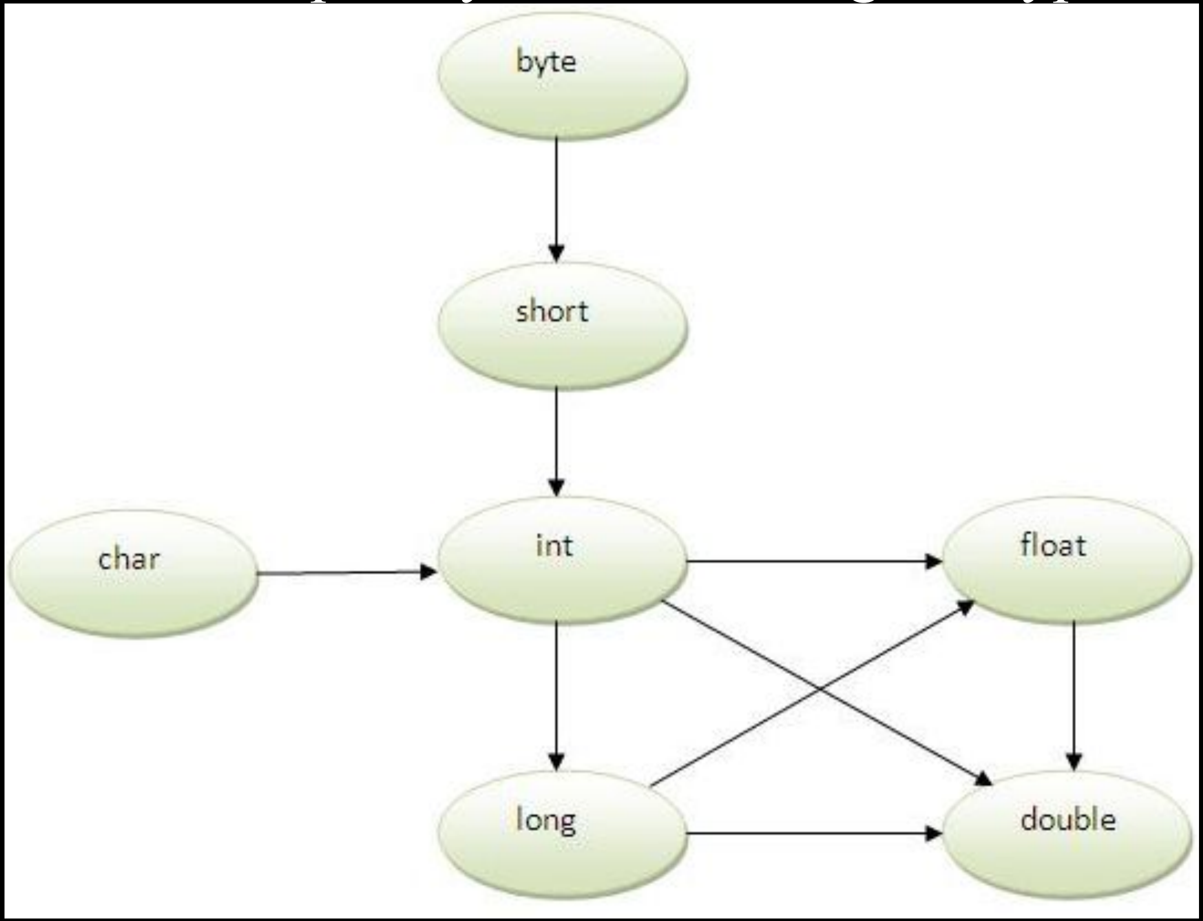


```
class Overloading1{
  public static void main(int a){
  System.out.println(a);
  }

  public static void main(String args[]){
  System.out.println("main() method invoked");
  main(10);
  }
}
```

Introducing Classes

Introducing Methods: ... *Overloading* ...

Method Overloading and Type Promotion: One type is promoted to another implicitly if no matching datatype is found.

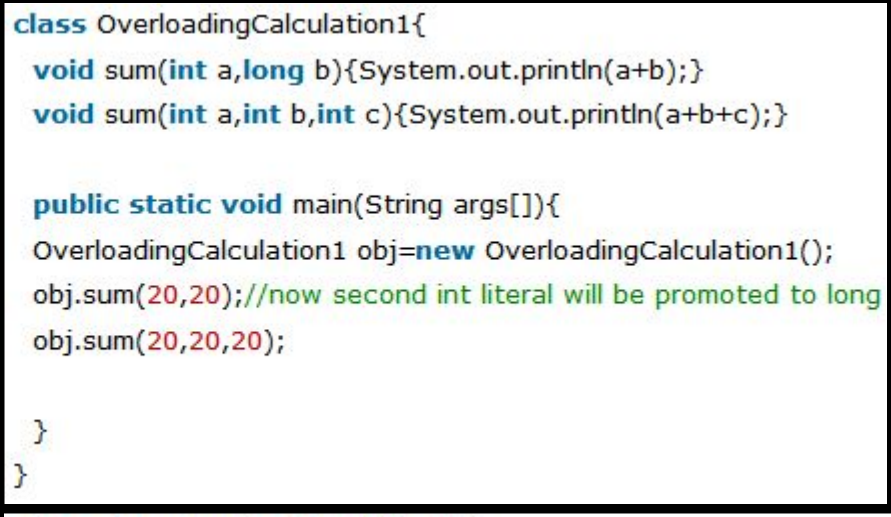


byte can be promoted to short, int, long, float or double. The short datatype can be promoted to int,long,float or double. The char datatype can be promoted to int, long, float or double and so on.

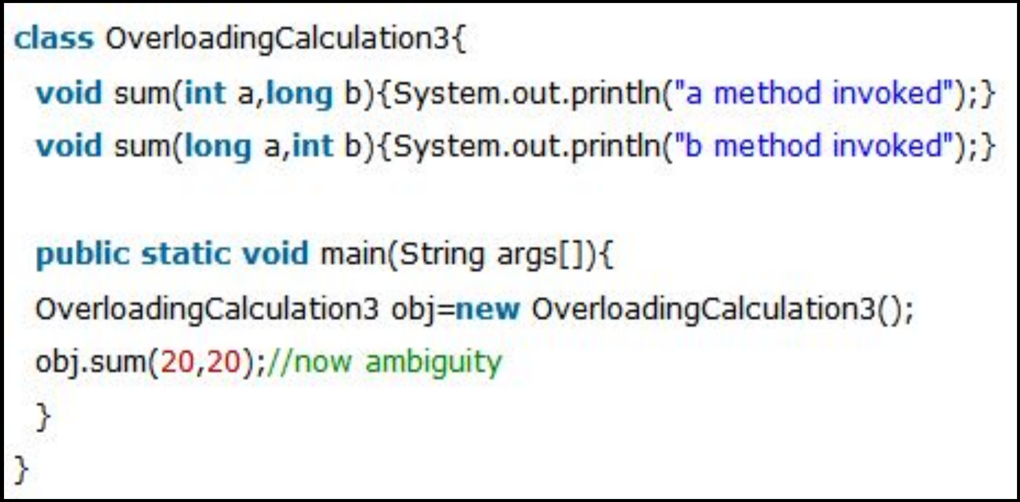
Introducing Classes

Introducing Methods: ... *Overloading* ...

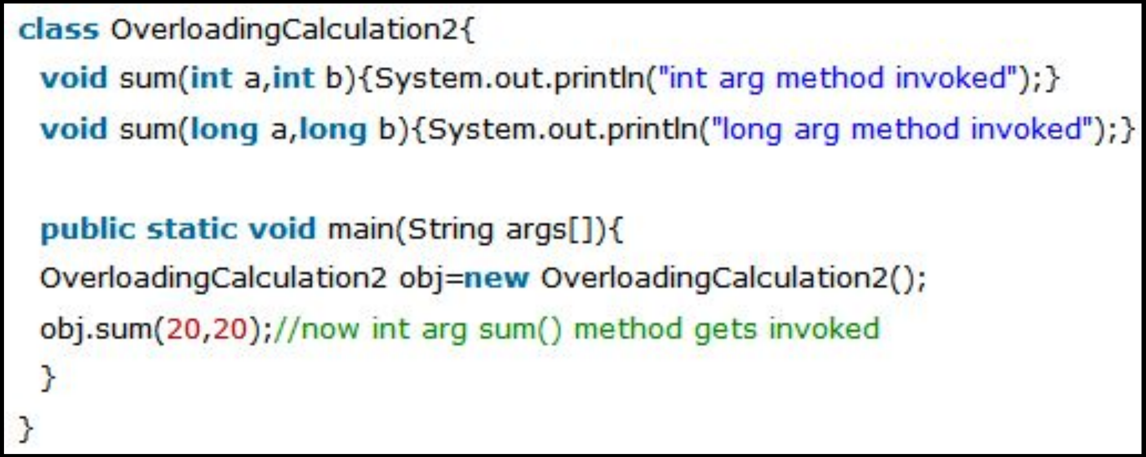
Method Overloading and Type Promotion:...



```
class OverloadingCalculation1{
  void sum(int a,long b){System.out.println(a+b);}
  void sum(int a,int b,int c){System.out.println(a+b+c);}

  public static void main(String args[]){
  OverloadingCalculation1 obj=new OverloadingCalculation1();
  obj.sum(20,20);//now second int literal will be promoted to long
  obj.sum(20,20,20);

  }
}
```



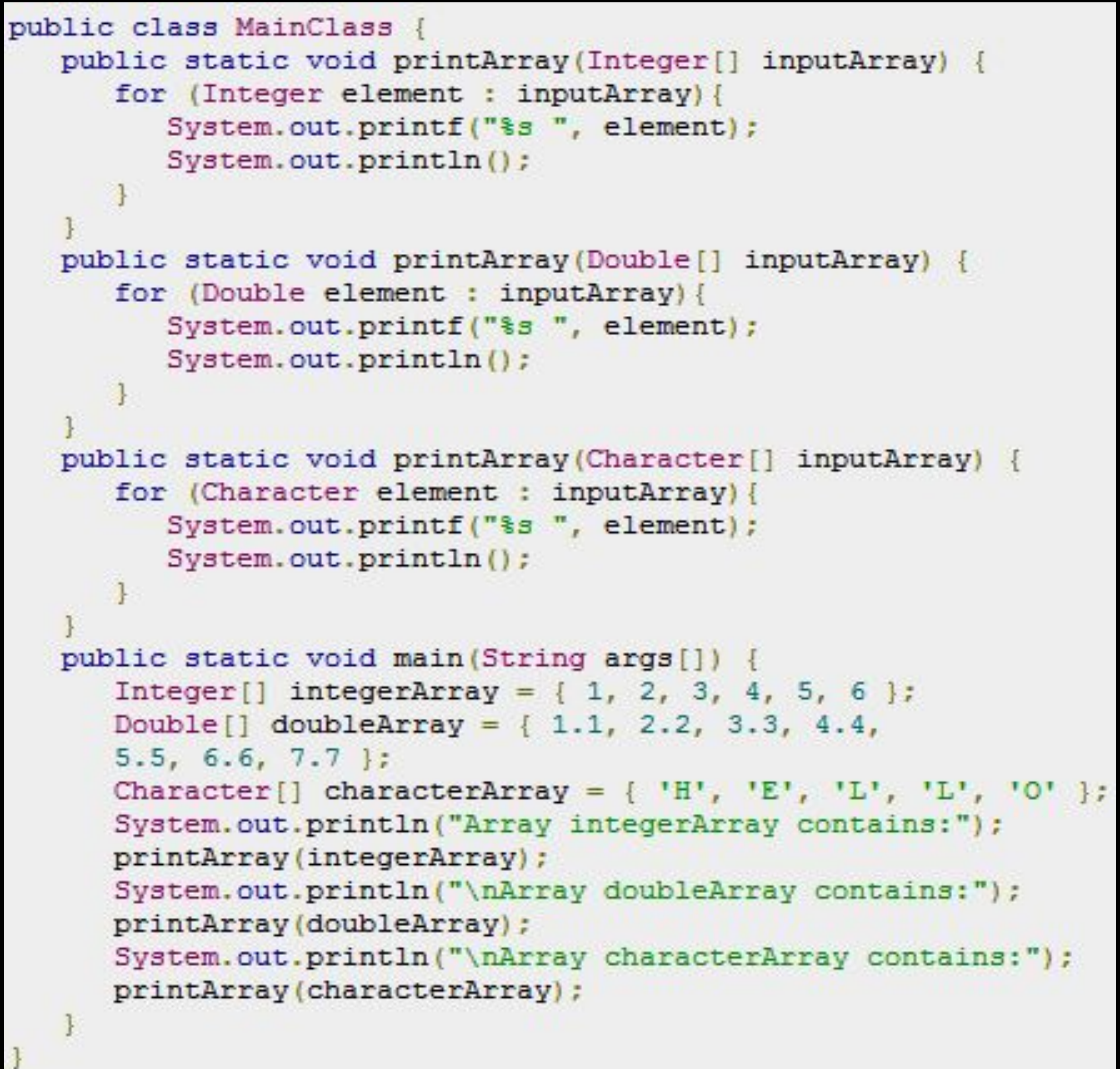
```
class OverloadingCalculation3{
  void sum(int a,long b){System.out.println("a method invoked");}
  void sum(long a,int b){System.out.println("b method invoked");}

  public static void main(String args[]){
  OverloadingCalculation3 obj=new OverloadingCalculation3();
  obj.sum(20,20);//now ambiguity
  }
}
```



```
class OverloadingCalculation2{
  void sum(int a,int b){System.out.println("int arg method invoked");}
  void sum(long a,long b){System.out.println("long arg method invoked");}

  public static void main(String args[]){
  OverloadingCalculation2 obj=new OverloadingCalculation2();
  obj.sum(20,20);//now int arg sum() method gets invoked
  }
}
```

Introducing Classes

Introducing Methods: ... *Overloading* ...

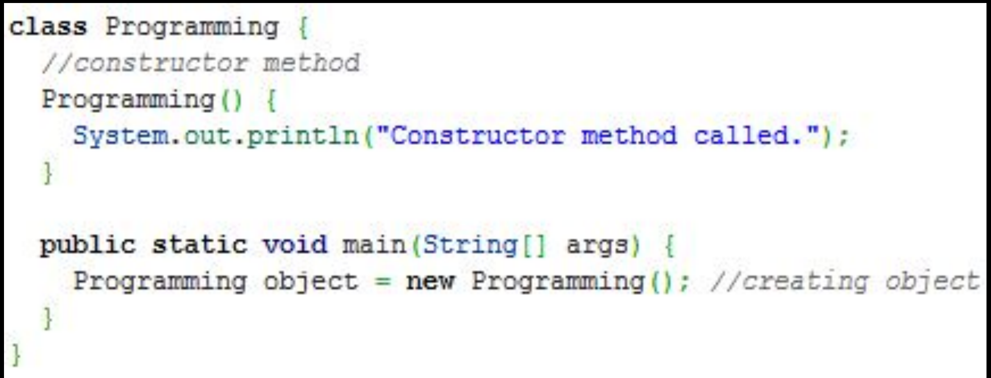


```
public class MainClass {
   public static void printArray(Integer[] inputArray) {
      for (Integer element : inputArray){
         System.out.printf("%s ", element);
         System.out.println();
      }
   }
   public static void printArray(Double[] inputArray) {
      for (Double element : inputArray){
         System.out.printf("%s ", element);
         System.out.println();
      }
   }
   public static void printArray(Character[] inputArray) {
      for (Character element : inputArray){
         System.out.printf("%s ", element);
         System.out.println();
      }
   }
   public static void main(String args[]) {
      Integer[] integerArray = { 1, 2, 3, 4, 5, 6 };
      Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4,
      5.5, 6.6, 7.7 };
      Character[] characterArray = { 'H', 'E', 'L', 'L', 'O' };
      System.out.println("Array integerArray contains:");
      printArray(integerArray);
      System.out.println("\nArray doubleArray contains:");
      printArray(doubleArray);
      System.out.println("\nArray characterArray contains:");
      printArray(characterArray);
   }
}
```

Introducing Classes

Constructors:

- It can be tedious to initialize all of the variables in a class each time an instance is created. 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, *before the **new** operator completes*. Constructors look a little strange because they have no return type, not even **void.**



```
class Programming {
  //constructor method
  Programming() {
    System.out.println("Constructor method called.");
  }

  public static void main(String[] args) {
    Programming object = new Programming(); //creating object
  }
}
```

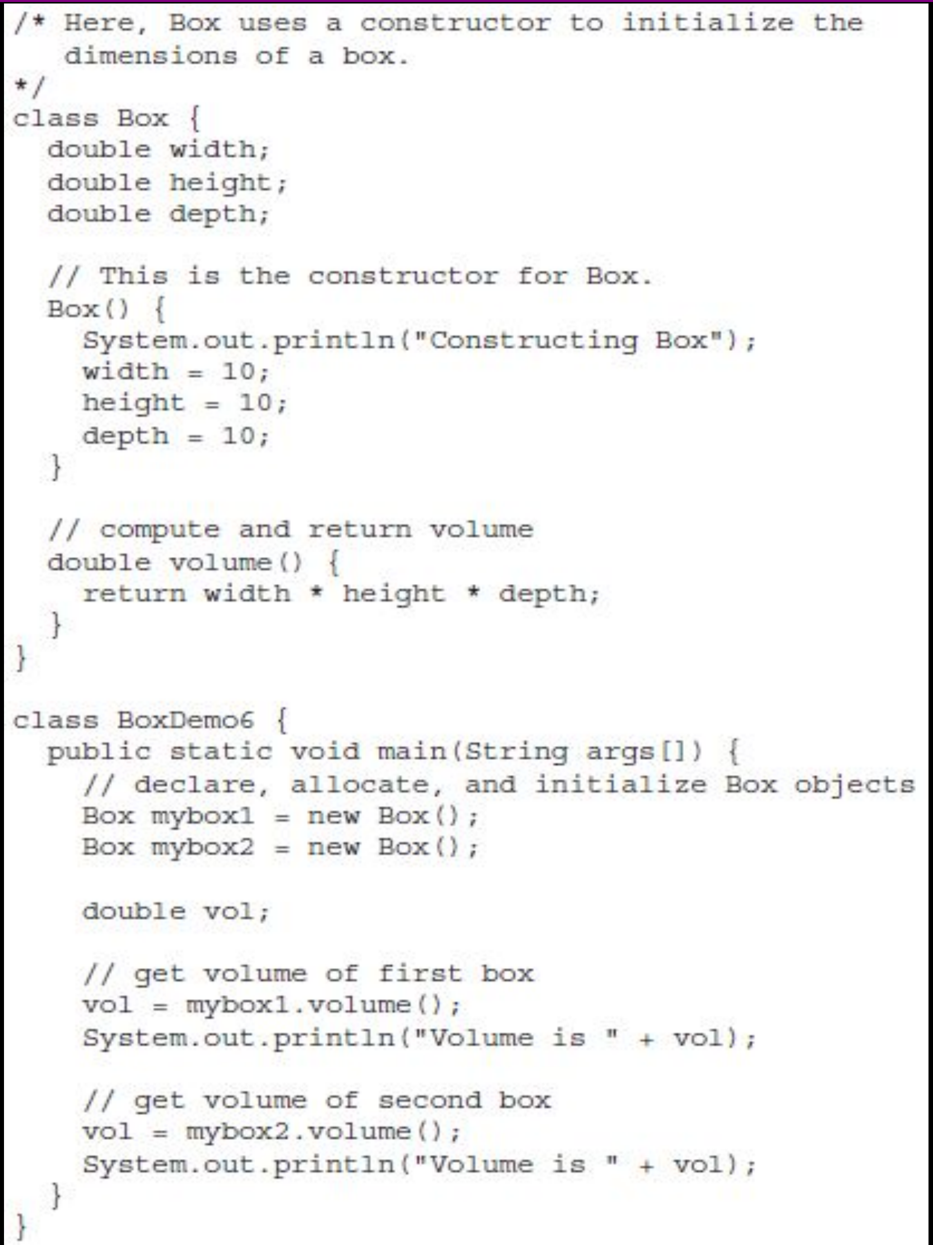
Introducing Classes

Constructors: ...

class-var = new classname( );

Box mybox1 = new Box();

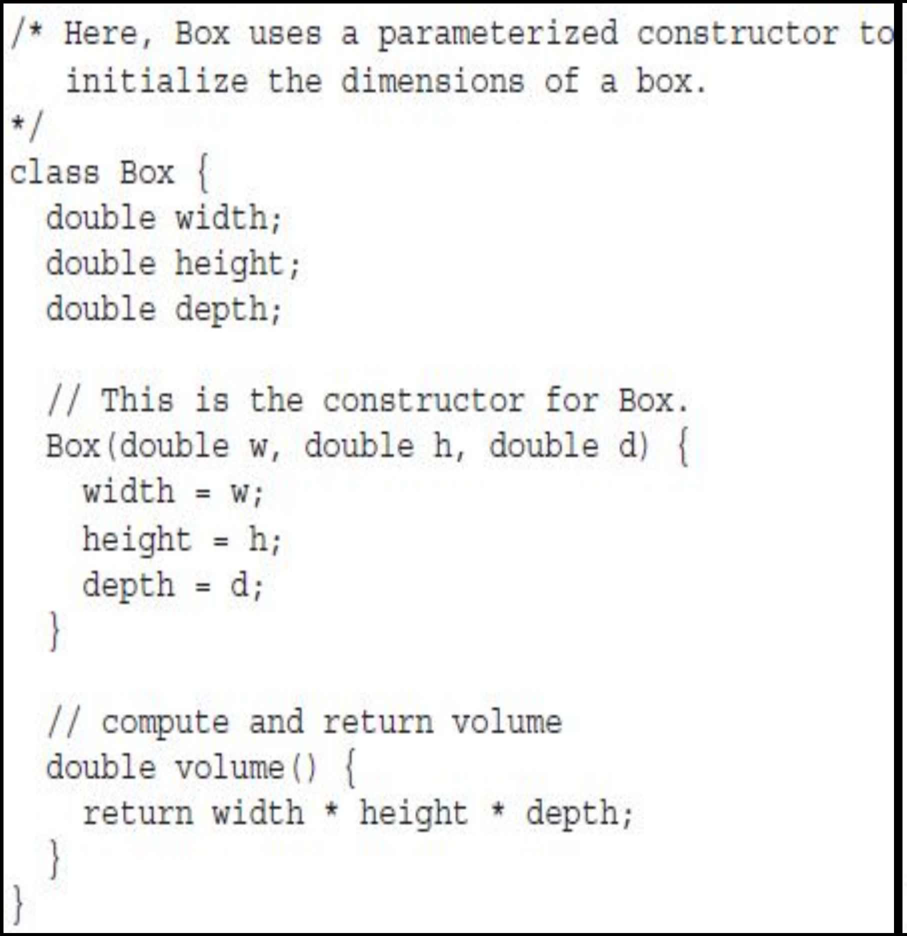
- When you do not explicitly define a constructor for a class, then Java creates a default constructor for the class.
- The default constructor automatically initializes all instance variables to zero.
- Once you define your own constructor, the default constructor is no longer used.



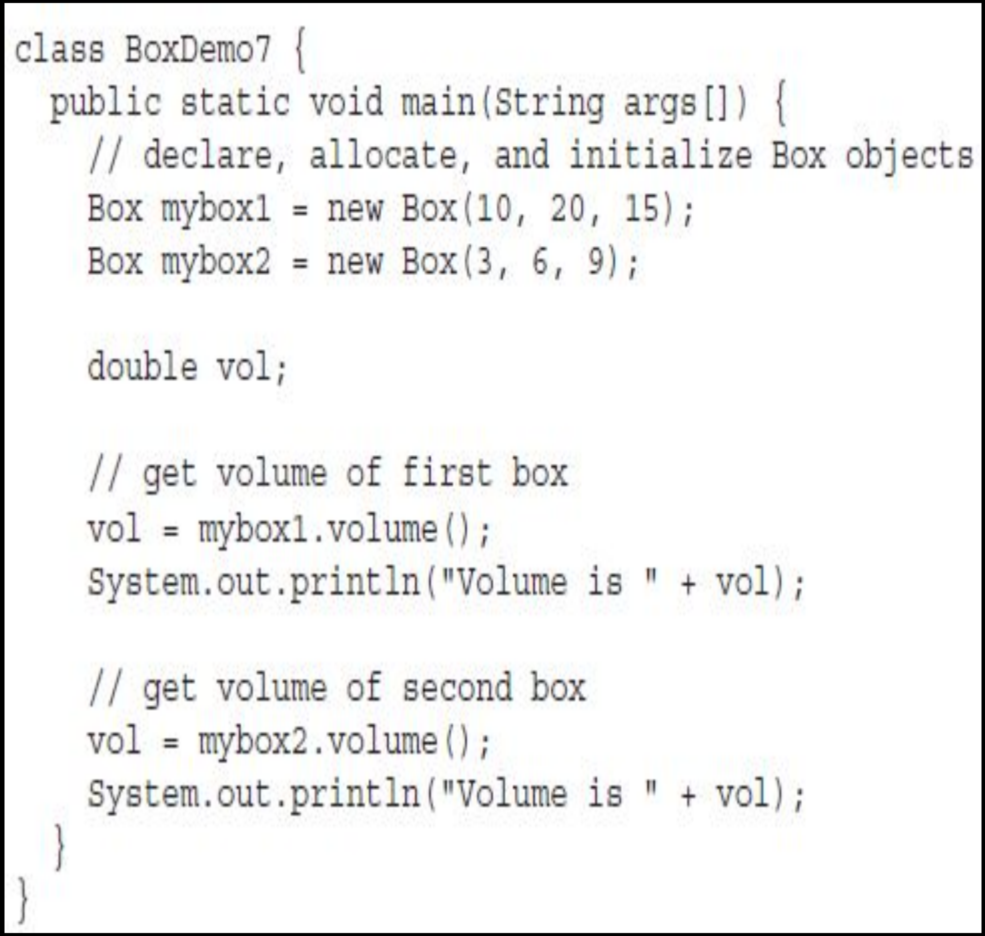
```
/* Here, Box uses a constructor to initialize the
   dimensions of a box.
*/
class Box {
  double width;
  double height;
  double depth;

  // 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();
    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);
  }
}
```

Introducing Classes

Constructors: ... *Parameterized Constructors*



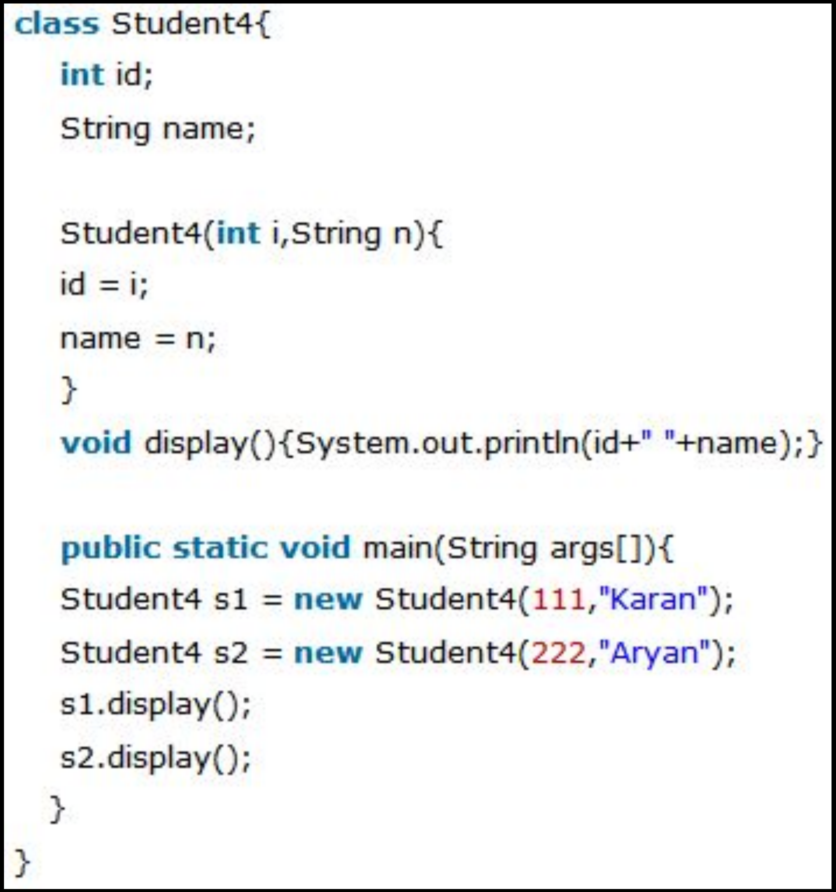
```
/* Here, Box uses a parameterized constructor to
   initialize the dimensions of a box.
*/
class Box {
  double width;
  double height;
  double depth;

  // 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;
  }
}
```



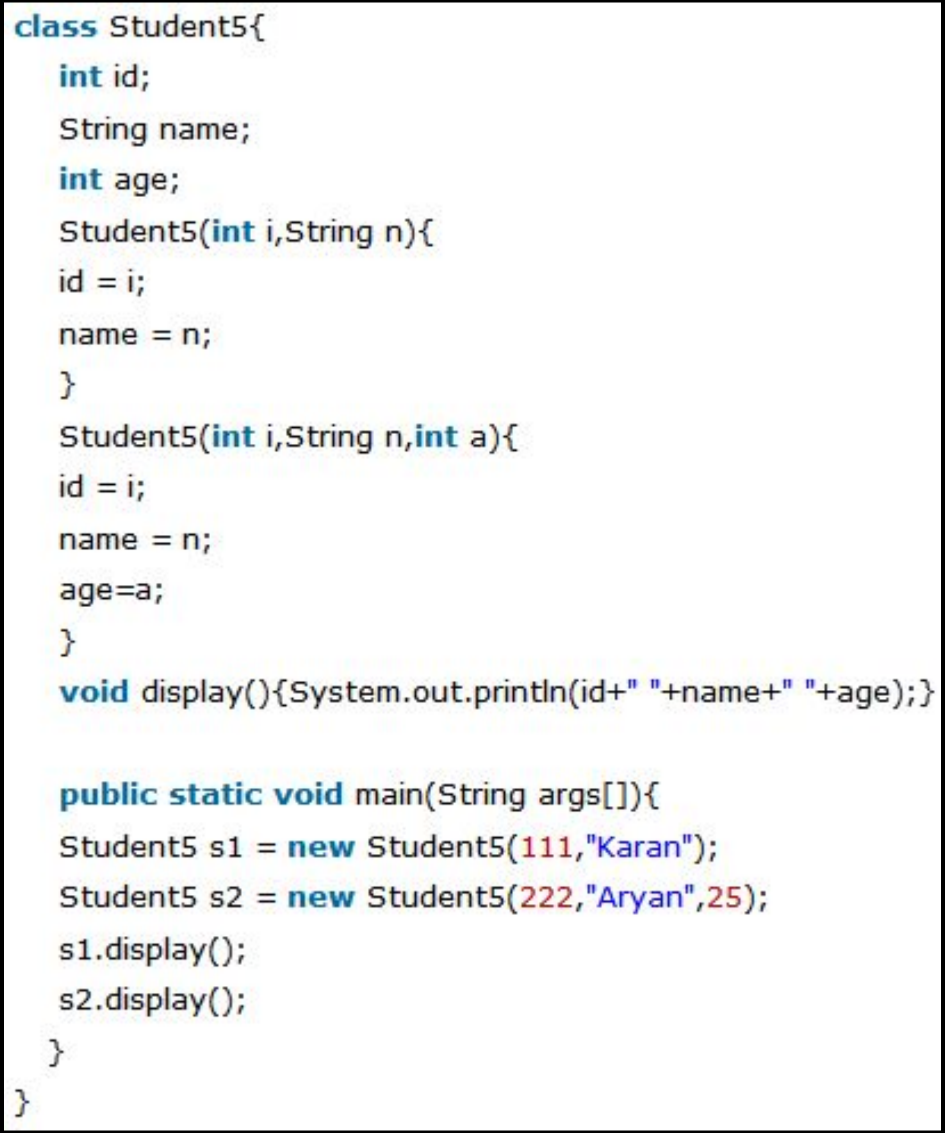
```
class BoxDemo7 {
  public static void main(String args[]) {
    // declare, allocate, and initialize Box objects
    Box mybox1 = new Box(10, 20, 15);
    Box mybox2 = new Box(3, 6, 9);

    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);
  }
}
```

Introducing Classes

Constructors: ... *Overloaded Constructors*



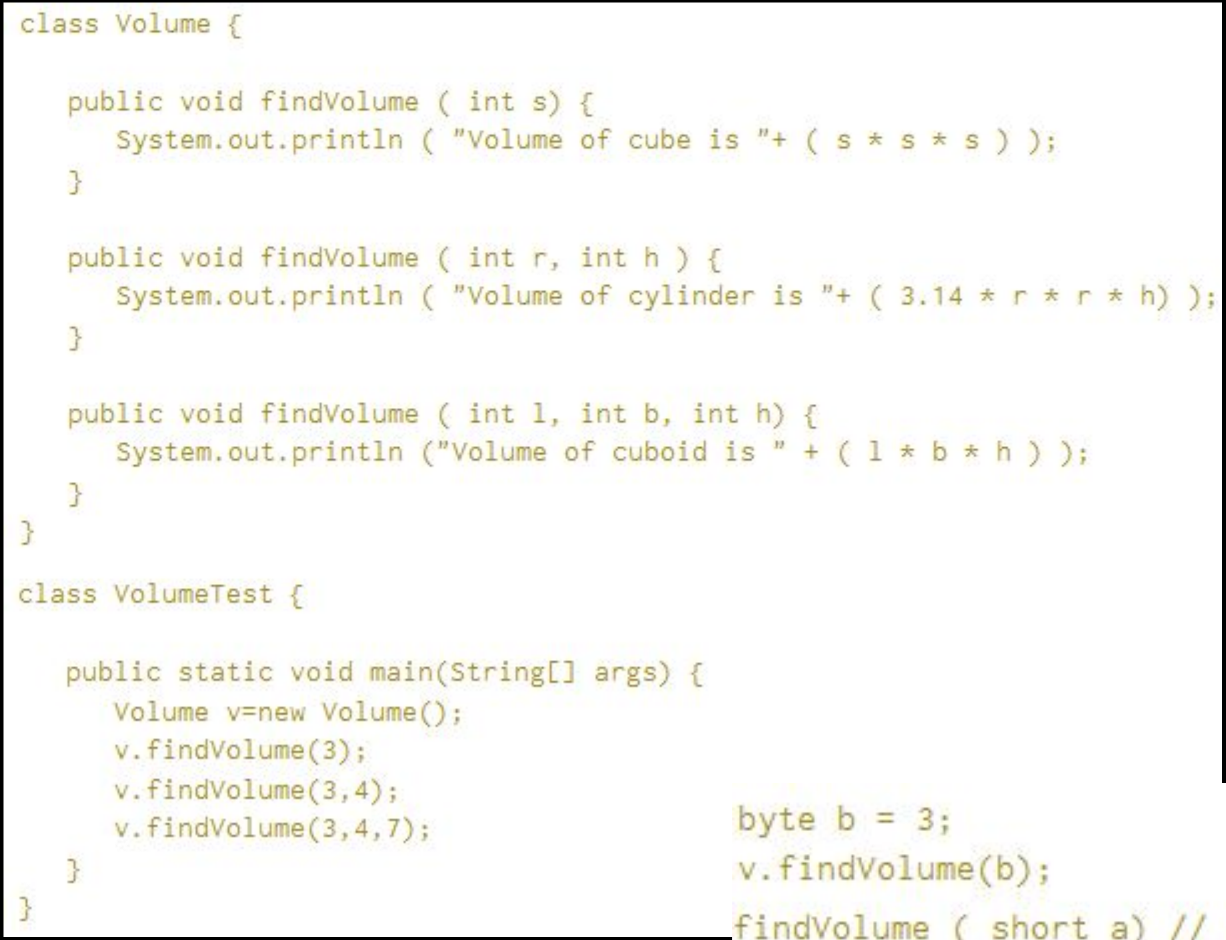
```
class Student4{
    int id;
    String name;

    Student4(int i,String n){
    id = i;
    name = n;
    }
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
    Student4 s1 = new Student4(111,"Karan");
    Student4 s2 = new Student4(222,"Aryan");
    s1.display();
    s2.display();
   }
}
```



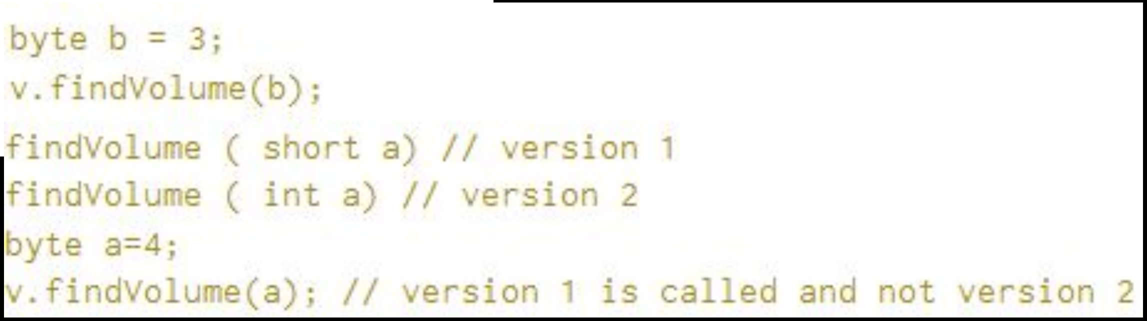
```
class Student5{
    int id;
    String name;
    int age;
    Student5(int i,String n){
    id = i;
    name = n;
    }
    Student5(int i,String n,int a){
    id = i;
    name = n;
    age=a;
    }
    void display(){System.out.println(id+" "+name+" "+age);}

    public static void main(String args[]){
    Student5 s1 = new Student5(111,"Karan");
    Student5 s2 = new Student5(222,"Aryan",25);
    s1.display();
    s2.display();
   }
}
```

Introducing Classes

Constructors: ... *Overloaded Methods...*



```
class Volume {

   public void findVolume ( int s) {
      System.out.println ( "Volume of cube is "+ ( s * s * s ) );
   }

   public void findVolume ( int r, int h ) {
      System.out.println ( "Volume of cylinder is "+ ( 3.14 * r * r * h) );
   }

   public void findVolume ( int l, int b, int h) {
      System.out.println ("Volume of cuboid is " + ( l * b * h ) );
   }
}

class VolumeTest {

   public static void main(String[] args) {
      Volume v=new Volume();
      v.findVolume(3);
      v.findVolume(3,4);
      v.findVolume(3,4,7);
   }
}
```

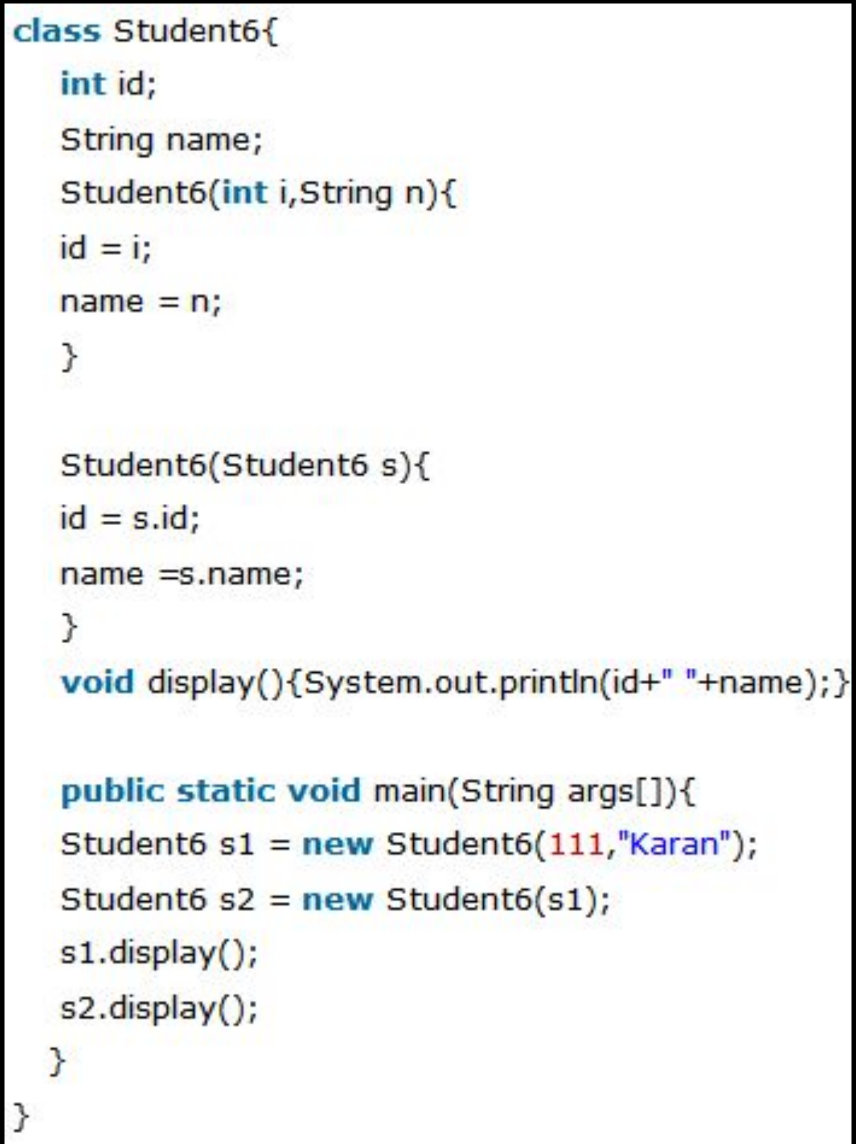


```
byte b = 3;
v.findVolume(b);
findVolume ( short a) // version 1
findVolume ( int a) // version 2
byte a=4;
v.findVolume(a); // version 1 is called and not version 2
```

Introducing Classes

Constructors: ... *Overloaded Constructors...*

Java's Copy Constructor



```
class Student6{
    int id;
    String name;
    Student6(int i,String n){
    id = i;
    name = n;
    }

    Student6(Student6 s){
    id = s.id;
    name =s.name;
    }
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
    Student6 s1 = new Student6(111,"Karan");
    Student6 s2 = new Student6(s1);
    s1.display();
    s2.display();
   }
}
```

Introducing Classes

this keyword:

- Sometimes a method will need to refer to the object that invoked it.
- **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.

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

Instance Variable Hiding

```
// Use this to resolve name-space collisions.
Box(double width, double height, double depth) {
        this.width = width;
        this.height = height;
        this.depth = depth;
}
```

Introducing Classes

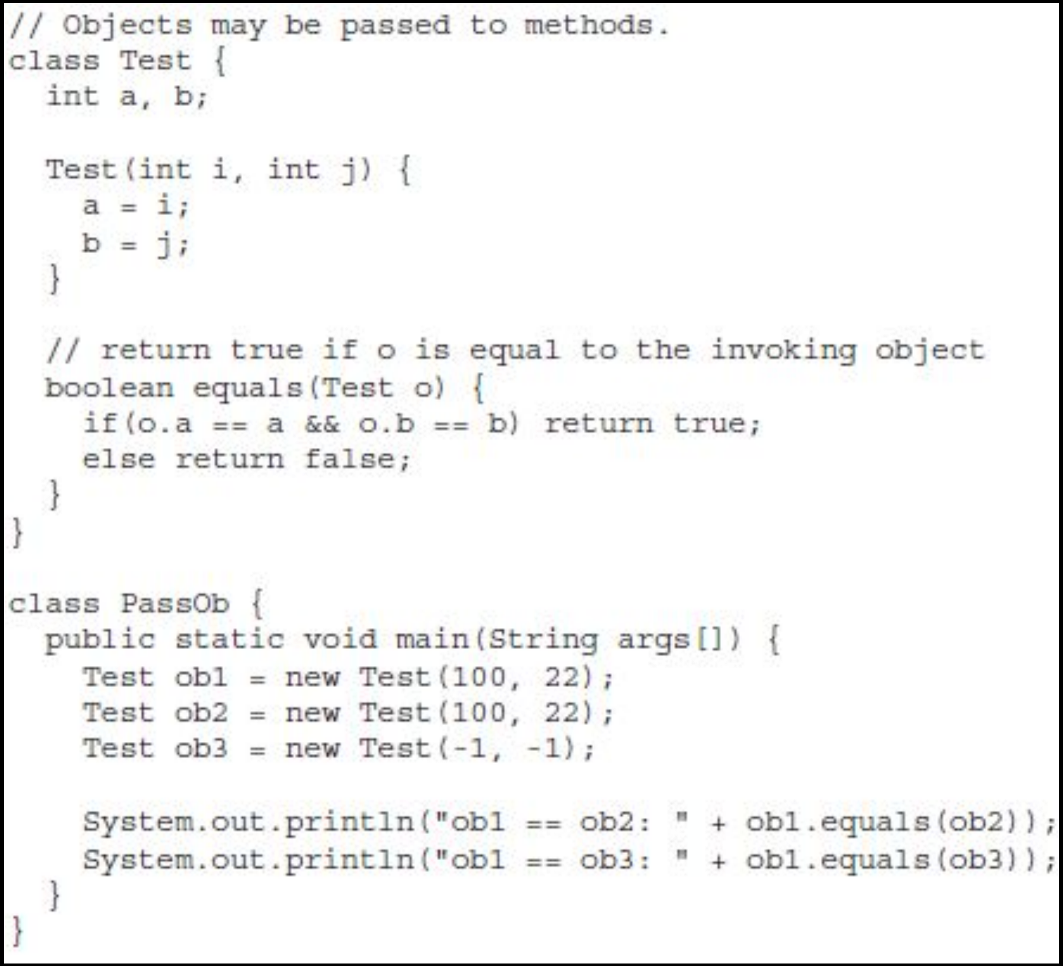
this keyword: ...

A word of caution: The use of **this** in such a context can sometimes be confusing, and some programmers are careful not to use local variables and formal parameter names that hide instance variables. ***Of course, other programmers believe the contrary—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.

Introducing Classes

Use of objects as parameter & Methods returning objects

- 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.



```
// 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 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));
  }
}
```

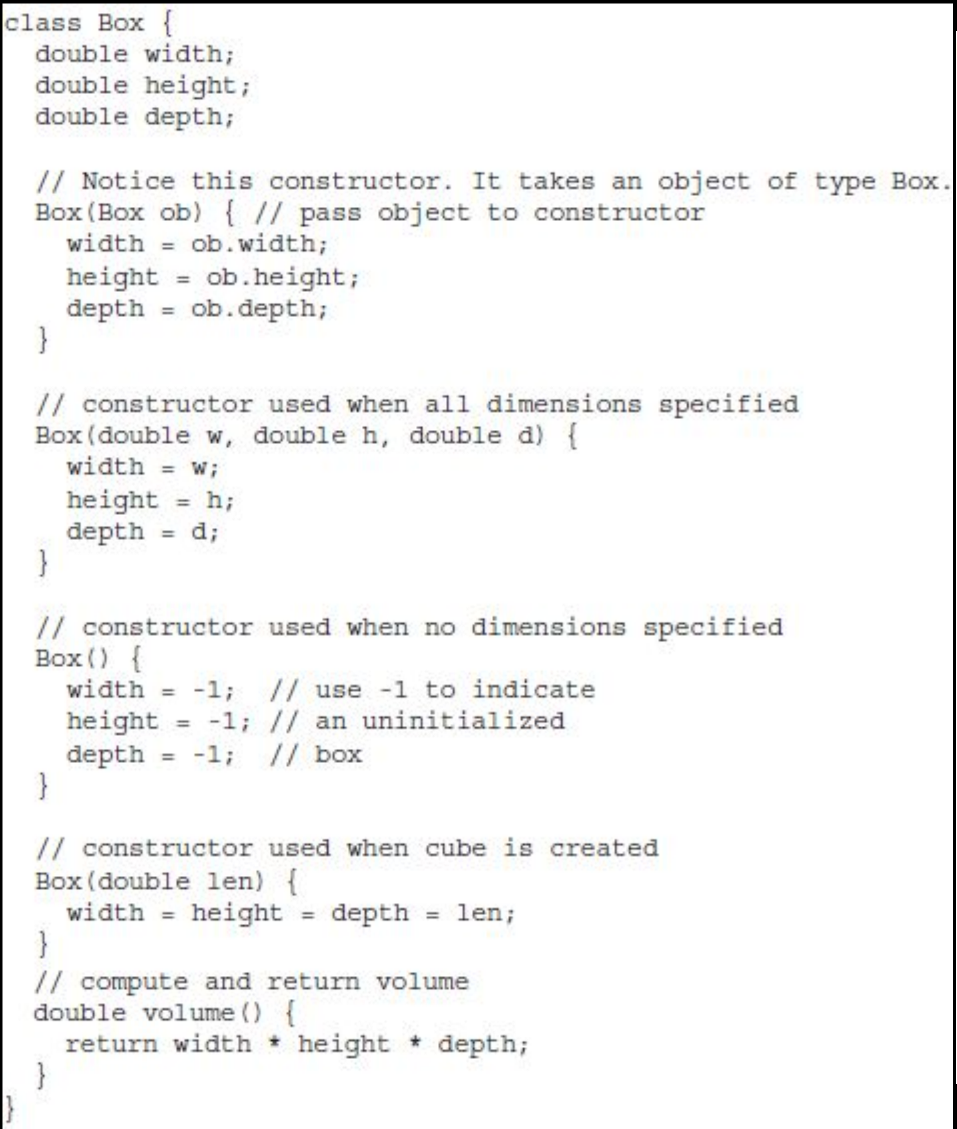
Introducing Classes

Use of objects as parameter & Methods returning objects

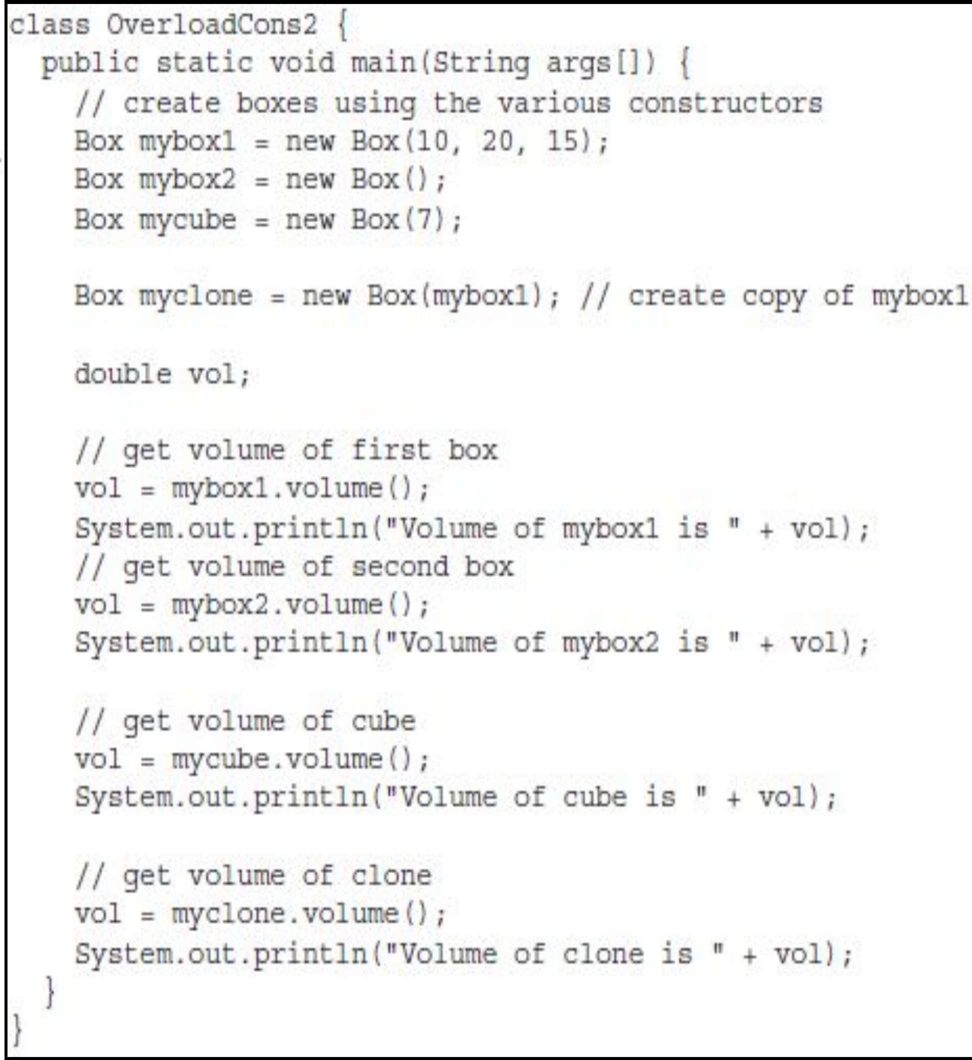
- One of the most common uses of object parameters involves constructors.
- Frequently, you will want to construct a new object so that it is initially the same as some existing object.
- ***Example….***

Introducing Classes

Use of objects as parameter & Methods returning objects



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

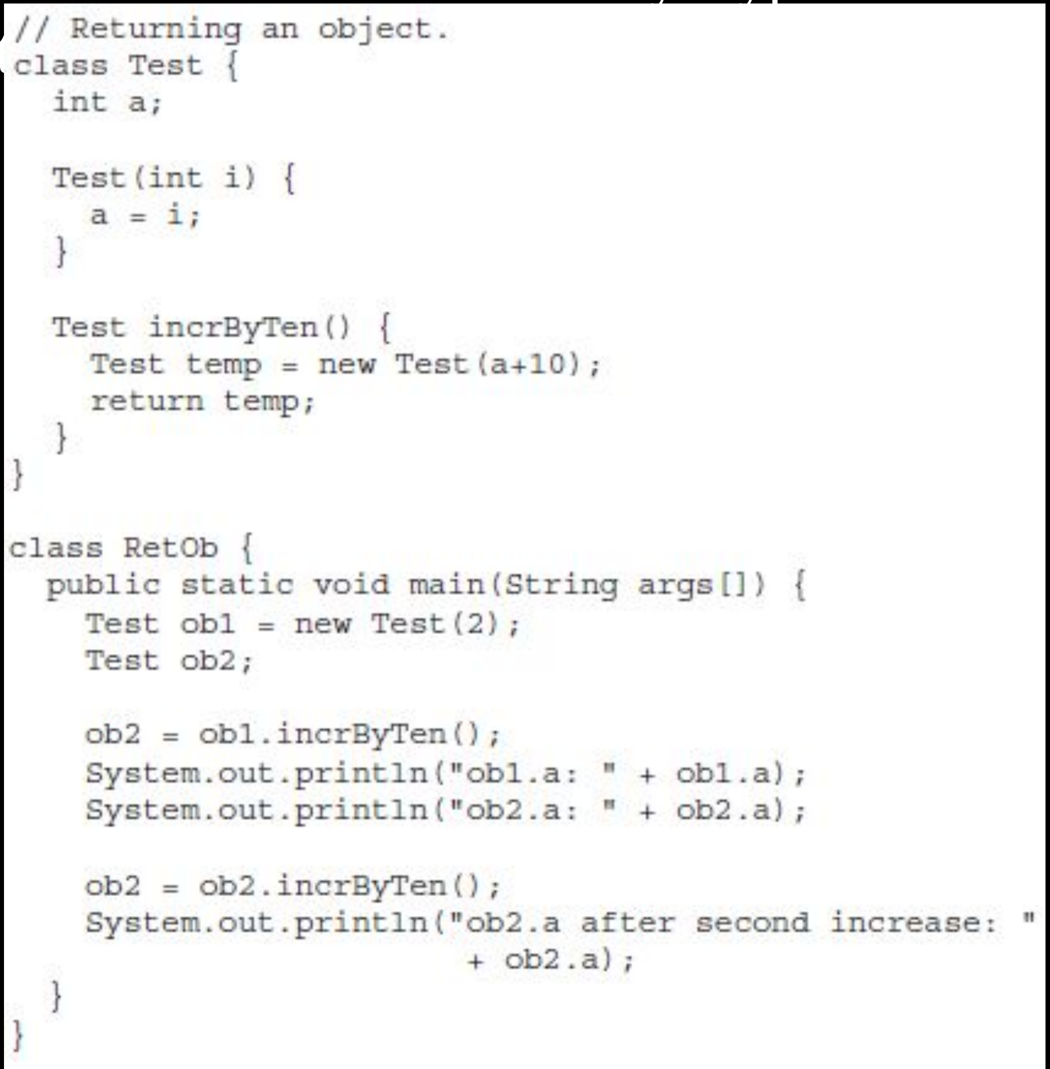


```
class OverloadCons2 {
  public static void main(String args[]) {
    // create boxes using the various constructors
    Box mybox1 = new Box(10, 20, 15);
    Box mybox2 = new Box();
    Box mycube = new Box(7);

    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);
  }
}
```

Introducing Classes

Use of objects as parameter & Methods returning objects

- **Returning Objects:** A method can return any type of data, including class types that



```
// Returning an object.
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);
  }
}
```

Introducing Classes

Call by value & Call by reference:

A Closer Look at Argument Passing

- In general, there are two ways that a computer language can pass an argument to a subroutine: *call-by-value and call-by-reference.*
- *call-by-value: 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.
- *call-by-reference:* a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, 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.

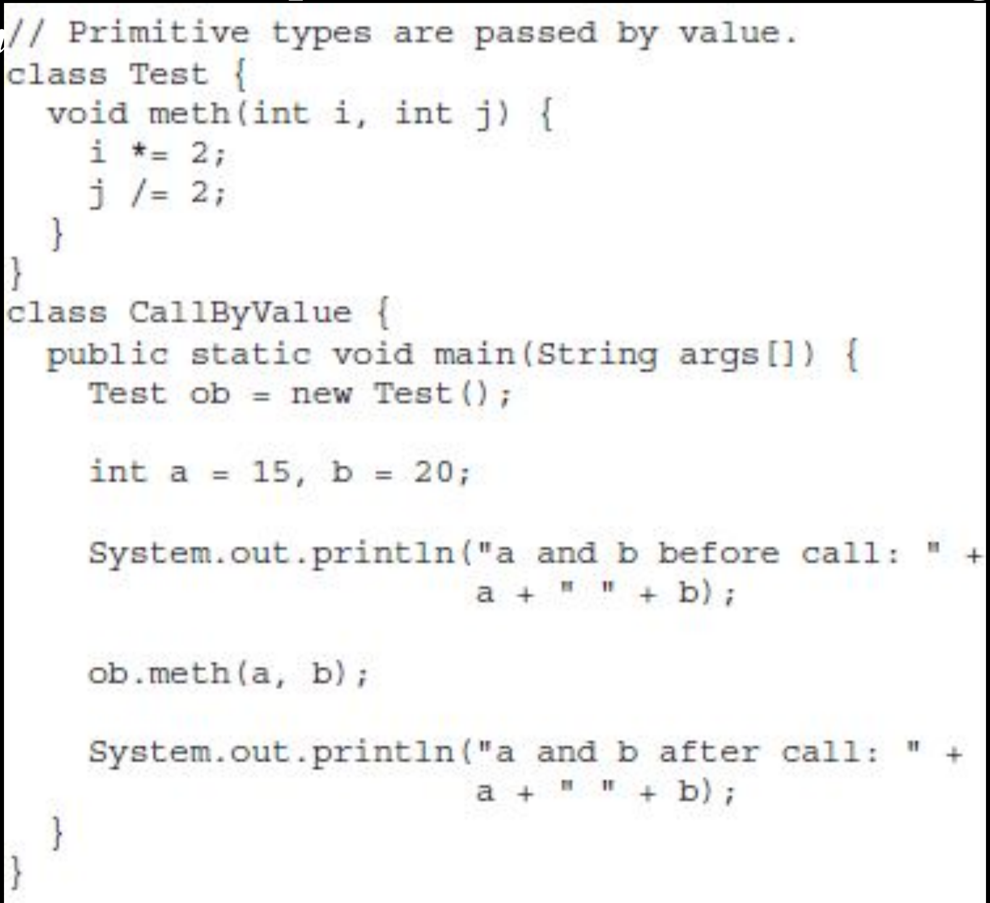
REMEMBER *When a primitive type is passed to a method, it is done by use of call-by-value. Objects are implicitly passed by use of call-by-reference.*

Introducing Classes

Call by value & Call by reference: ...

A Closer Look at Argument Passing ...

- *In Java, when you pass a primitive type to a method, it is passed by value. Thus, what occurs to the parameter that receives the argument has no effect outside the m*



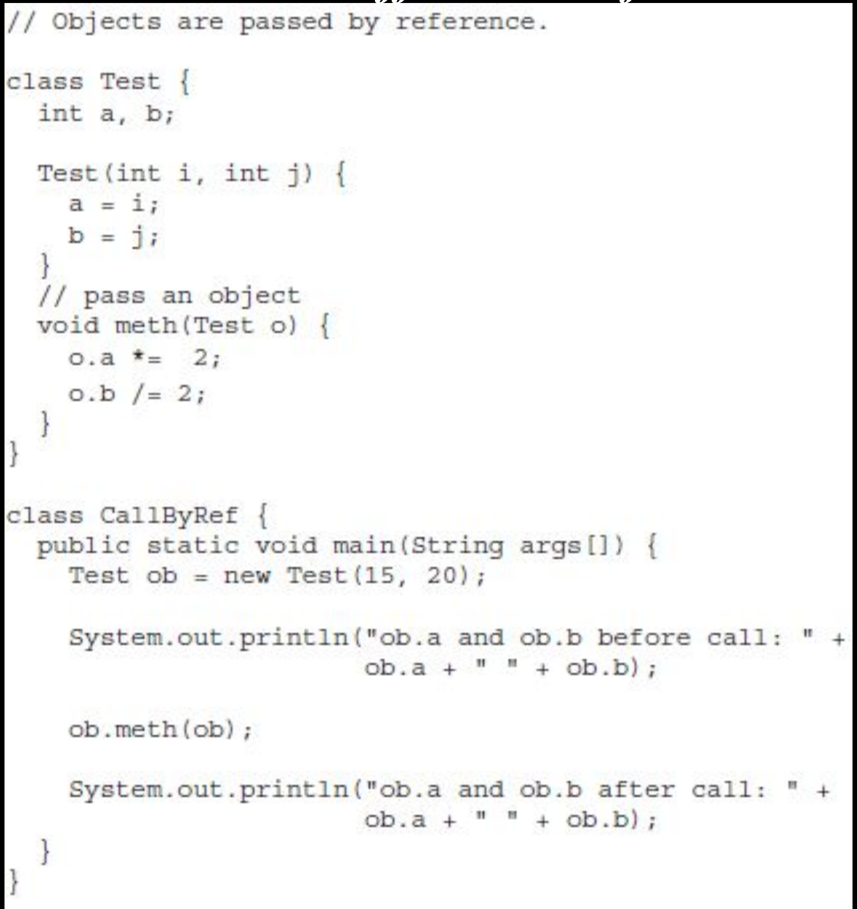
```
// Primitive types are passed by value.
class Test {
  void meth(int i, int j) {
    i *= 2;
    j /= 2;
  }
}
class CallByValue {
  public static void main(String args[]) {
    Test ob = new Test();

    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);
  }
}
```

Introducing Classes

Call by value & Call by reference: ...

A Closer Look at Argument Passing ...

- *The objects are passed to methods by use of* ***call-by-reference****. Changes to the object inside the method do affect the object used as an argument.*



```
// Objects are passed 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);
  }
}
```

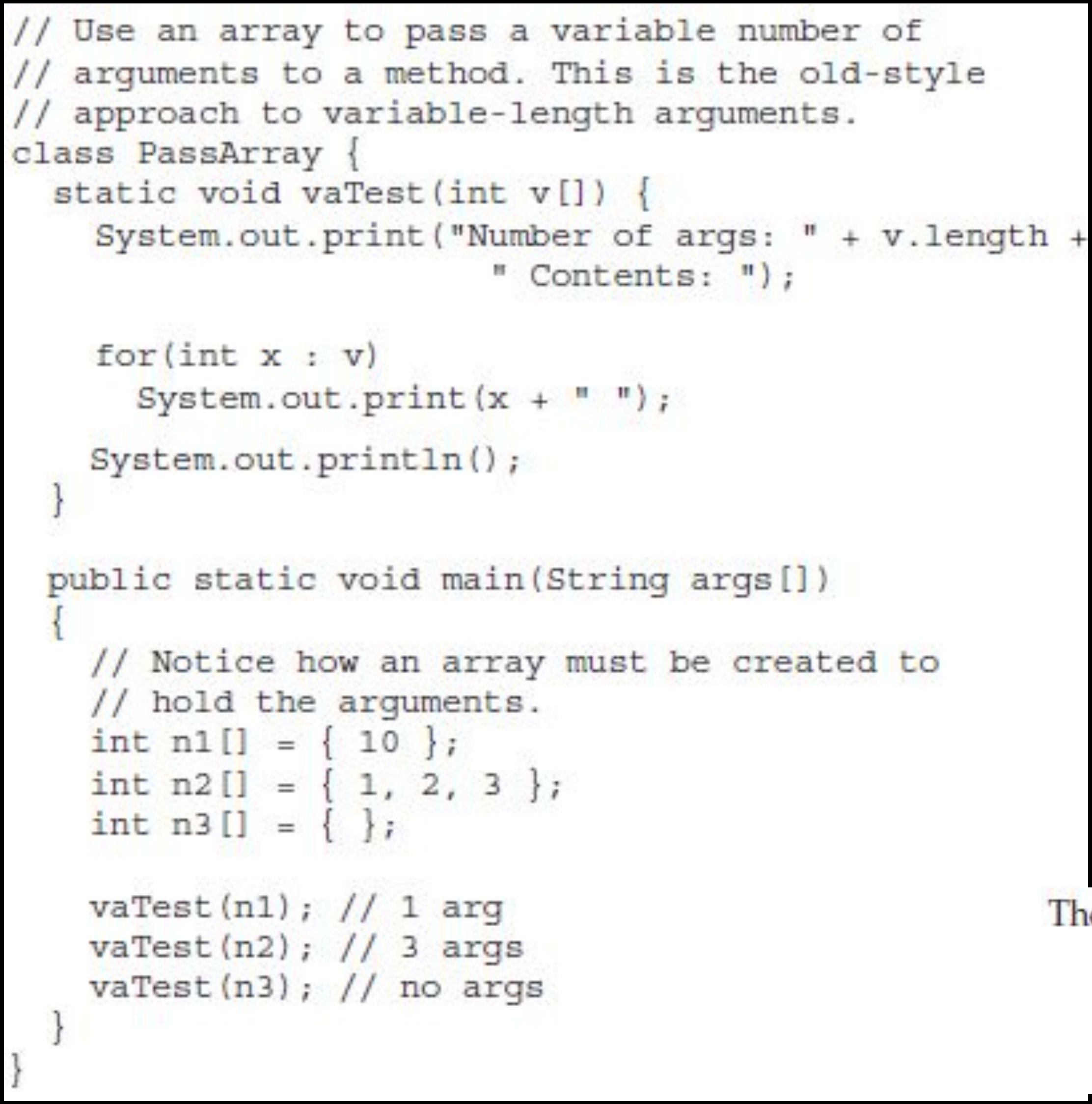
Introducing Classes

Varargs: Variable-Length Arguments:

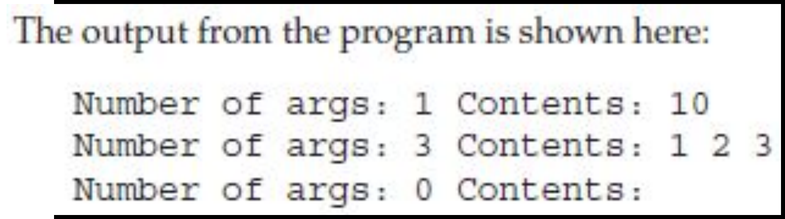
- Beginning with JDK 5, Java has included a feature that simplifies the creation of methods that need to take a variable number of arguments. This feature is called *varargs* and it is short for *variable-length arguments*.
- A method that takes a variable number of arguments is called a *variable-arity method*, or simply a *varargs method*.
- Situations that require that a variable number of arguments be passed to a method are not unusual.
- For example, a method that opens an Internet connection might take a user name, password, filename, protocol, and so on, but supply defaults if some of this information is not provided.

Introducing Classes

Varargs: Variable-Length Arguments:...



```
// Use an array to pass a variable number of
// arguments to a method. This is the old-style
// approach to variable-length arguments.
class PassArray {
  static void vaTest(int v[]) {
    System.out.print("Number of args: " + v.length +
                     " Contents: ");

    for(int x : v)
      System.out.print(x + " ");

    System.out.println();
  }

  public static void main(String args[])
  {
    // Notice how an array must be created to
    // hold the arguments.
    int n1[] = { 10 };
    int n2[] = { 1, 2, 3 };
    int n3[] = { };

    vaTest(n1); // 1 arg
    vaTest(n2); // 3 args
    vaTest(n3); // no args
  }
}
```



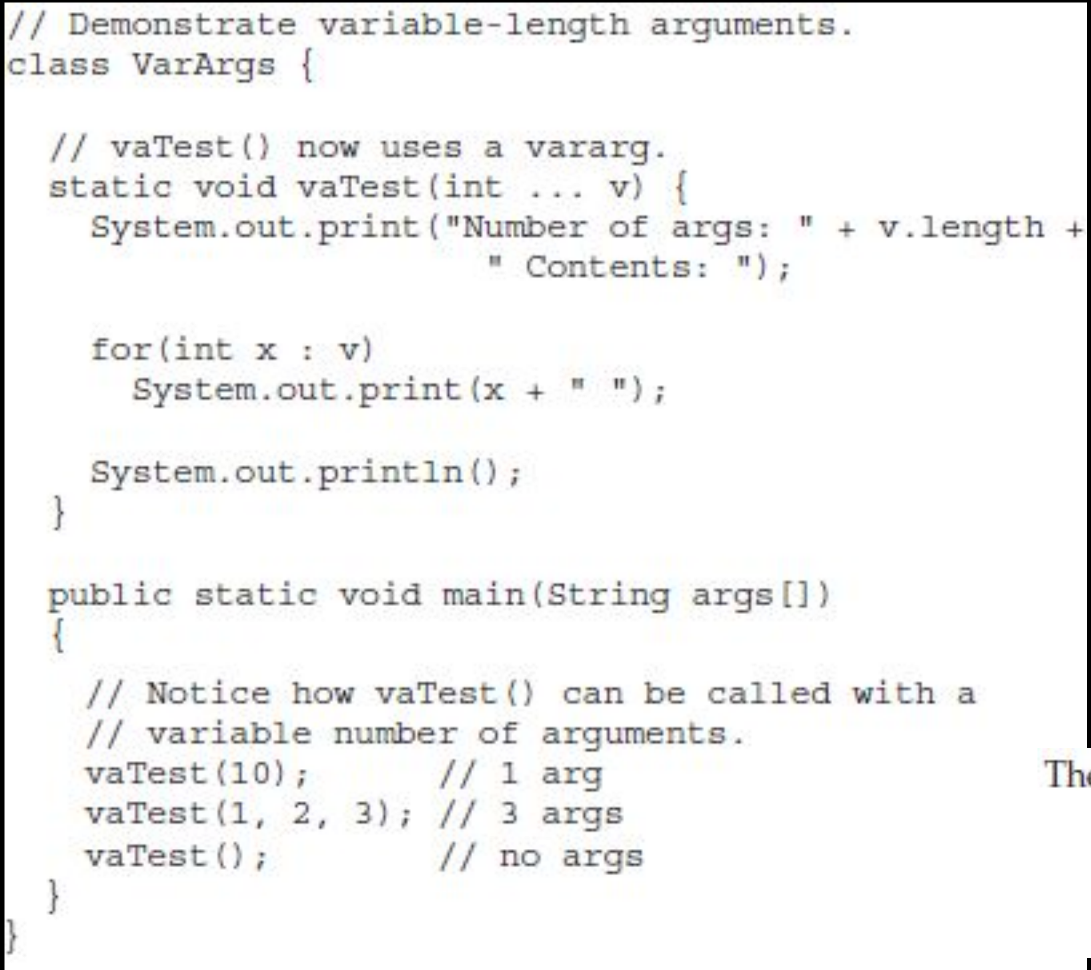
The output from the program is shown here:

```
Number of args: 1 Contents: 10
Number of args: 3 Contents: 1 2 3
Number of args: 0 Contents:
```

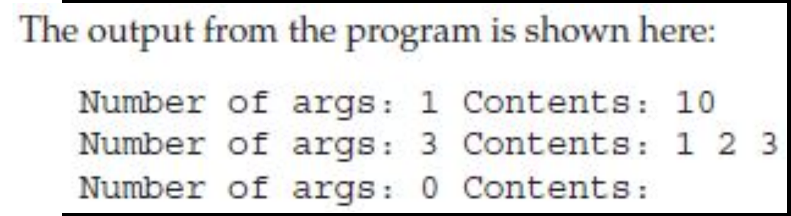
Introducing Classes

Varargs: Variable-Length Arguments:...

- Avariable-length argument is specified by three periods (**...**)
- **Eg:** vaTest( ) is written using a **vararg:** ***static void vaTest(int ... v)***



```
// Demonstrate variable-length arguments.
class VarArgs {

  // vaTest() now uses a vararg.
  static void vaTest(int ... v) {
    System.out.print("Number of args: " + v.length +
                     " Contents: ");

    for(int x : v)
      System.out.print(x + " ");

    System.out.println();
  }

  public static void main(String args[])
  {

    // Notice how vaTest() can be called with a
    // variable number of arguments.
    vaTest(10);      // 1 arg
    vaTest(1, 2, 3); // 3 args
    vaTest();        // no args
  }
}
```



The output from the program is shown here:

```
Number of args: 1 Contents: 10
Number of args: 3 Contents: 1 2 3
Number of args: 0 Contents:
```

Introducing Classes

Varargs: Variable-Length Arguments:...

- A method can have "normal" parameters along with a variable-length parameter. However, the variable-length parameter must be the last parameter declared by the method.
- **Eg:** This method declaration is perfectly acceptable:

  int doIt(int a, int b, double c, int ... vals) {
- Remember, the varargs parameter must be last.
- For example, the following declaration is incorrect:

int doIt(int a, int b, double c, int ... vals, boolean stopFlag) { // Error!

int doIt(int a, int b, double c, int ... vals, double ... morevals) { // Error!

Introducing Classes

Varargs: Variable-Length Arguments:...

```java
// Use varargs with standard arguments.
class VarArgs2 {

  // Here, msg is a normal parameter and v is a
  // varargs parameter.
  static void vaTest(String msg, int ... v) {
    System.out.print(msg + v.length +
                     " Contents: ");

    for(int x : v)
      System.out.print(x + " ");

    System.out.println();
  }
  public static void main(String args[])
  {
    vaTest("One vararg: ", 10);
    vaTest("Three varargs: ", 1, 2, 3);
    vaTest("No varargs: ");
  }
}
```

The output from this program is shown here:

```
One vararg: 1 Contents: 10
Three varargs: 3 Contents: 1 2 3
No varargs: 0 Contents:
```

Introducing Classes

Static Variables & Methods:

Understanding static

- Normally, a class member must be accessed only in conjunction with an object of its class. However, 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 its 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.
- 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.

Introducing Classes

Static Variables & Methods: ...

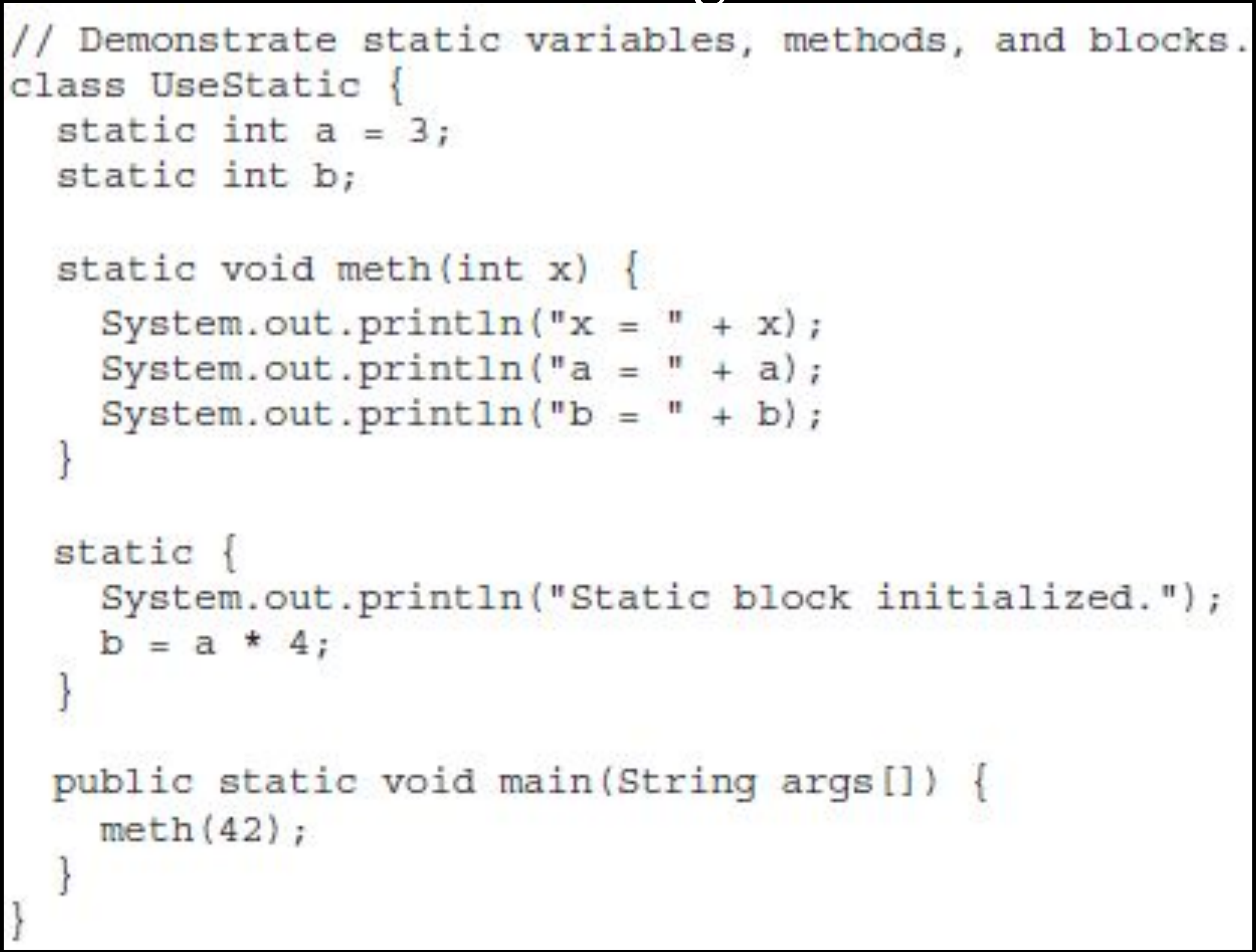
Understanding static...

- 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. (The keyword **super** relates to inheritance)
- If you need to do computation in order to initialize your **static** variables, you can declare a **static** block that gets executed exactly once, when the class is first loaded.

Introducing Classes

Static Variables & Methods: ...

Understanding static...



```
// Demonstrate 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);
  }
}
```

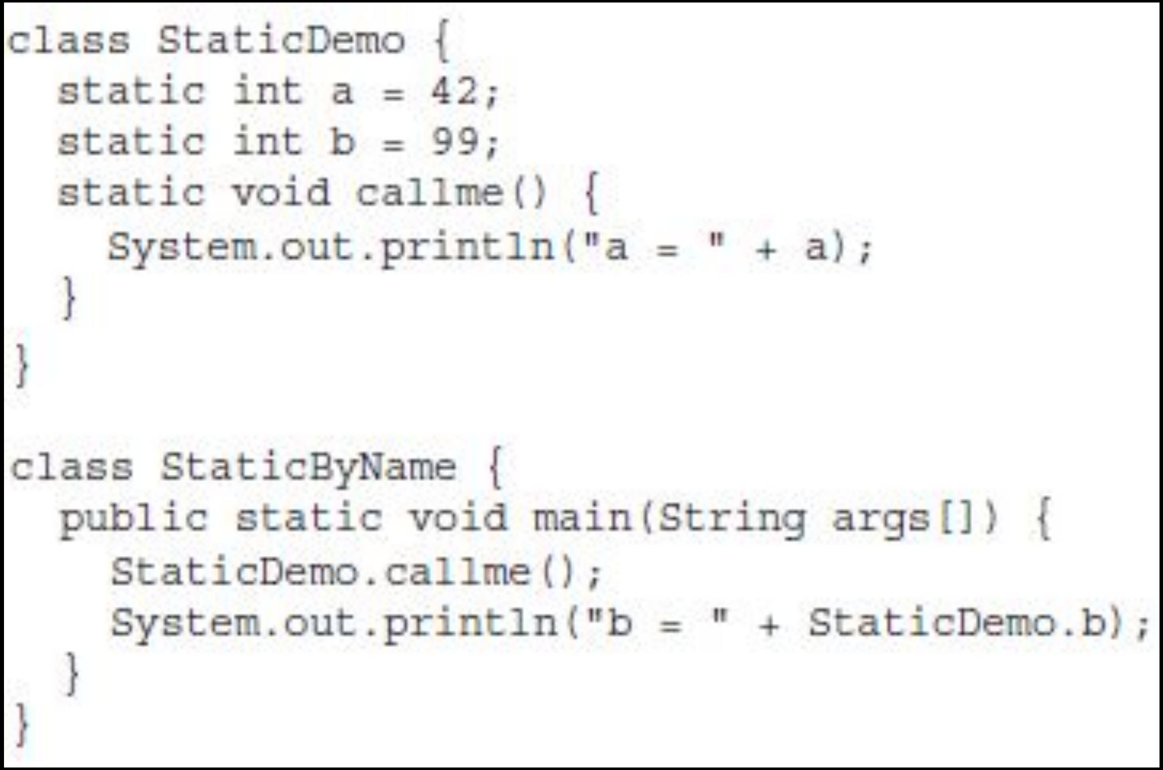
Introducing Classes

Static Variables & Methods: ...

Understanding static...

- Outside of the class in which they are defined, **static** methods and variables can be used independently of any object.

classname.method( )



```
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);
  }
}
```

Introducing Classes

Introducing final:

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

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
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. Thus, a **final** variable is essentially a constant.

Introducing Classes

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. 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.
- 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.

Introducing Classes

Garbage collection: ...

The finalize( ) Method:

- Sometimes an object will need to perform some action when it is destroyed.
- For example, if an object is holding some non-Java resource such as a file handle or character font, then you might want to make sure these resources are freed before an object is destroyed.
- 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.***

Introducing Classes

Garbage collection: ...

The finalize( ) Method:...

- 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
}
```

Introducing Classes

Garbage collection: ...

The finalize( ) Method:...

- 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. This means that you cannot know when—or even if—**finalize( )** will be executed.
- Therefore, your program should provide other means of releasing system resources, etc., used by the object. It must not rely on **finalize( )** for normal program operation.

Introducing Classes

Nested & Inner classes:

- It is possible to define a class within another class; such classes are known as ***nested classes.***
- The scope of a nested class is bounded by the scope of its enclosing class.
- ***Eg:*** If class B is defined within class A, then B does not exist independently of A.
- A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class.
- A nested class that is declared directly within its enclosing class scope is a member of its enclosing class. It is also possible to declare a nested class that is local to a block.

Introducing Classes

Nested & Inner classes: ...

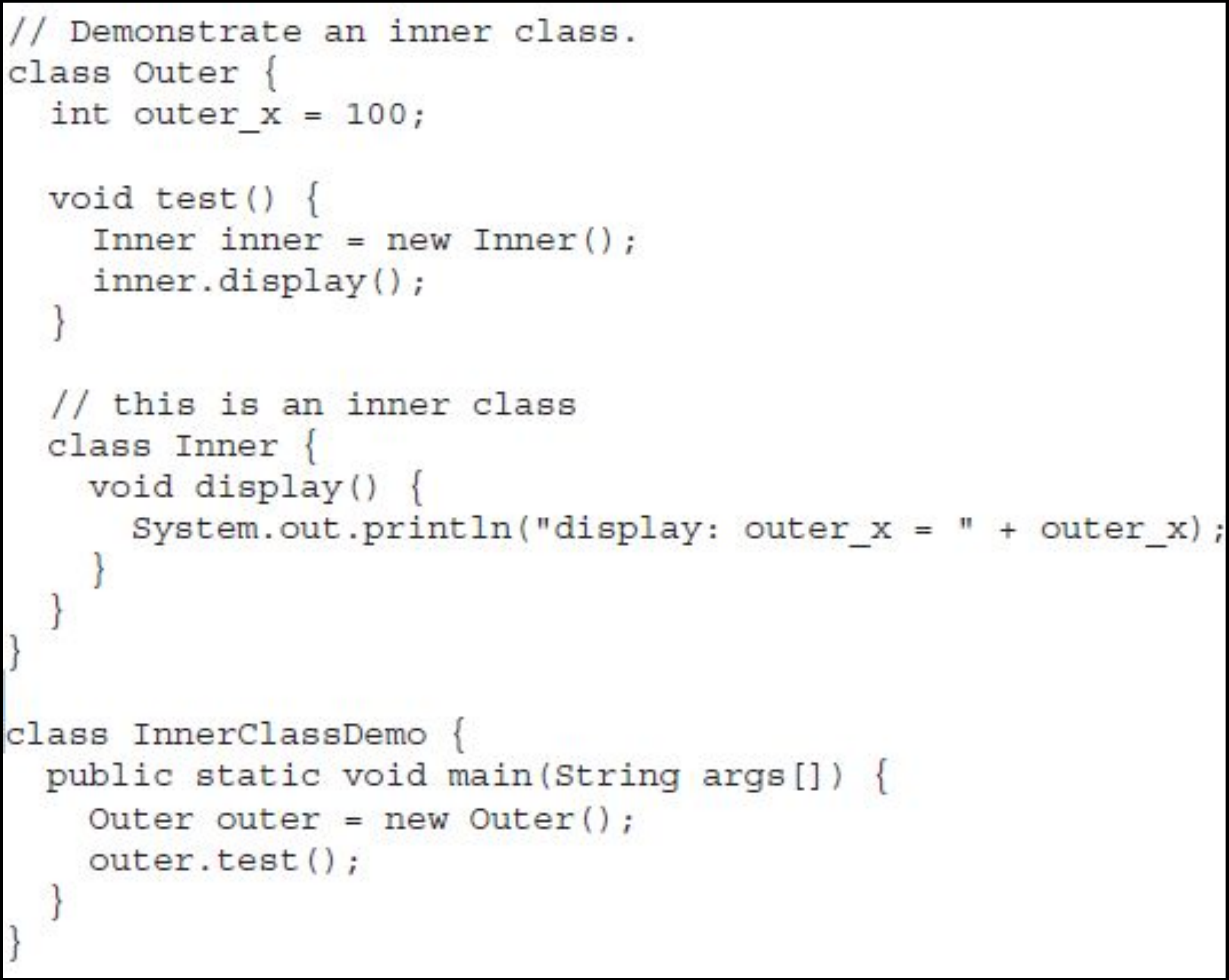
There are two types of nested classes: ***static and non-static***

- ***Static:*** A static nested class is one that has the **static** modifier applied. Because it is static, it must access the members of its enclosing class through an object. That is, it cannot refer to members of its enclosing class directly. Because of this restriction, static nested classes are seldom used.

- ***Non-static:*** The most important type of nested class is the *inner* class. An inner class is a non-static nested class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do.

Introducing Classes

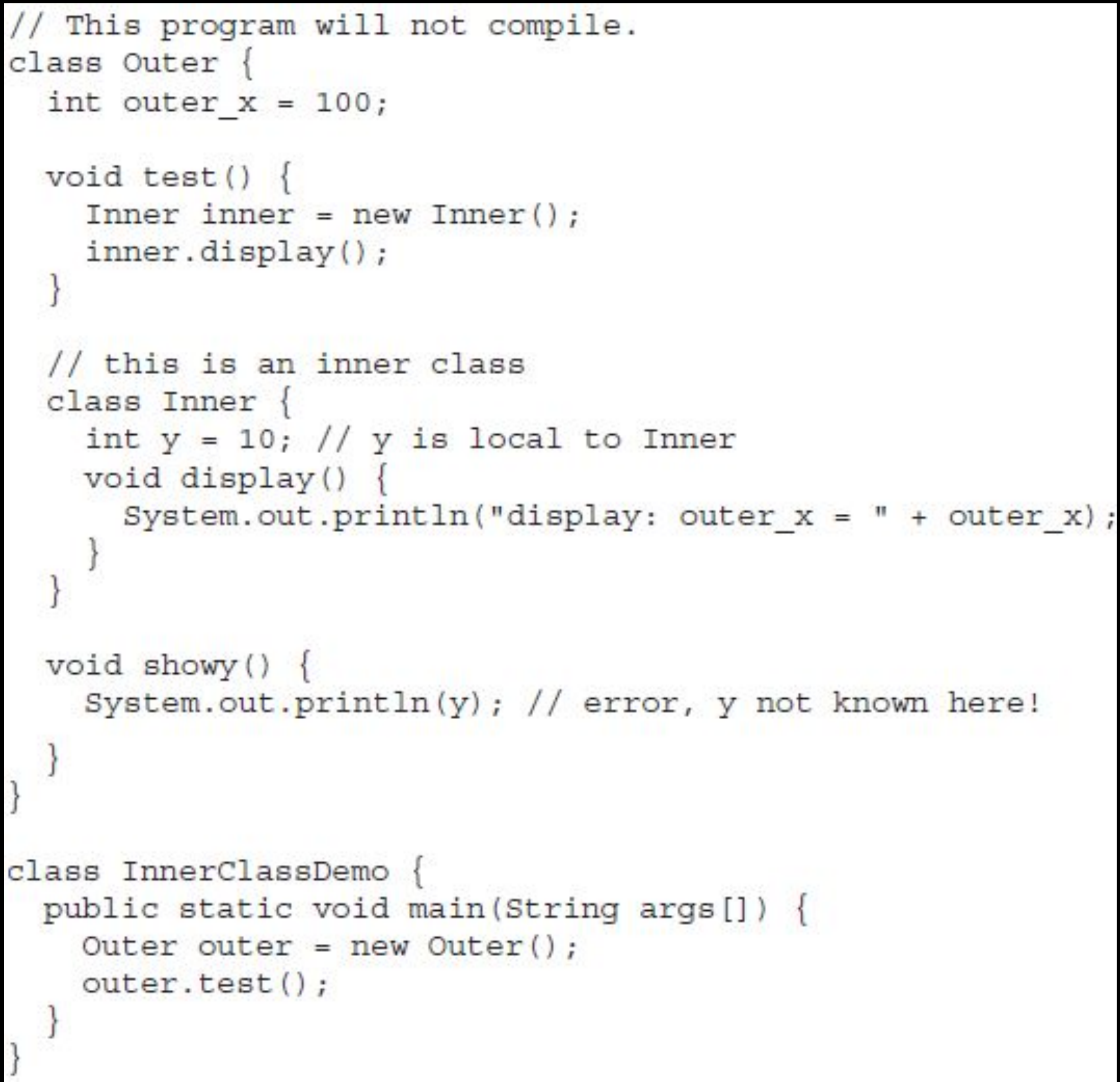
Nested & Inner classes: ... *Inner/Non-Static classes*



```
// Demonstrate an inner class.
class Outer {
  int outer_x = 100;

  void test() {
    Inner inner = new Inner();
    inner.display();
  }

  // this is an inner class
  class Inner {
    void display() {
      System.out.println("display: outer_x = " + outer_x);
    }
  }
}

class InnerClassDemo {
  public static void main(String args[]) {
    Outer outer = new Outer();
    outer.test();
  }
}
```

Introducing Classes

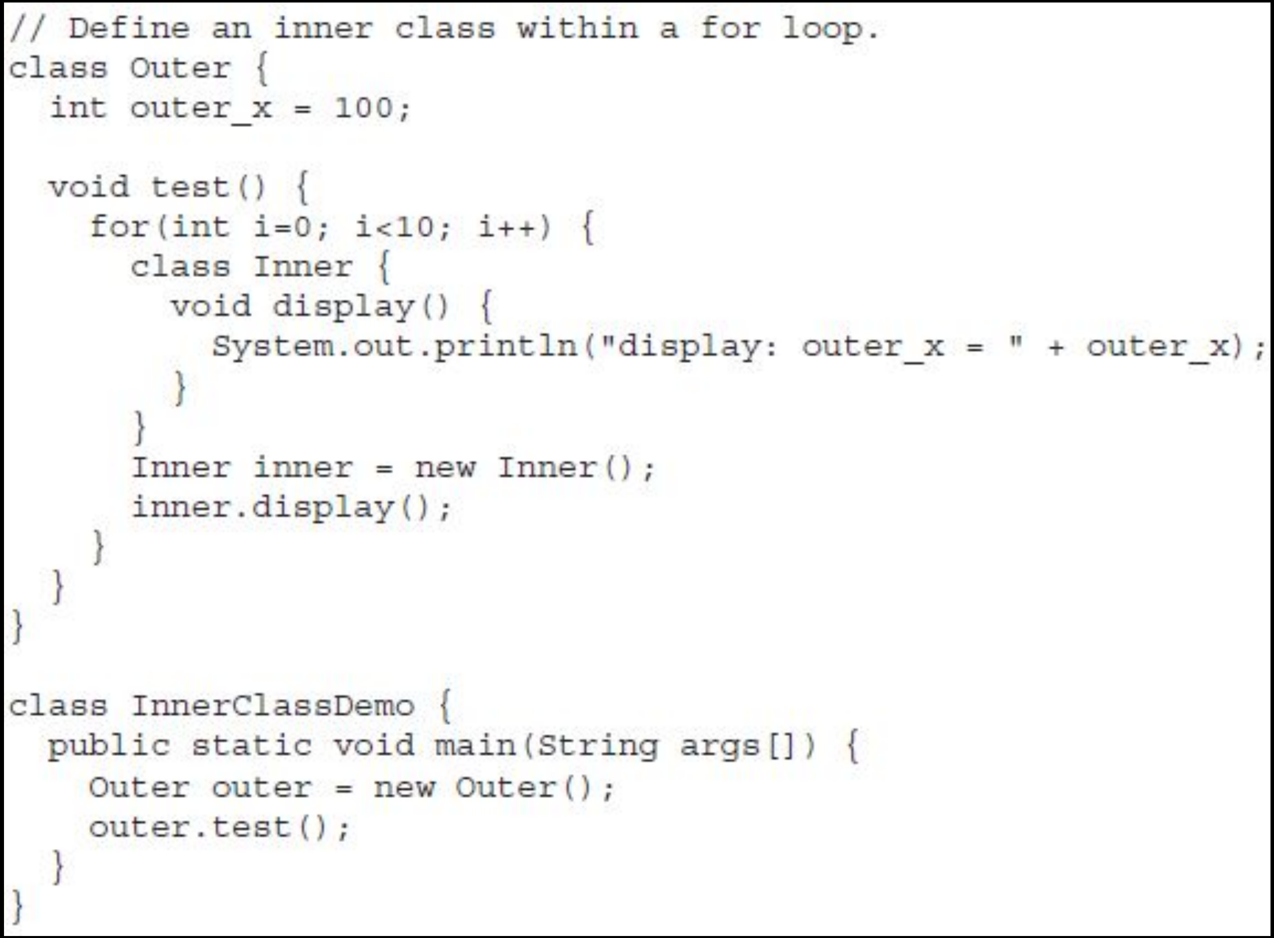
Nested & Inner classes: ... ***Inner/Non-Static classes...***



```
// This program will not compile.
class Outer {
  int outer_x = 100;

  void test() {
    Inner inner = new Inner();
    inner.display();
  }

  // this is an inner class
  class Inner {
    int y = 10; // y is local to Inner
    void display() {
      System.out.println("display: outer_x = " + outer_x);
    }
  }

  void showy() {
    System.out.println(y); // error, y not known here!
  }
}

class InnerClassDemo {
  public static void main(String args[]) {
    Outer outer = new Outer();
    outer.test();
  }
}
```

Introducing Classes

Nested & Inner classes: ... *Inner/Non-Static classes...*



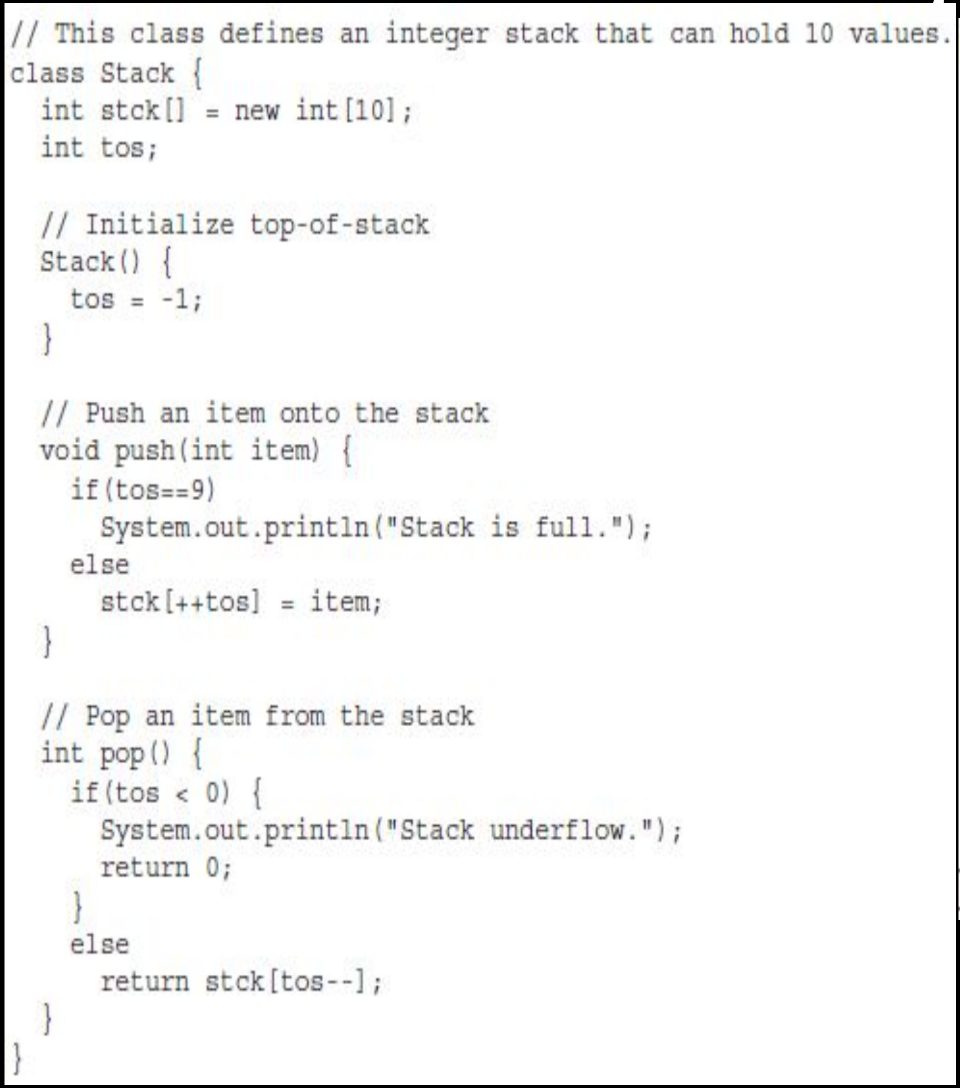
```
// Define an inner class within a for loop.
class Outer {
  int outer_x = 100;

  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();
    }
  }
}

class InnerClassDemo {
  public static void main(String args[]) {
    Outer outer = new Outer();
    outer.test();
  }
}
```

While nested classes are not applicable to all situations, they are particularly helpful when handling events.

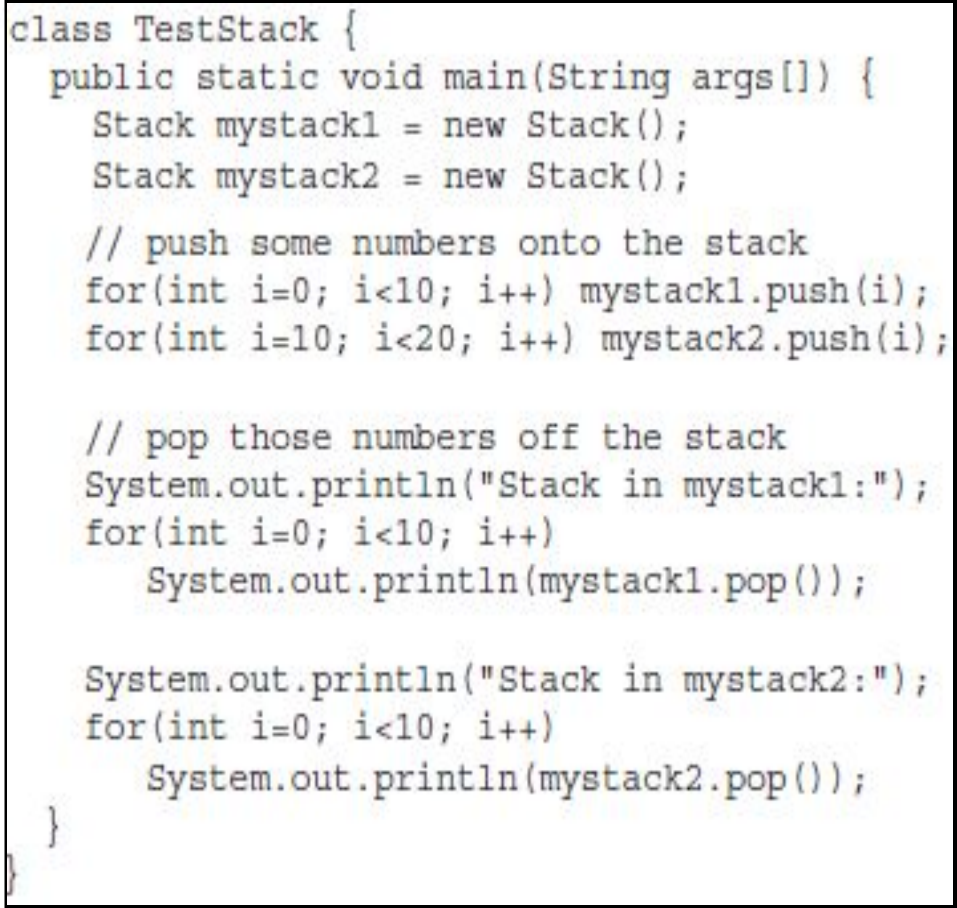
One final point: Nested classes were not allowed by the original 1.0 specification for Java. They were added by Java 1.1

Introducing Classes

The Stack class: *An example*...



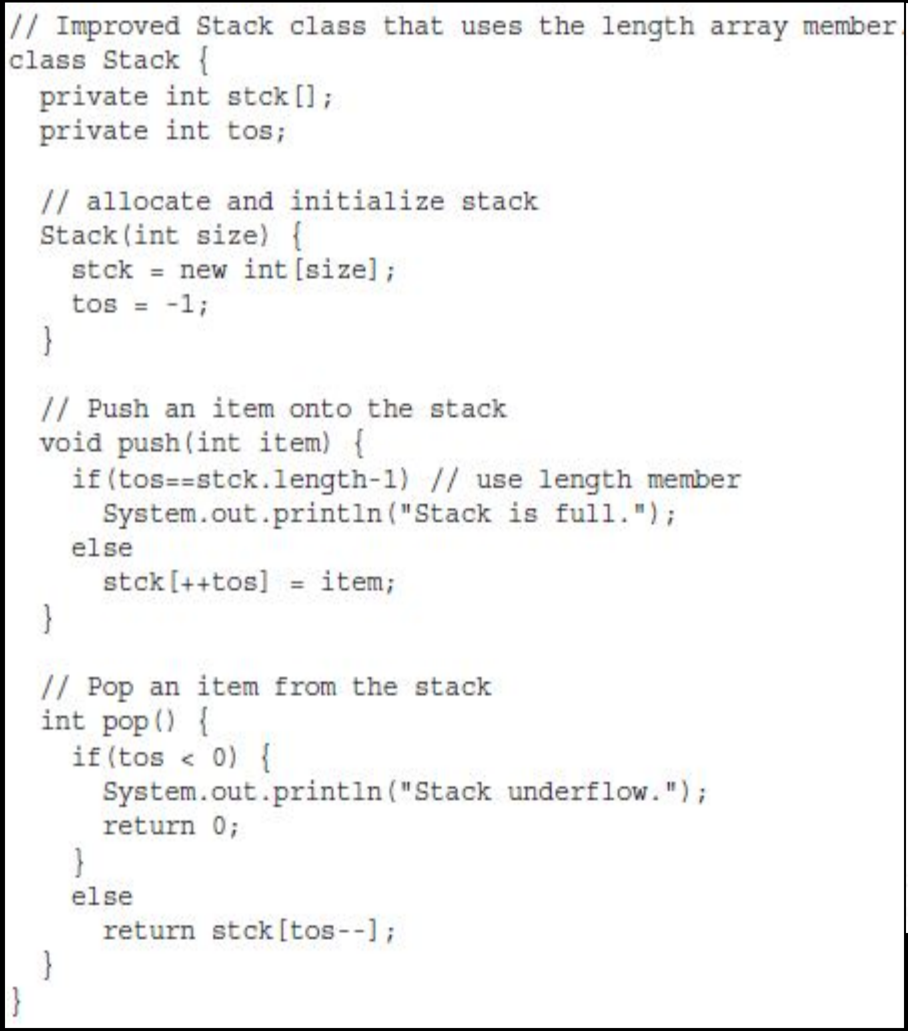
```
// This class defines an integer stack that can hold 10 values.
class Stack {
  int stck[] = new int[10];
  int tos;

  // 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--];
  }
}
```



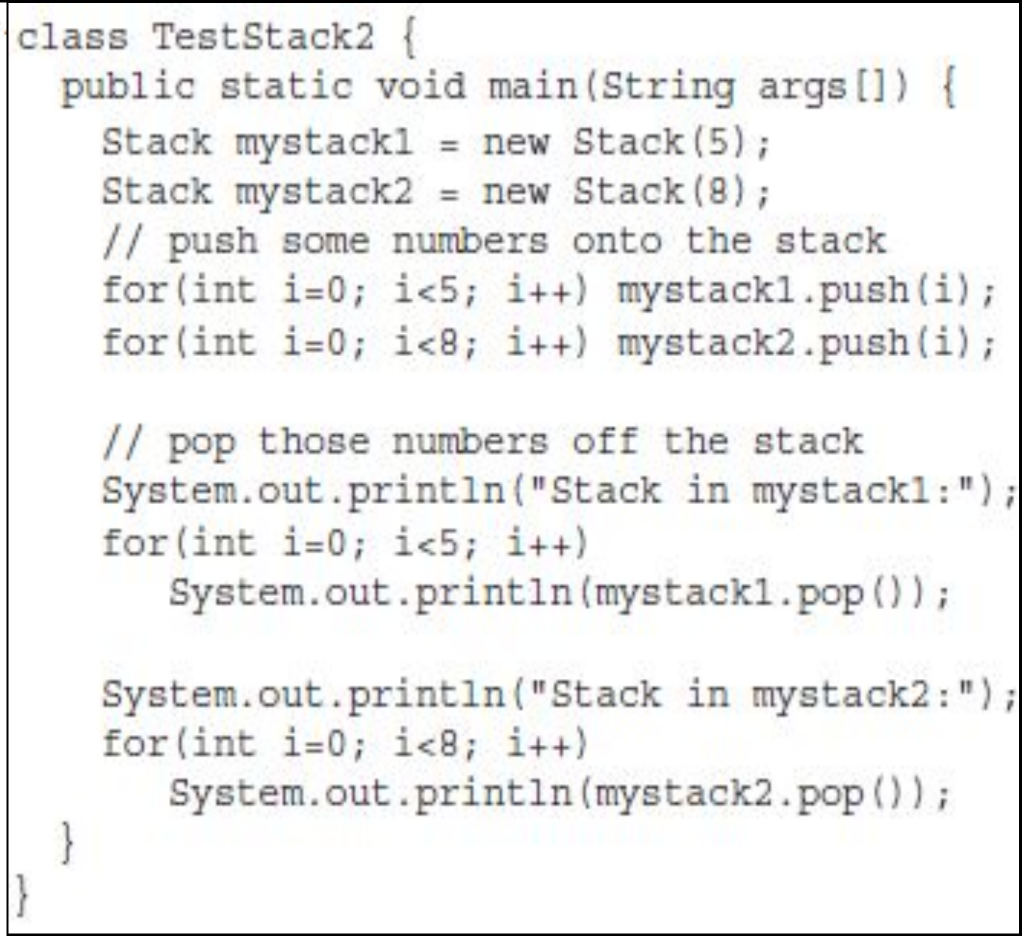
```
class TestStack {
  public static void main(String args[]) {
    Stack mystack1 = new Stack();
    Stack mystack2 = new Stack();

    // 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());
  }
}
```

Introducing Classes

The Stack class: *An example...Modified*



```
// Improved Stack class that uses the length array member.
class Stack {
  private int stck[];
  private int tos;

  // allocate and initialize stack
  Stack(int size) {
    stck = new int[size];
    tos = -1;
  }

  // Push an item onto the stack
  void push(int item) {
    if(tos==stck.length-1) // use length member
      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--];
  }
}
```



```
class TestStack2 {
  public static void main(String args[]) {
    Stack mystack1 = new Stack(5);
    Stack mystack2 = new Stack(8);
    // push some numbers onto the stack
    for(int i=0; i<5; i++) mystack1.push(i);
    for(int i=0; i<8; i++) mystack2.push(i);

    // pop those numbers off the stack
    System.out.println("Stack in mystack1:");
    for(int i=0; i<5; i++)
       System.out.println(mystack1.pop());

    System.out.println("Stack in mystack2:");
    for(int i=0; i<8; i++)
       System.out.println(mystack2.pop());
  }
}
```