CST281
Object Oriented Programming

MODULE 2

Object Oriented Programming in Java

Syllabus

- Primitive Data types - Integers, Floating Point Types, Characters, Boolean. Literals, Type Conversion and Casting, Variables, Arrays, Strings, Vector class.
- Operators - Arithmetic Operators, Bitwise Operators, Relational Operators, Boolean Logical Operators, Assignment Operator, Conditional (Ternary) Operator, Operator Precedence.
- Control Statements - Selection Statements, Iteration Statements and Jump Statements.
- Object Oriented Programming in Java - Class Fundamentals, Declaring Objects, Object Reference, Introduction to Methods, Constructors, ***this* *Keyword, Method Overloading, Using*** Objects as Parameters, Returning Objects, Recursion, Access Control, Static Members, Final Variables, Inner Classes, Command-Line Arguments, Variable Length Arguments.

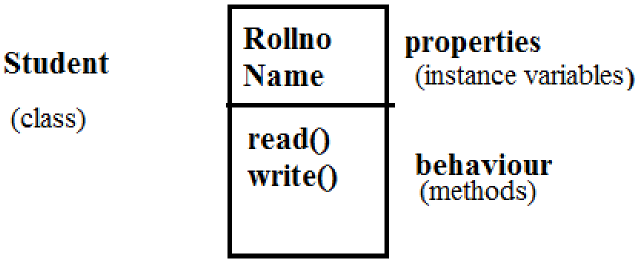
Methods, Constructors, *this Keyword, Method Overloading, Using* Objects as Parameters, Returning Objects, Recursion, Access Control, Static Members, Final Variables, Inner Classes, Command-Line Arguments, Variable Length Arguments.

OBJECT ORIENTED PROGRAMMING IN JAVA

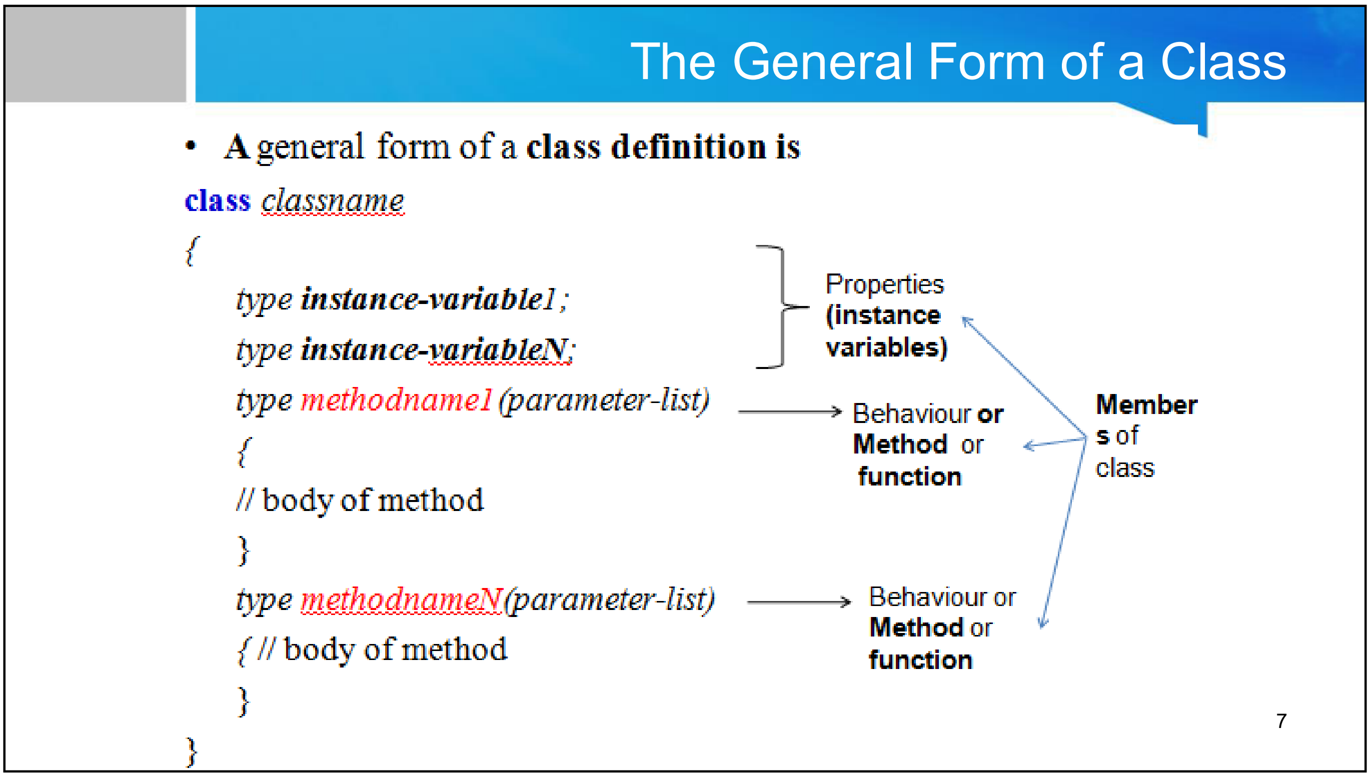
Class Fundamentals

- The class is the core of Java.
  - The class forms the basis for object-oriented programming in Java.
- A class is a "blueprint" for creating objects
- A class is a *template for an object.*
  - An object is an *instance of a class.*
- A class defines a *new type of data*.
- A class creates a *logical framework* that defines the relationship between its members.

Example



- A class is declared using the keyword class
- The data or 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.
  - the data for each object is separate and unique.
- Functions inside class are called methods.
- The methods and variables defined within a class are called members of the class.



The General Form of a Class

- A general form of a **class definition is**

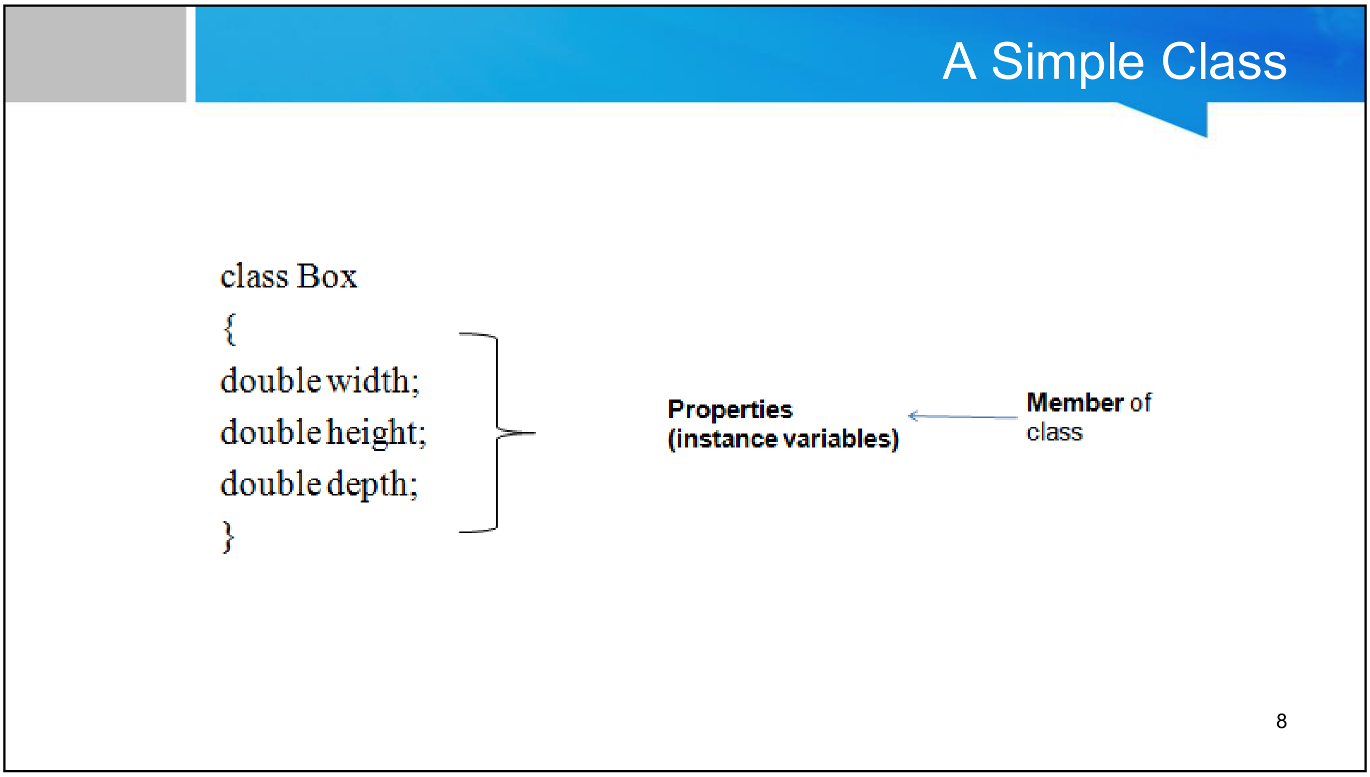
```
class classname
{
    type instance-variable1;
    type instance-variableN;
    type methodname1 (parameter-list)
    {
    // body of method
    }
    type methodnameN(parameter-list)
    { // body of method
    }
}
```

Properties (instance variables)

Behaviour **or** **Method** or **function**

Behaviour or **Method** or **function**

Members of class



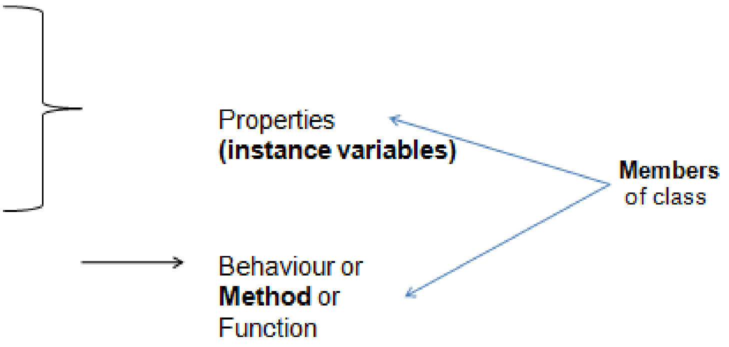
A Simple Class

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

Properties (instance variables)

Member of class

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

void volume()
{
//statements
}

}
```



Declaring Objects

- When we create a class, we are creating a new data type.
  - We can use this type to declare objects of that type.
- Obtaining objects of a class is a two-step process.
  - First, we must declare a variable of the class type.
    - This variable does not define an object.
    - It is simply a variable that can refer to an object.
  - Second, we must acquire an actual, physical copy of the object and assign it to that variable (using new operator)

```
Classname objectname ;        // declare reference to object
objectname = new Classname(); // allocate an object
```

- We can write this in a single statement

```
Classname objectname = new Classname();
```

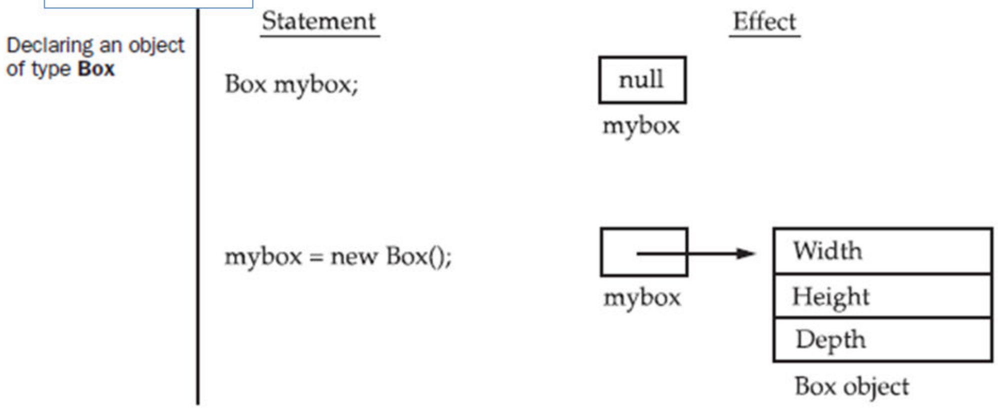
```
class Box
{
    double Width;
    double Height;
    double Depth;
}
Box mybox;
```

- This line declares **mybox** as a **reference to** an object of type **Box**.
- Here **mybox** contains the value **null**, which indicates that it does not yet point to an actual object

mybox = new **Box**();

- This line allocates an actual object and assigns a reference to it to **mybox.**
- **mybox** holds the memory address of the actual Box object.

```
class Box
{double Width;
double Height;
double Depth;
}
```



- The class name followed by parentheses specifies the constructor for the class.

Box mybox=new Box();

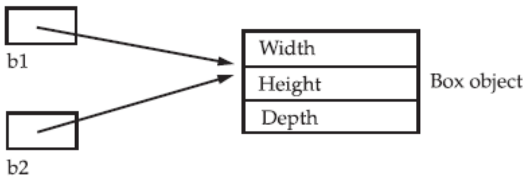
- Here Box is the class. Box()is the constructor.
- A constructor defines what occurs when an object of a class is created.

Assigning Object Reference Variables

- Object reference variables act differently when an assignment takes place
- E.g.

Box b1 = new Box();

Box b2 = b1;



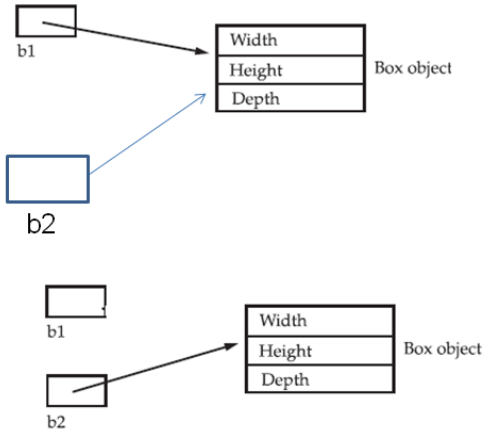
- Here b1 and b2 will both refer to the ***same object.***
- Any changes made to the object through b2 will affect the object which is referred by b1, because they are the same object.

Box b1 = new Box();

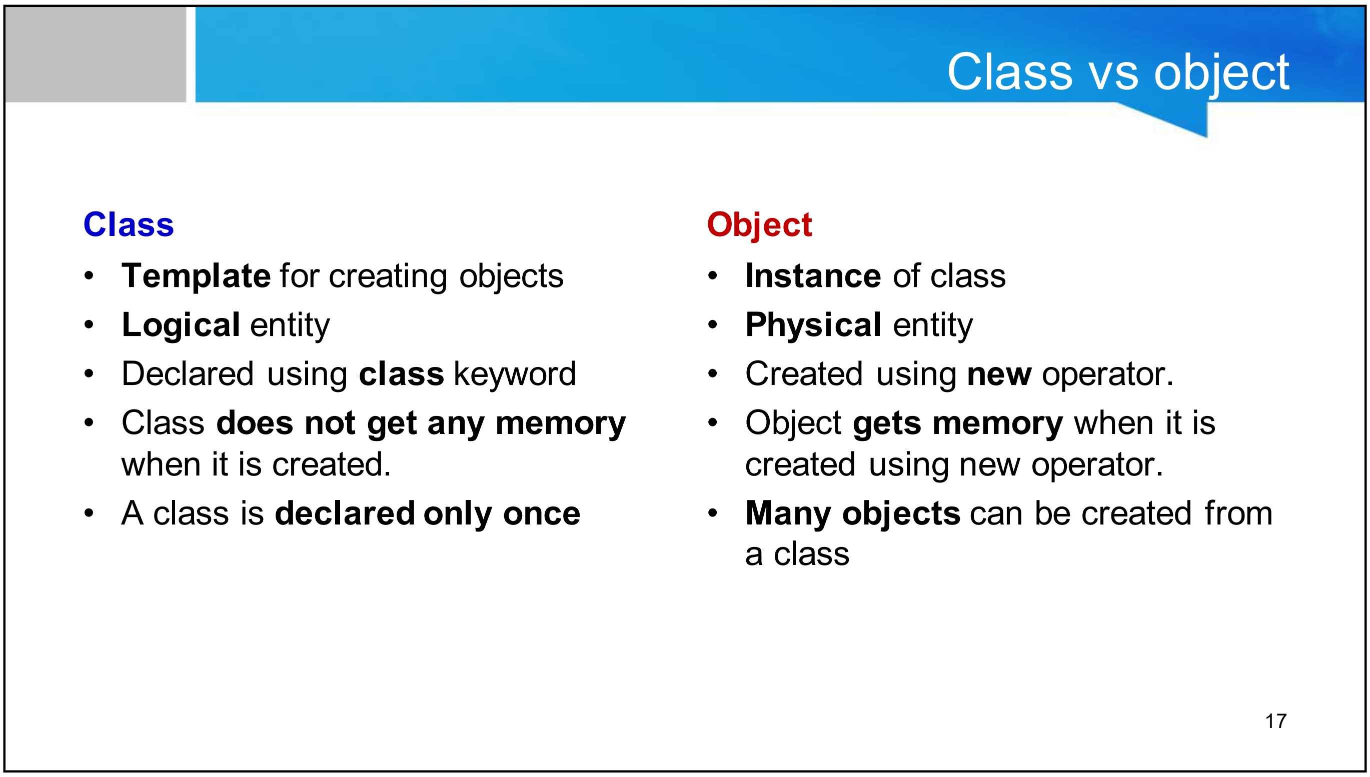
Box b2 = b1;

// ...

b1 = null;



- Here at the end b1 has been set to null, but b2 still points to the original object.



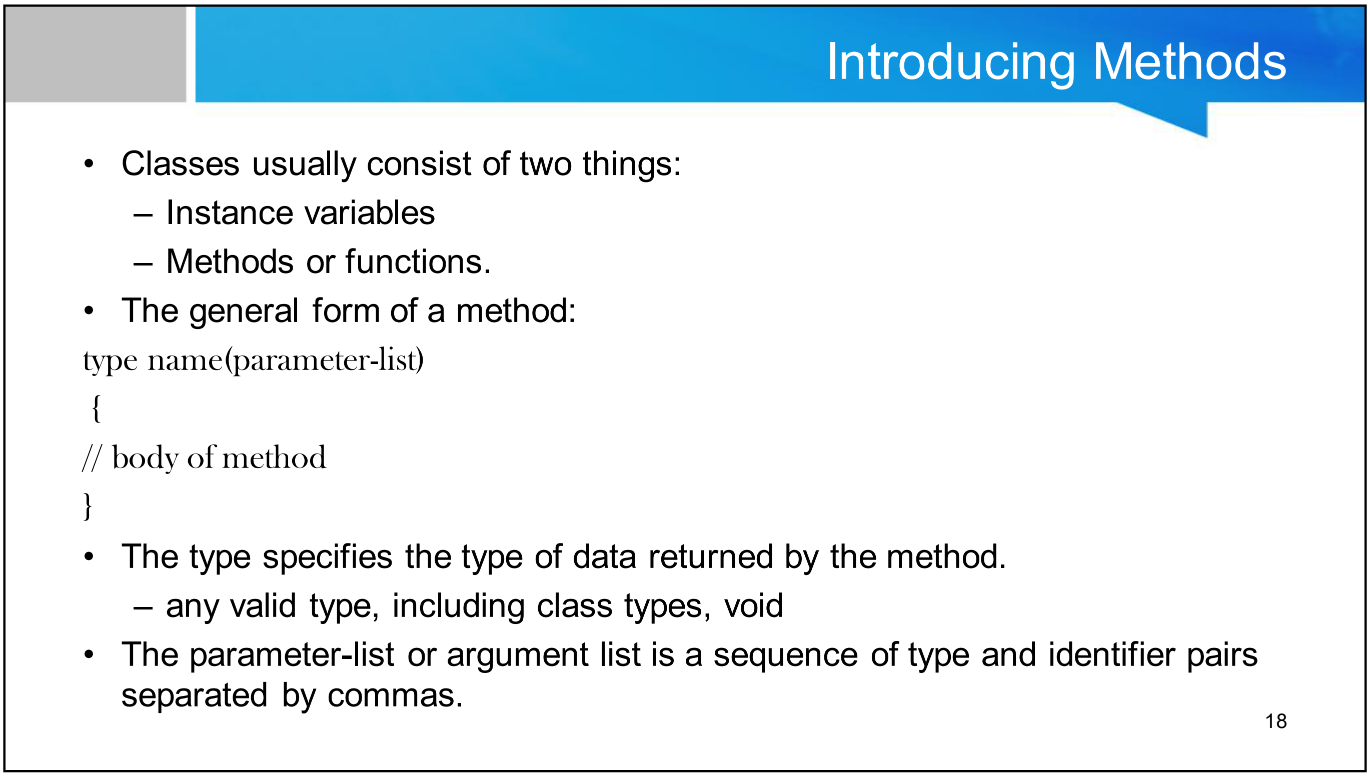
Class vs object

Class

- **Template** for creating objects
- **Logical** entity
- Declared using **class** keyword
- Class **does not get any memory** when it is created.
- A class is **declared only once**

Object

- **Instance** of class
- **Physical** entity
- Created using **new** operator.
- Object **gets memory** when it is created using new operator.
- **Many objects** can be created from a class



Introducing Methods

- Classes usually consist of two things:
  - – Instance variables
  - – Methods or functions.
- The general form of a method:

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

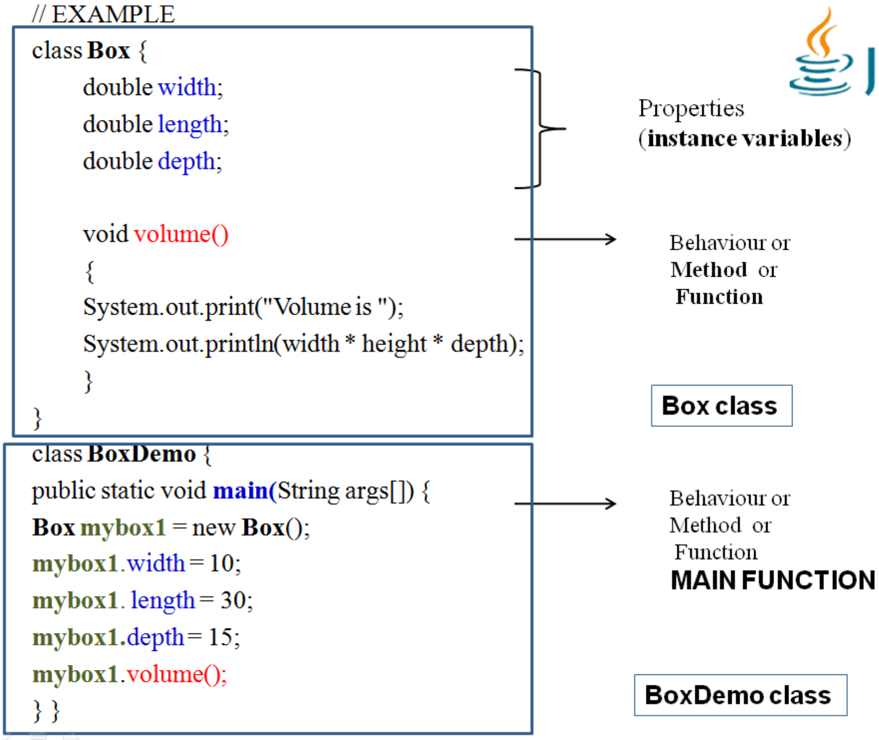
- The type specifies the type of data returned by the method.
  - – any valid type, including class types, void
- The parameter-list or argument list is a sequence of type and identifier pairs separated by commas.

- 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;

- Method of one class can be invoked by functions of other classes through objects of former class.

Objectname.method(parameters);



```
// EXAMPLE
class Box {
        double width;
        double length;
        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. length = 30;
mybox1.depth = 15;
mybox1.volume();
} }
```

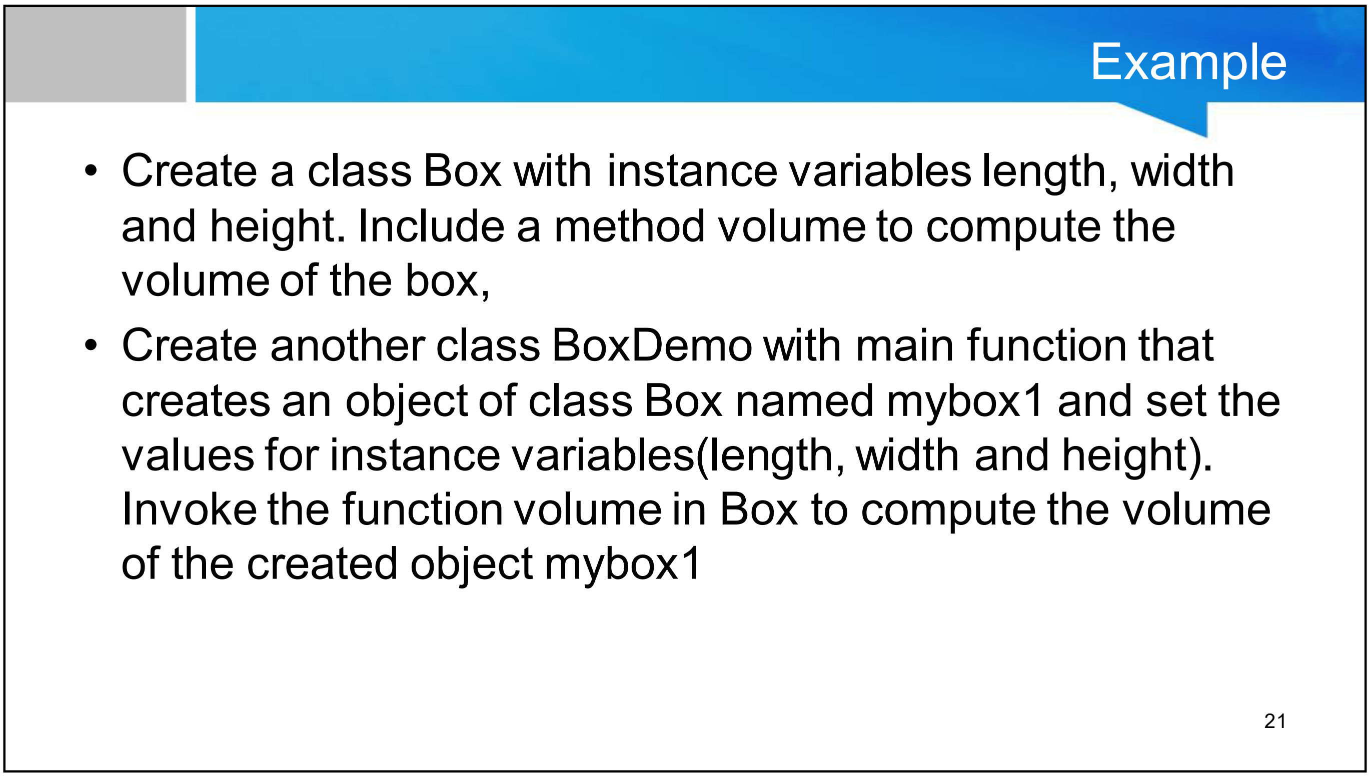
Properties
(**instance variables**)

Behaviour or
Method or
Function

Box class

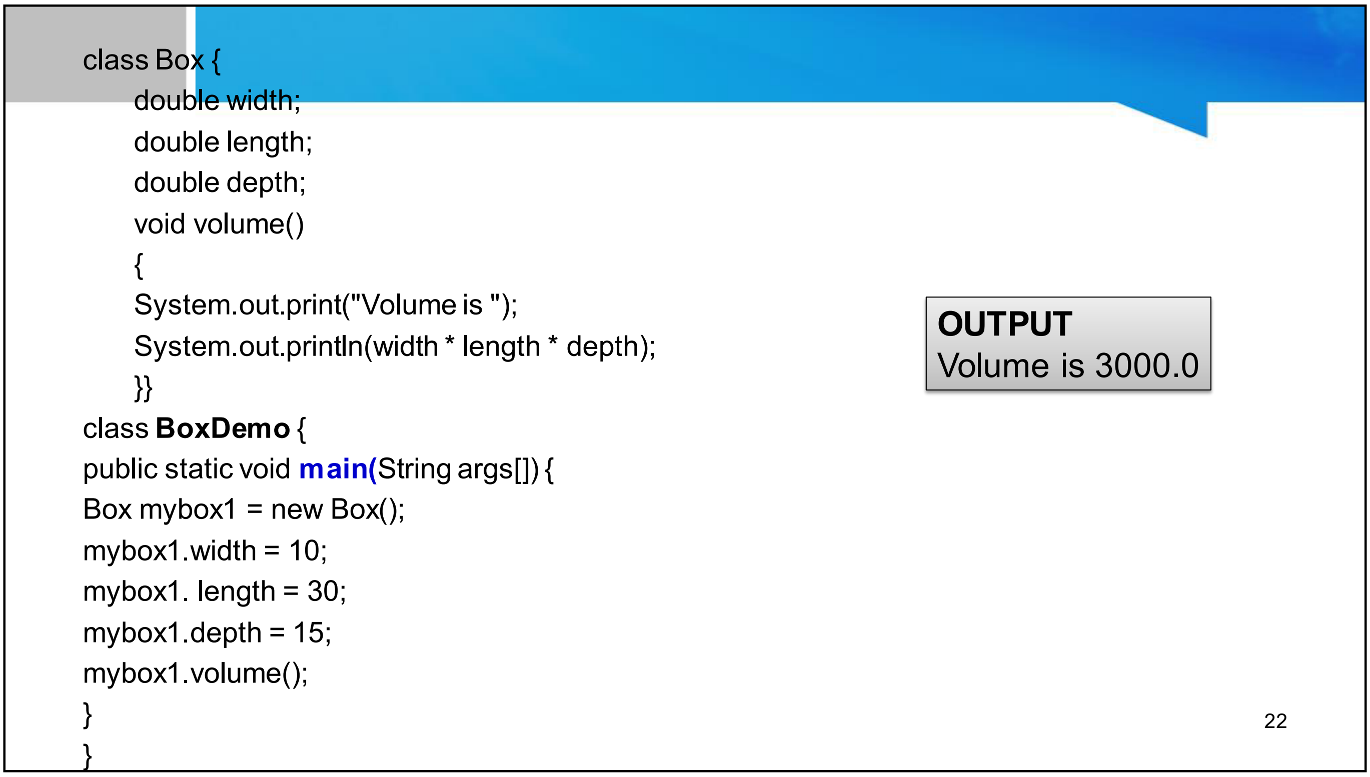
Behaviour or
Method or
Function
MAIN FUNCTION

BoxDemo class



Example

- Create a class Box with instance variables length, width and height. Include a method volume to compute the volume of the box,
- Create another class BoxDemo with main function that creates an object of class Box named mybox1 and set the values for instance variables(length, width and height). Invoke the function volume in Box to compute the volume of the created object mybox1



```
class Box {
        double width;
        double length;
        double depth;
        void volume()
        {
        System.out.print("Volume is ");
        System.out.println(width * length * depth);
        }}
class BoxDemo {
public static void main(String args[]) {
Box mybox1 = new Box();
mybox1.width = 10;
mybox1. length = 30;
mybox1.depth = 15;
mybox1.volume();
}
}
```

OUTPUT

```
Volume is 3000.0
```

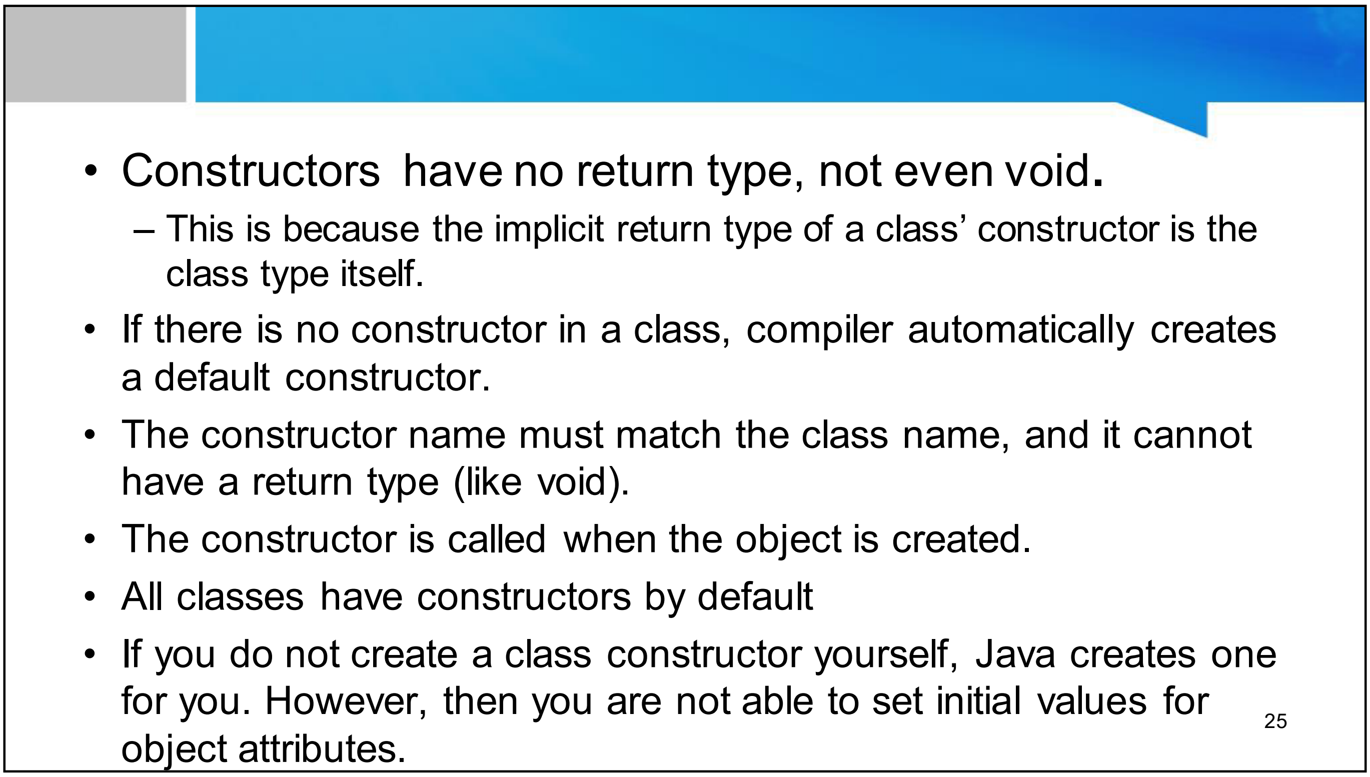
```
// program using return statement
class Box {
        double width;
        double height;
        double depth;

        int volume()
        {
        return(width * length * depth);
        }
}
class BoxDemo {
public static void main(String args[]) {
Box mybox1 = new Box();
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
int v=mybox1.volume();
System.out.println(”Volume=“+v);
} }
```

OUTPUT

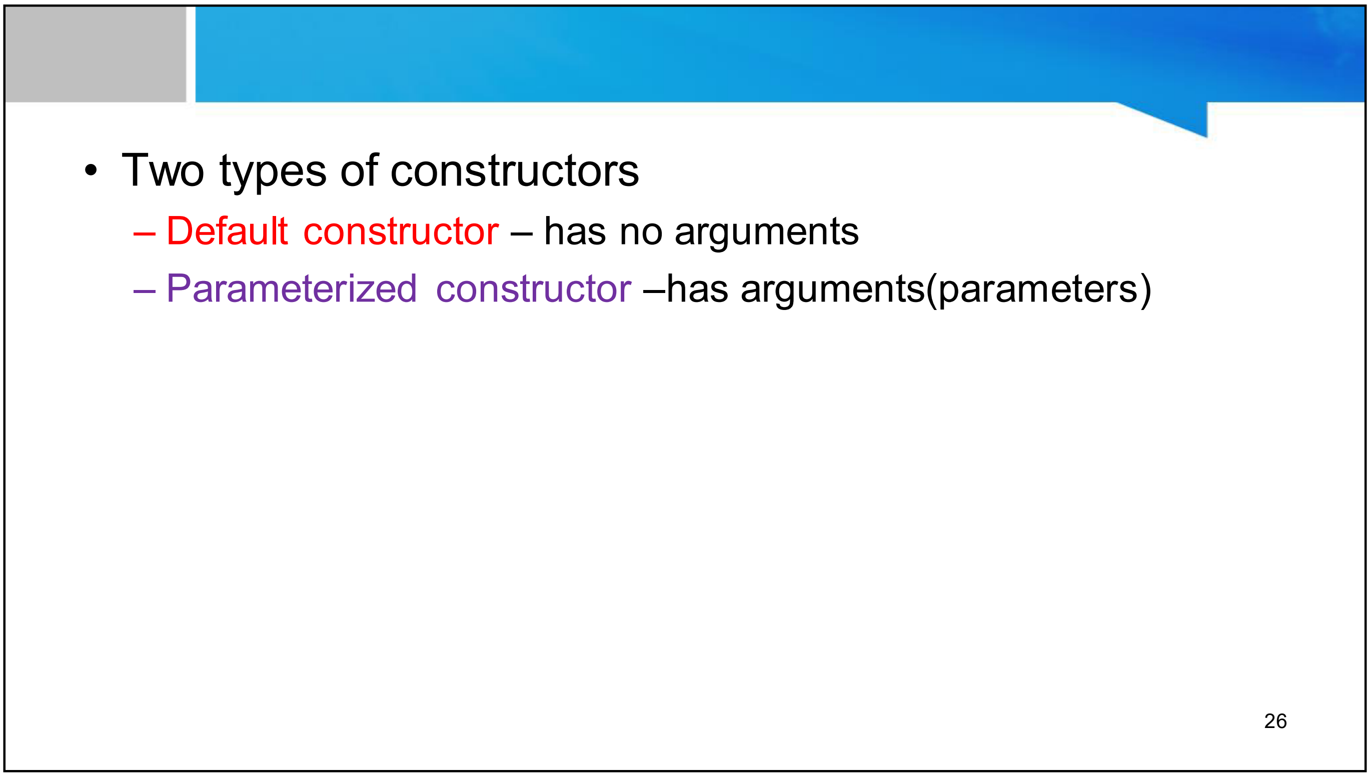
```
Volume is 3000.0
```

Constructor

- A constructor help to initialize an object(give values) immediately upon creation.
- Constructor is a special method inside the class.
- Constructor has the same name as the class in which it resides.
- Once defined, the constructor is automatically called immediately after the object is created, before the new operator completes.



- Constructors have no return type, not even void.
  - This is because the implicit return type of a class' constructor is the class type itself.
- If there is no constructor in a class, compiler automatically creates a default constructor.
- The constructor name must match the class name, and it cannot have a return type (like void).
- The constructor is called when the object is created.
- All classes have constructors by default
- If you do not create a class constructor yourself, Java creates one for you. However, then you are not able to set initial values for object attributes.

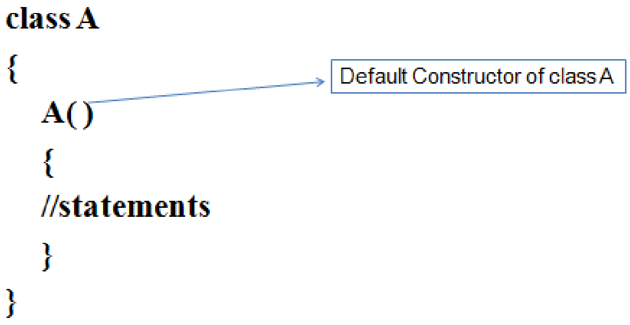


- Two types of constructors
  - Default constructor – has no arguments
  - Parameterized constructor –has arguments(parameters)

Default constructor

- Default constructor has no arguments or parameters.
- When we do not explicitly define a constructor for a class, then Java creates a default constructor for the class.
- The purpose of a default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

E.g.



```
class A
{
   A()
   {
   //statements
   }
}
```

Default Constructor of class A

```
class Box
{
int width ,length,height;
Box()
{
width=10;
length=10;
height=10;
}}
```

- The following statement creates an object of class Box.

Box mybox1 = **new Box();**

- Here **new Box( )** is calling the **Box( )** constructor.

```
class Box {
      int length;
      int height;
      int width;
      Box()
      {System.out.println(“Constructor");
      width = 10;
      length = 10;
      height= 10;}
      int volume()
      {
      return width * length * height;
      }
}
```

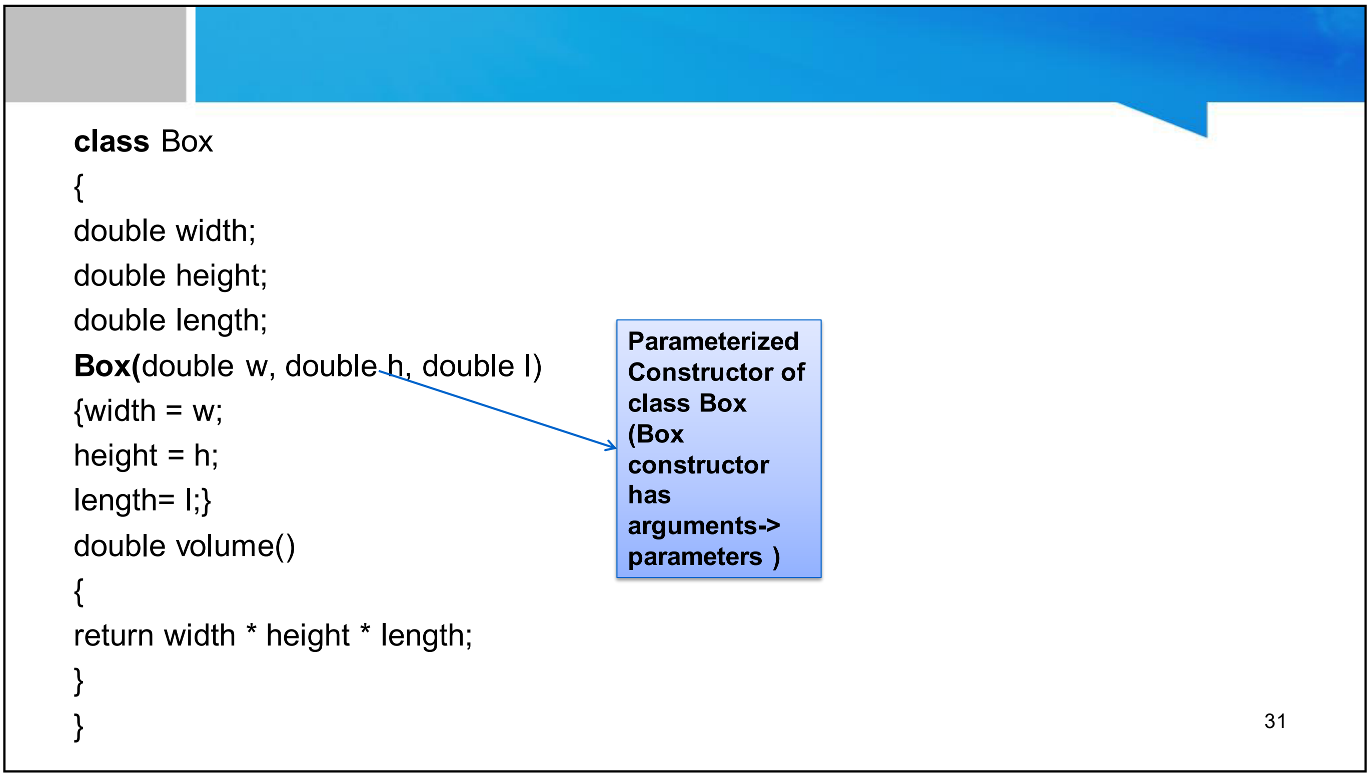
```
class BoxDemo {
public static void main(String args[])
{
Box mybox1 = new Box();
Box mybox2 = new Box();
int vol;
vol = mybox1.volume();
System.out.println("Volume is " + vol);
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

OUTPUT

```
Constructor
Constructor
Volume is 1000
Volume is 1000
```

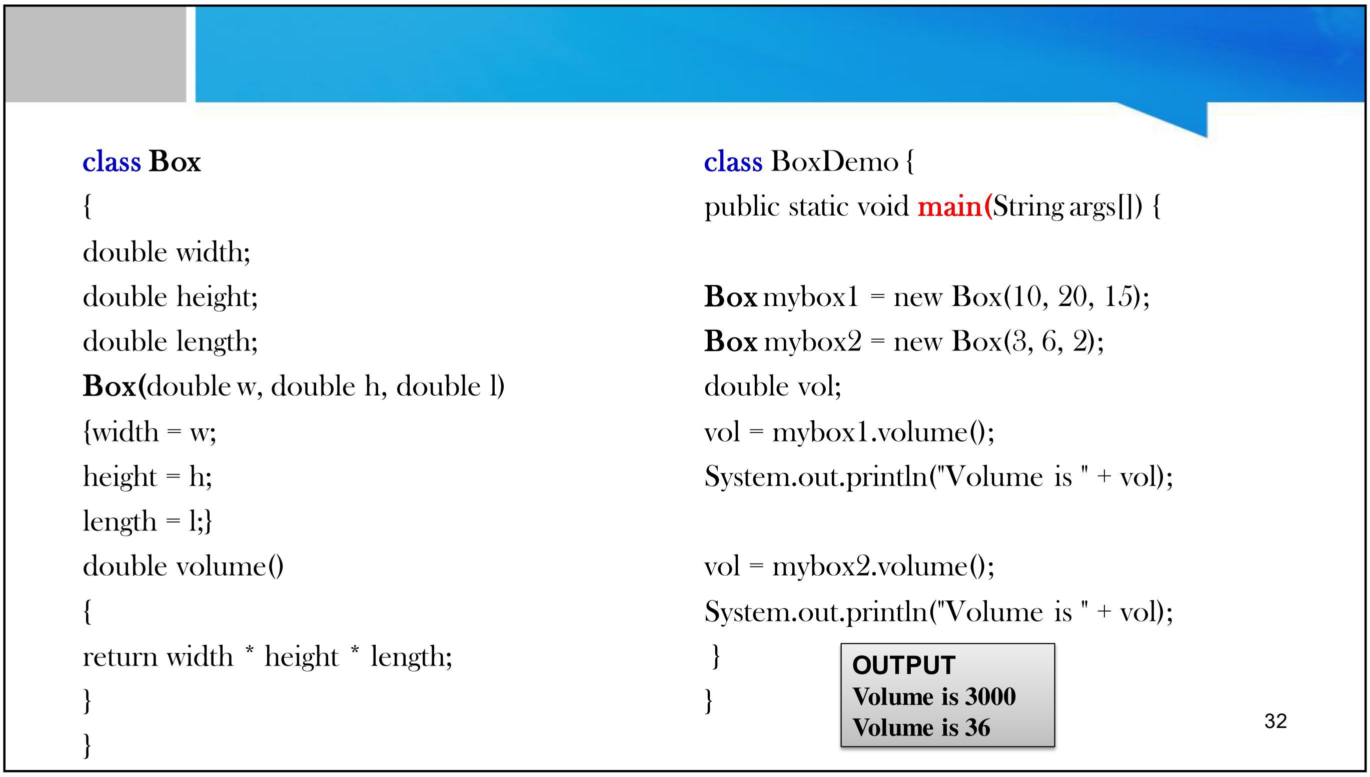
Parameterized Constructors

- Constructors with arguments are called parameterized constructors.



```
class Box
{
double width;
double height;
double length;
Box(double w, double h, double l)
{width = w;
height = h;
length= l;}
double volume()
{
return width * height * length;
}
}
```

Parameterized Constructor of class Box (Box constructor has arguments-> parameters)



```
class Box
{
double width;
double height;
double length;
Box(double w, double h, double l)
{width = w;
height = h;
length = l;}
double volume()
{
return width * height * length;
}
}
```

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

Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box(3, 6, 2);
double vol;
vol = mybox1.volume();
System.out.println("Volume is " + vol);

vol = mybox2.volume();
System.out.println("Volume is " + vol);
 }
}
```

OUTPUT

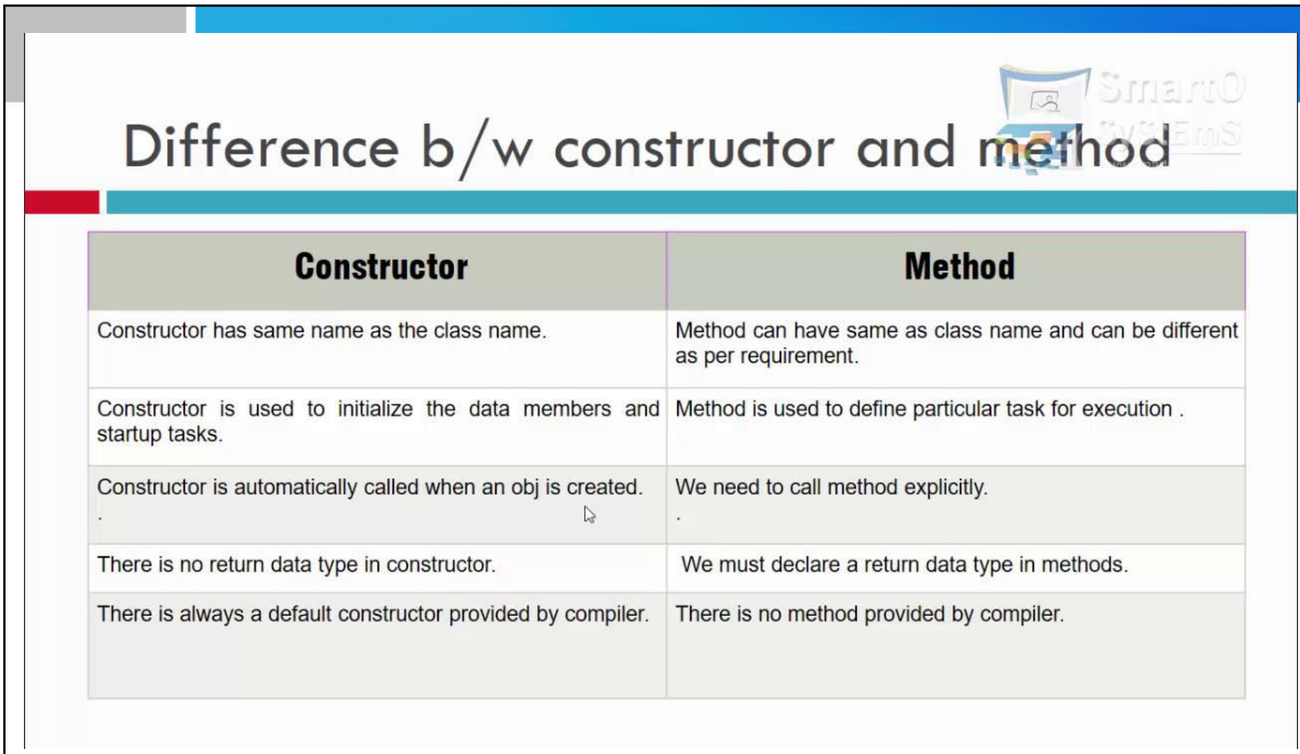
```
Volume is 3000
Volume is 36
```

```
Box mybox1 = new Box(10, 20, 15);
```

- Here the values 10, 20, and 15 are passed to the **Box( )** constructor when new creates the object mybox1.
- The parameterized constructor is

```
Box(double w, double h, double l)
{
width = w;
height = h;
length = l;
}
```

- Thus, value of mybox1 object’s width, height, and depth will be set as 10, 20, and 15 respectively.



Difference b/w constructor and method

Constructor	Method
Constructor has same name as the class name.	Method can have same as class name and can be different as per requirement.
Constructor is used to initialize the data members and startup tasks.	Method is used to define particular task for execution .
Constructor is automatically called when an obj is created. .	We need to call method explicitly. .
There is no return data type in constructor.	We must declare a return data type in methods.
There is always a default constructor provided by compiler.	There is no method provided by compiler.

The this Keyword

- The **this** keyword can be used inside any method to refer to the **current object.**
- **this** is always a reference to the object on which the method was invoked.
- **this** can be used to refer current class instance variable.
- **this** can be used to invoke current class method (implicitly)
- **this**() can be used to invoke current class constructor.
- **this** can be passed as an argument in the method call.
- **this** can be passed as argument in the constructor call.

```
Box(double w, double h, double l)
 {
this.width = w;
this.height = h;
this.length = l;
}
```

Here **this** will always refer to the object invoking the method

```
class Box
{
double width;
double length;
double height;
Box(double w, double l, double h)
{
this.width = w;
this.length = l;
this.height = h;
}
}
```

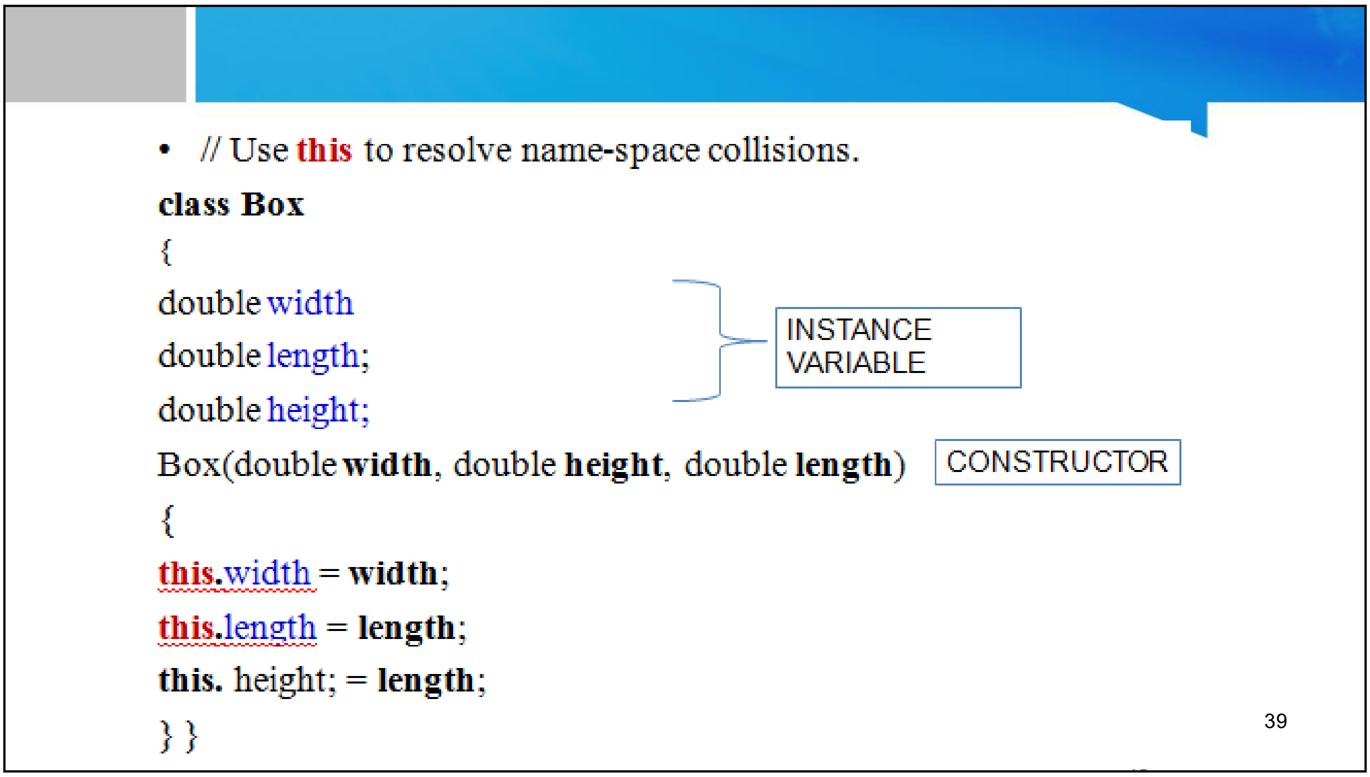
```
class BoxDemo {
public static void main(String args[]) {
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box(3, 6, 2);
}
}
```

Here in statement
Box mybox1 = new Box(10, 20, 15);
mybox1 object is created by calling parameterized constructor.
Box(double w, double l, double d)
Here **this** inside constructor refers to object mybox1.

Next when **mybox2** object is created, **this** refers to object mybox2.

Instance variable hiding-using this

- We can have local variables, including formal parameters to methods, which has the same name of the class’ instance variables(attributes).
- But when a local variable has the same name as an instance variable, the local variable hides the instance variable.
  - this helps to solve this. Use this. along with instance variables.

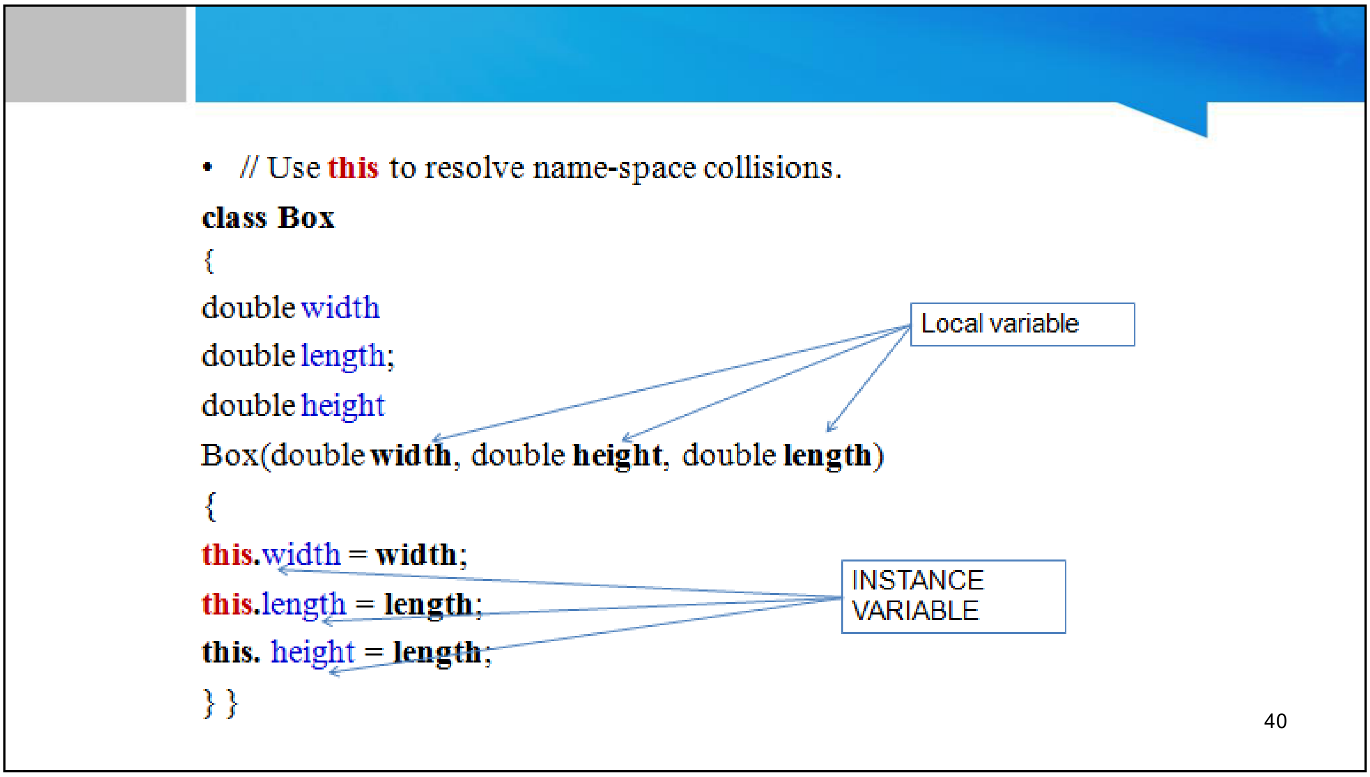


- // Use **this** to resolve name-space collisions.

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

INSTANCE VARIABLE

CONSTRUCTOR



- // Use **this** to resolve name-space collisions.

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

Local variable

INSTANCE VARIABLE

- Object Oriented Programming in Java

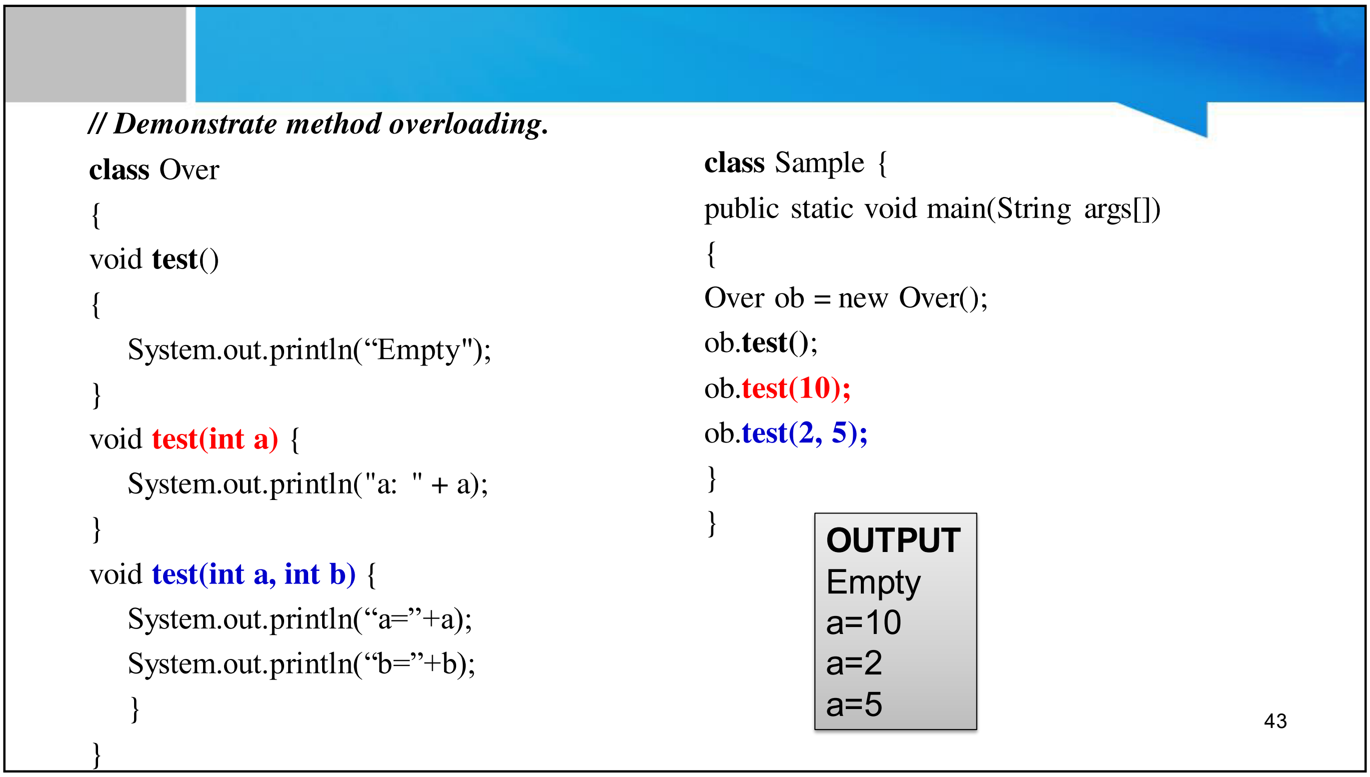
  ✓ 1. Class Fundamentals, Declaring Objects, Object Reference, Introduction to Methods,

  ✓ 2. Constructors, this Keyword,

  3. Method Overloading, Using Objects as Parameters, Returning Objects, Recursion,
  4. Access Control, Static Members, Final Variables, Inner Classes, Command-Line Arguments, Variable Length Arguments.

Method Overloading

- It is possible to define two or more methods with same name within the same class, but their parameter declarations should be different.
  - This is called method overloading.
  - This is a form of polymorphism (many forms)
- Overloaded methods must differ in the type and/or number of their parameters. (return types is not significant.)
- When an overloaded method is invoked, Java uses the type and/or number of arguments to determine which version of the overloaded method to actually call.

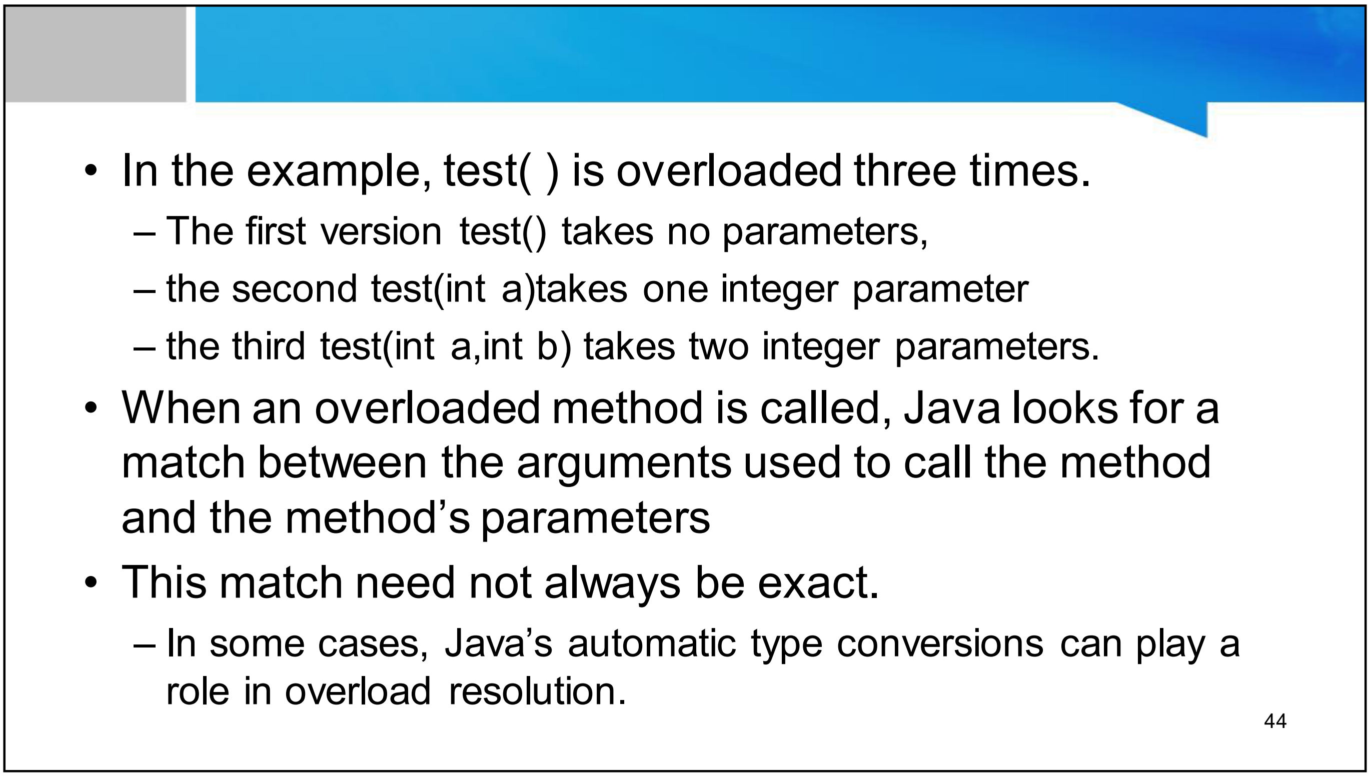


```
// Demonstrate method overloading.
class Over
{
void test()
{
    System.out.println("Empty");
}
void test(int a) {
    System.out.println("a:  " + a);
}
void test(int a, int b) {
    System.out.println("a="+a);
    System.out.println("b="+b);
    }
}
```

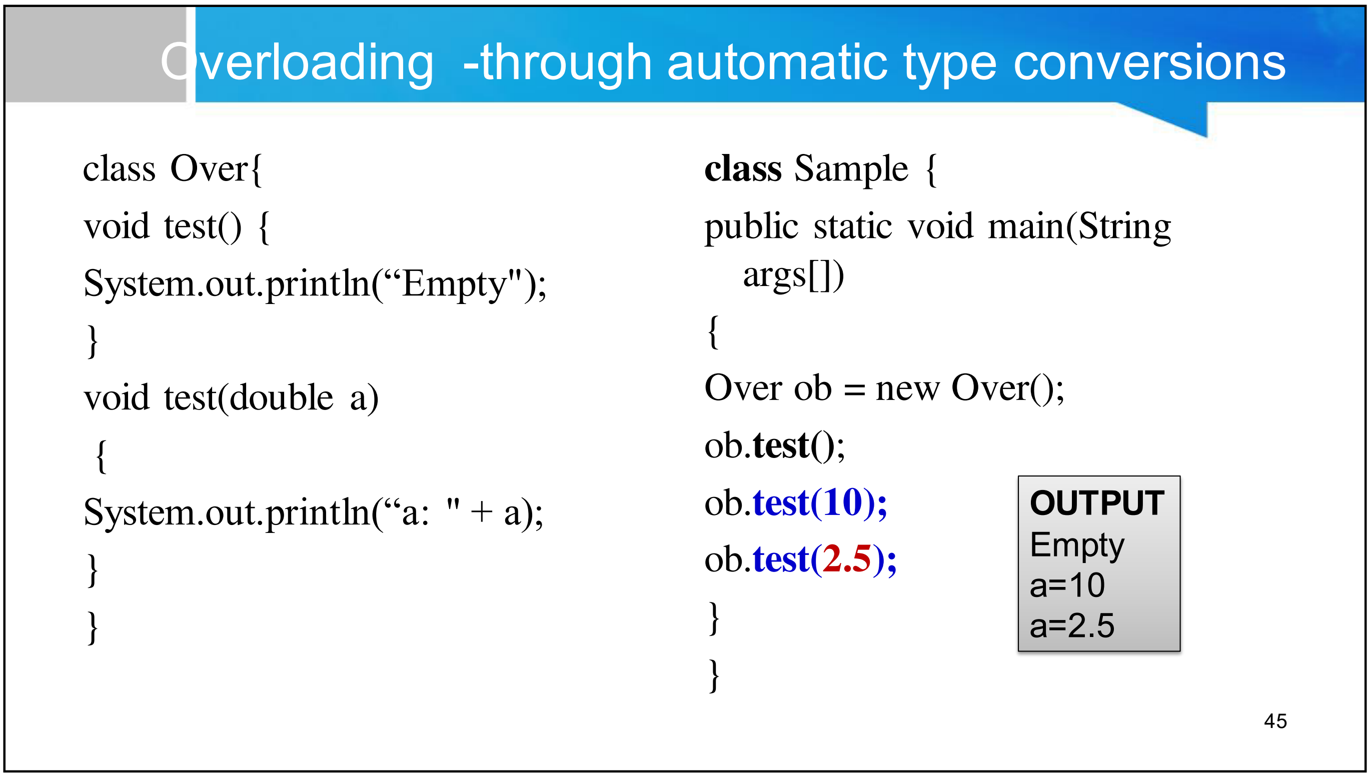
```
class Sample {
public static void main(String args[])
{
Over ob = new Over();
ob.test();
ob.test(10);
ob.test(2, 5);
}
}
```

OUTPUT

```
Empty
a=10
a=2
a=5
```



- In the example, test( ) is overloaded three times.
  - The first version test() takes no parameters,
  - the second test(int a)takes one integer parameter
  - the third test(int a,int b) takes two integer parameters.
- When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method’s parameters
- This match need not always be exact.
  - In some cases, Java’s automatic type conversions can play a role in overload resolution.



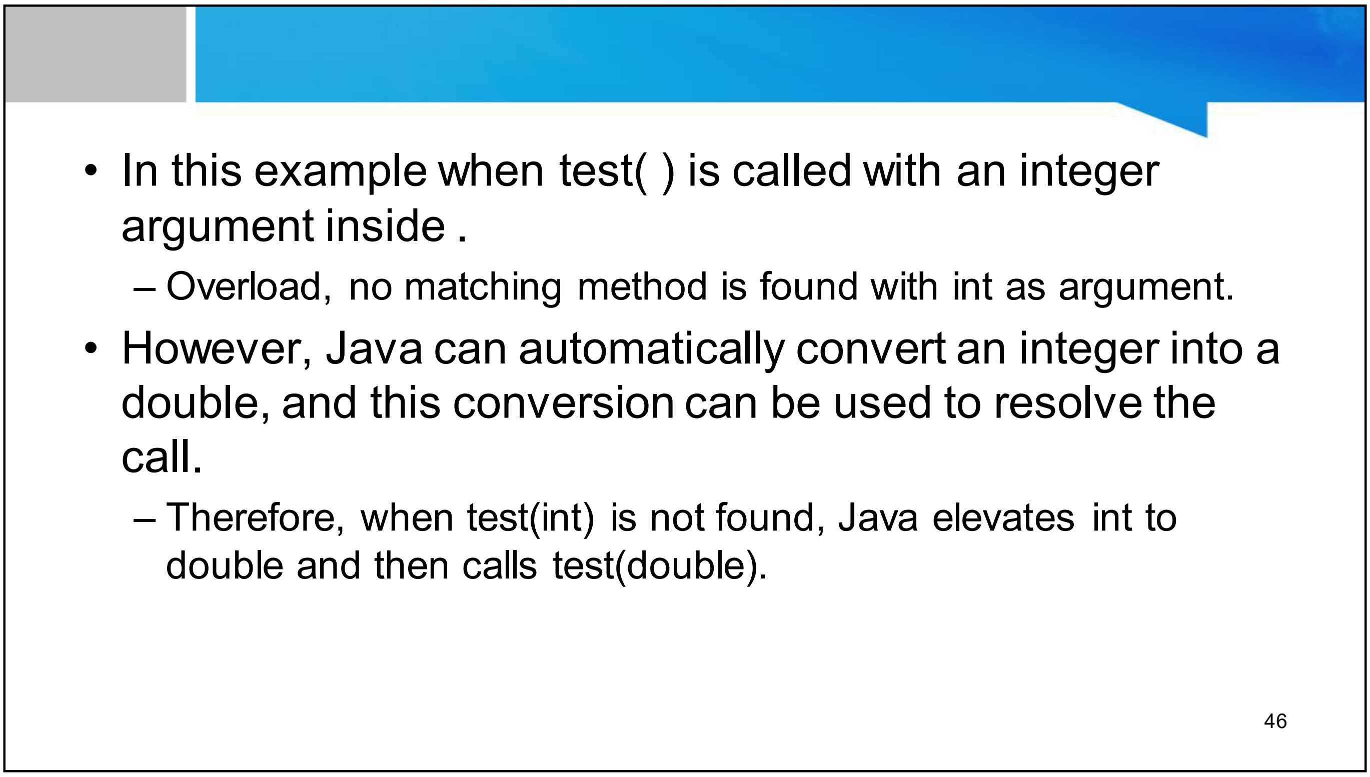
Overloading -through automatic type conversions

```
class Over{
void test() {
System.out.println(“Empty");
}
void test(double a)
 {
System.out.println(“a: " + a);
}
}
```

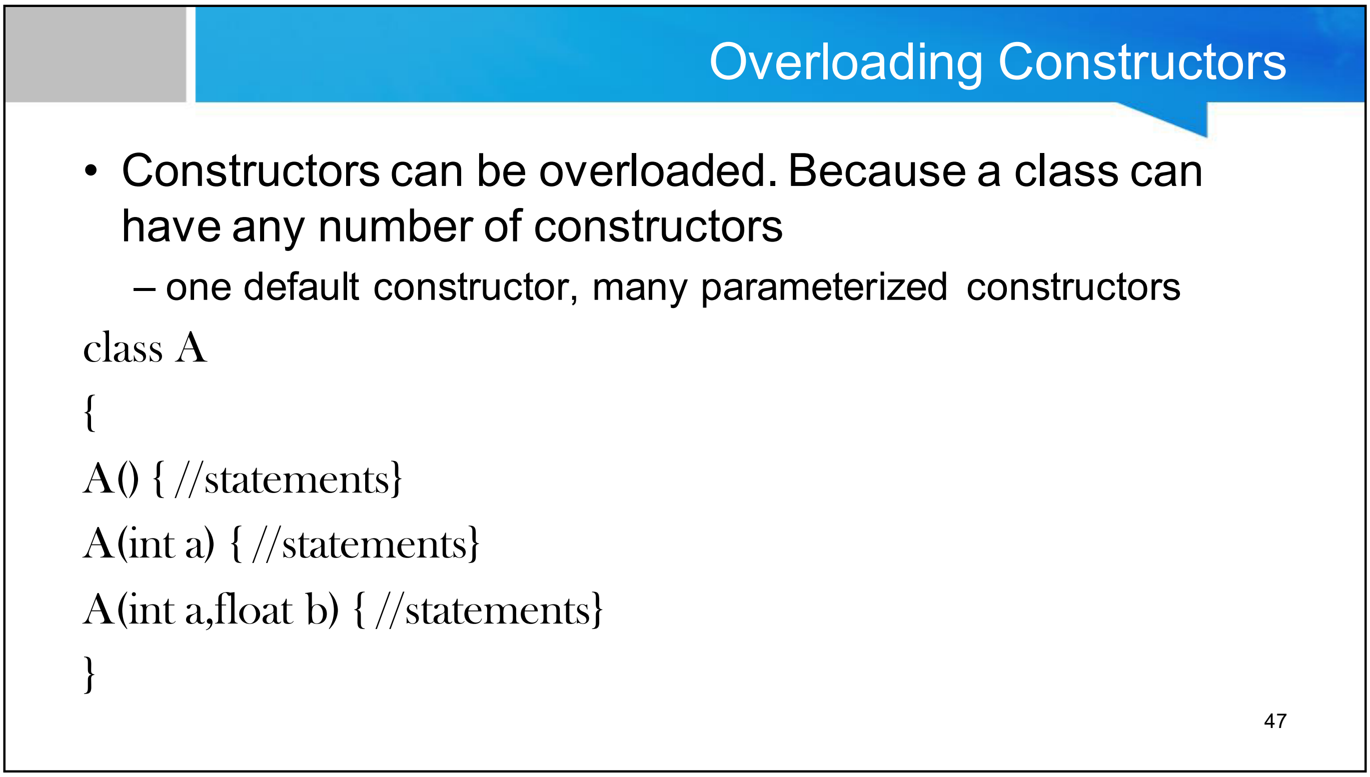
```
class Sample {
public static void main(String
   args[])
{
Over ob = new Over();
ob.test();
ob.test(10);
ob.test(2.5);
}
}
```

OUTPUT

```
Empty
a=10
a=2.5
```



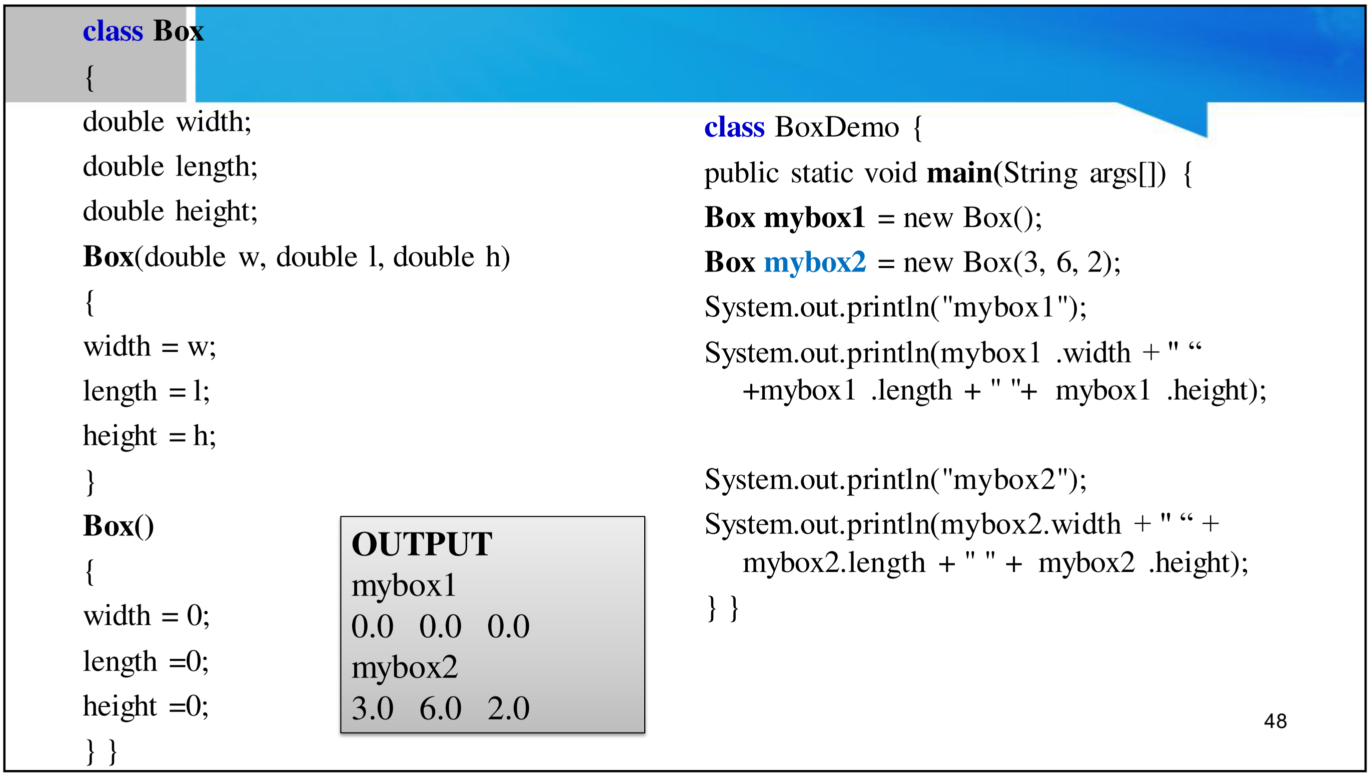
- In this example when test( ) is called with an integer argument inside .
  - Overload, no matching method is found with int as argument.
- However, Java can automatically convert an integer into a double, and this conversion can be used to resolve the call.
  - Therefore, when test(int) is not found, Java elevates int to double and then calls test(double).



Overloading Constructors

- Constructors can be overloaded. Because a class can have any number of constructors
  - – one default constructor, many parameterized constructors

```
class A
{
A() { //statements}
A(int a) { //statements}
A(int a,float b) { //statements}
}
```



```
class Box
{
double width;
double length;
double height;
Box(double w, double l, double h)
{
width = w;
length = l;
height = h;
}
Box()
{
width = 0;
length =0;
height =0;
} }
```

OUTPUT

```
mybox1
0.0   0.0   0.0
mybox2
3.0   6.0   2.0
```

```
class BoxDemo {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box(3, 6, 2);
System.out.println("mybox1");
System.out.println(mybox1 .width + " “
   +mybox1 .length + " "+ mybox1 .height);

System.out.println("mybox2");
System.out.println(mybox2.width + " “ +
   mybox2.length + " " + mybox2 .height);
} }
```

```
class Box
{
double width
double length;
double height;
Box(double w, double l, double h)
{
this.width = w;
this.length = l;
this.height = h;
}
}
```

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

Box mybox1 = new Box(); //ERROR
Box mybox2 = new Box(3, 6, 2);
}
}
```

ERROR

Here following statement tries to create object mybox1 of class Box ,

Box mybox1 = new Box();

This should call default constructor **Box()** in class Box.

But Box class has constructor but no default constructor is there.

So ERROR occurs

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

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

Box mybox1 = new Box();
}
}
```

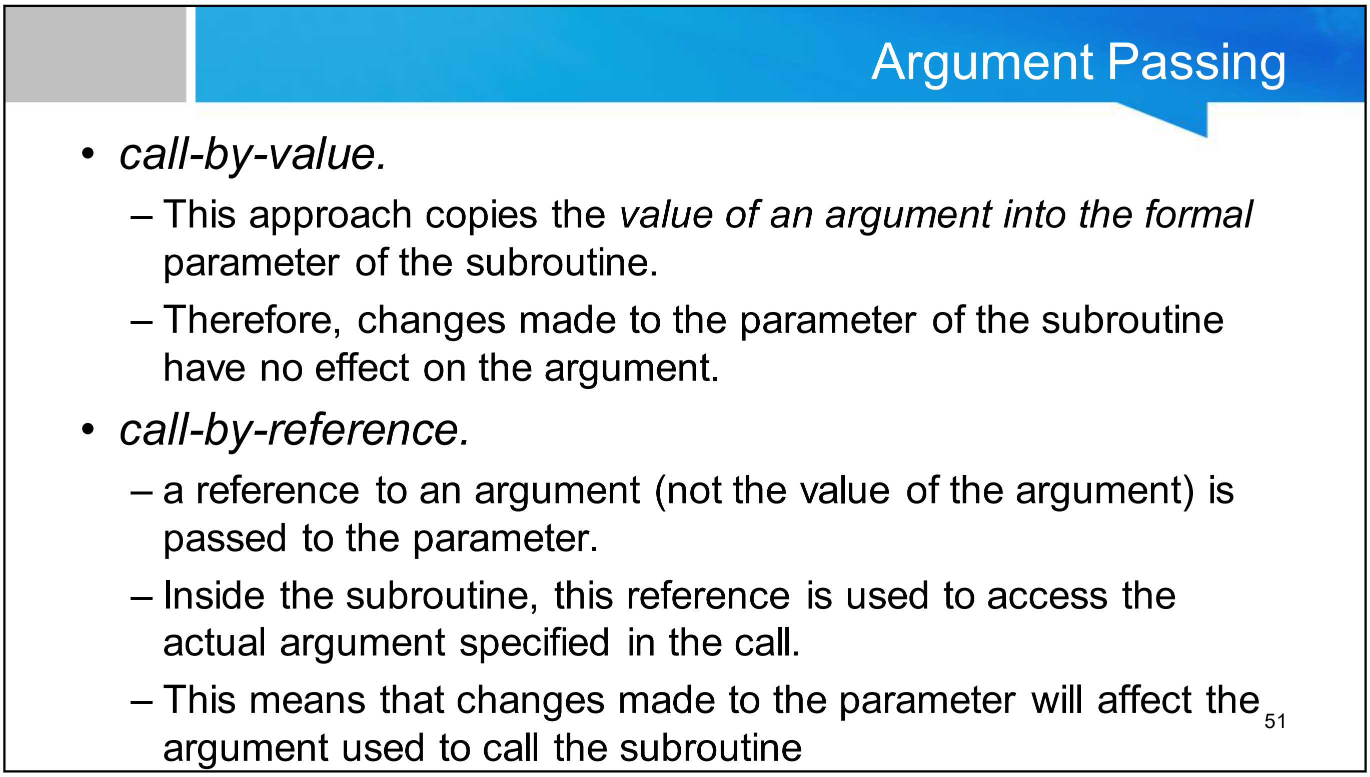
NO ERROR in this code

The following statement creates object of Box class mybox1

Box mybox1 = new Box();

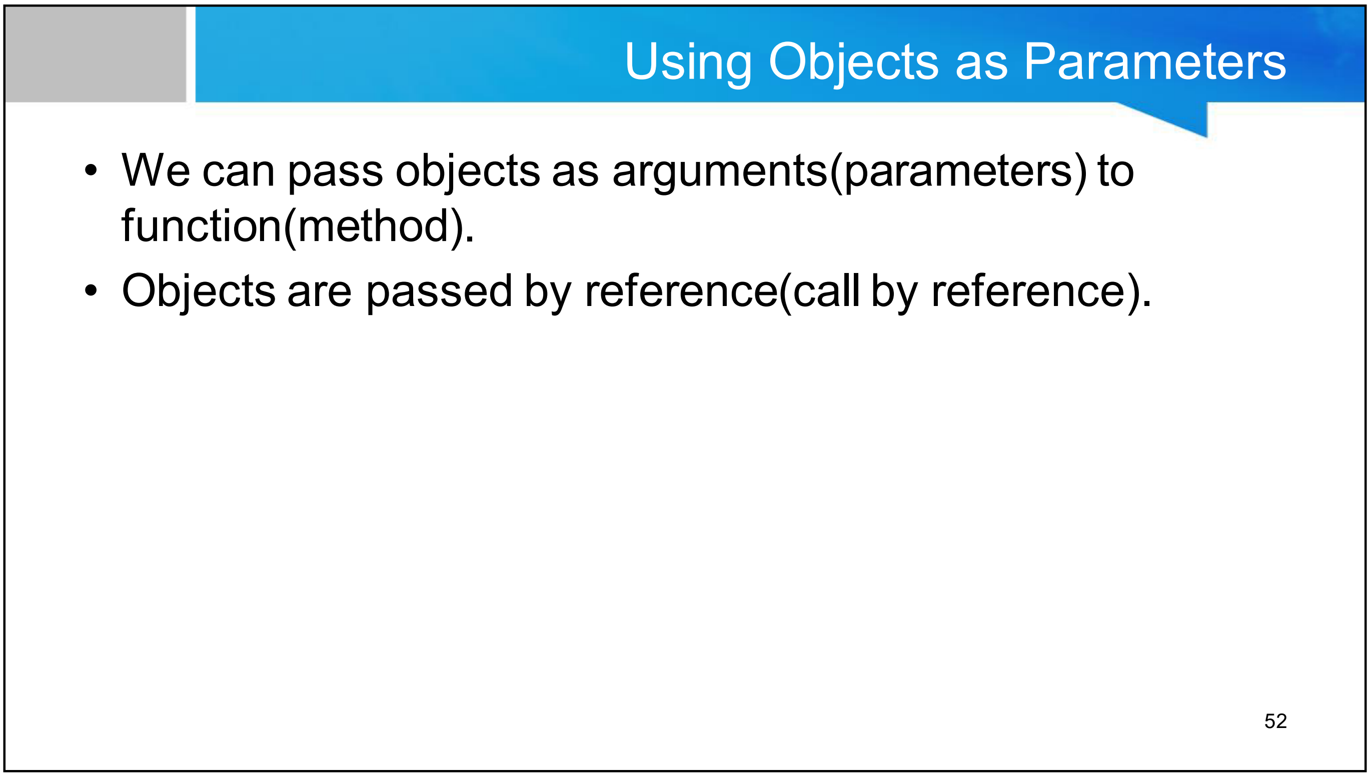
Since no constructors are not there,

Java provides the default constructor.



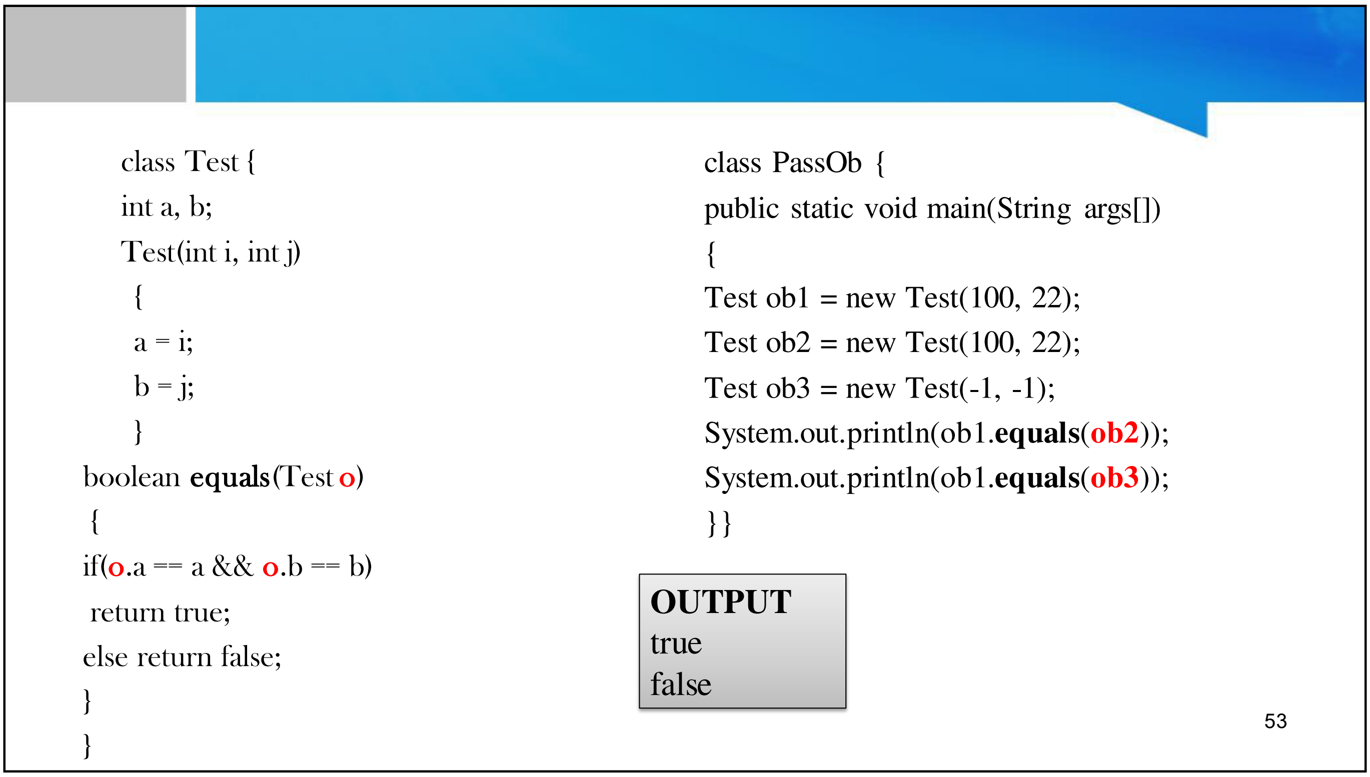
Argument Passing

- *call-by-value.*
  - This 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.
- *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



Using Objects as Parameters

- We can pass objects as arguments(parameters) to function(method).
- Objects are passed by reference(call by reference).

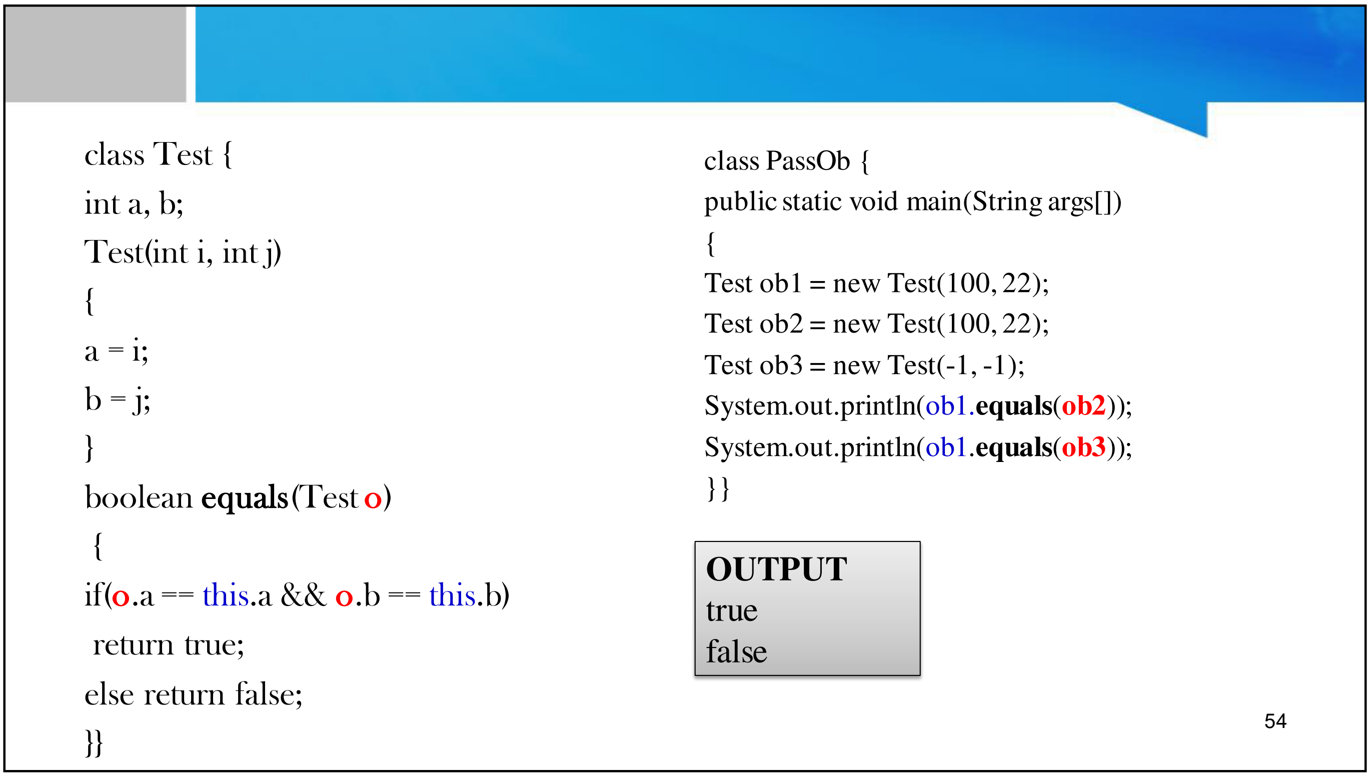


```
    class Test {
    int a, b;
    Test(int i, int j)
     {
     a = i;
     b = j;
     }
boolean equals(Test o)
 {
if(o.a == a && o.b == b)
 return true;
else return false;
}
}
```

```
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.equals(ob2));
System.out.println(ob1.equals(ob3));
}}
```

OUTPUT

```
true
false
```



```
class Test {
int a, b;
Test(int i, int j)
{
a = i;
b = j;
}
boolean equals(Test o)
 {
if(o.a == this.a && o.b == this.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.equals(ob2));
System.out.println(ob1.equals(ob3));
}}
```

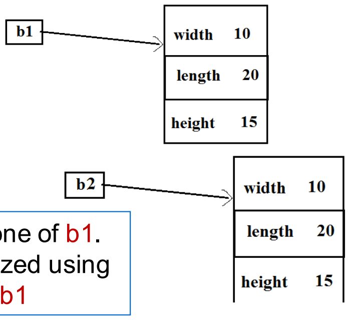
OUTPUT

```
true
false
```

Object to initialize another object

```
class Box
{
double width
double length;
double height;
Box(double w, double l, double h)
{
width = w;
length = l;
height = h;
}
}
```

```
class BoxDemo {
public static void main(String args[])
{
Box b1 = new Box(10, 20, 15);
Box b2 = new Box(b1);
}
}
```



Here object b2 is a clone of b1. The object b2 is initialized using initial values of object b1

Passing arguments to function

- Primitive types(int,char,double etc.) are passed by value.
- Objects are **passed by reference.**

```
class Test {
int a;
Test(int i)
{
a = i;
}
void calc(Test o)
{
o.a *= 2;
}
void calc(int a)
{
a*=2;
}
}
```

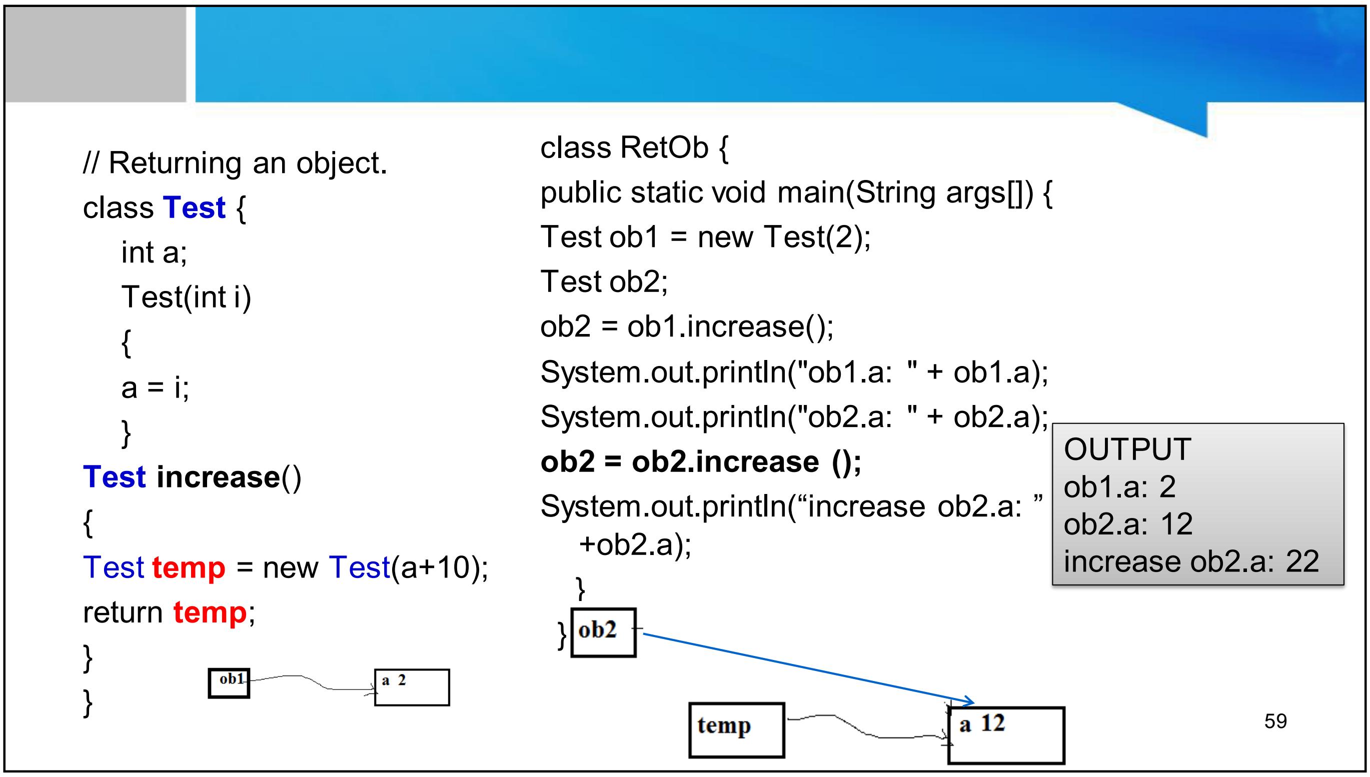
```
class Obcall {
public static void main(String args[])
{
Test ob = new Test(15);
System.out.println("Object parameter");
System.out.println("Before call: " + ob.a );
ob.calc(ob);// //Call by reference
System.out.println("After call: " + ob.a );
int a=15;
System.out.println("Integer parameter");
System.out.println("Before call: " + a);
ob.calc(a); //Call by value
System.out.println("After call: " + a);
} }
```

OUTPUT

```
Object parameter
Before call: 15
After call: 30
Integer parameter
Before call: 15
After call: 15
```

Returning objects

- A method can return any type of data,
  - Primitive data (int ,float, char, double etc.)
  - class types(objects) that you create.
  - etc.

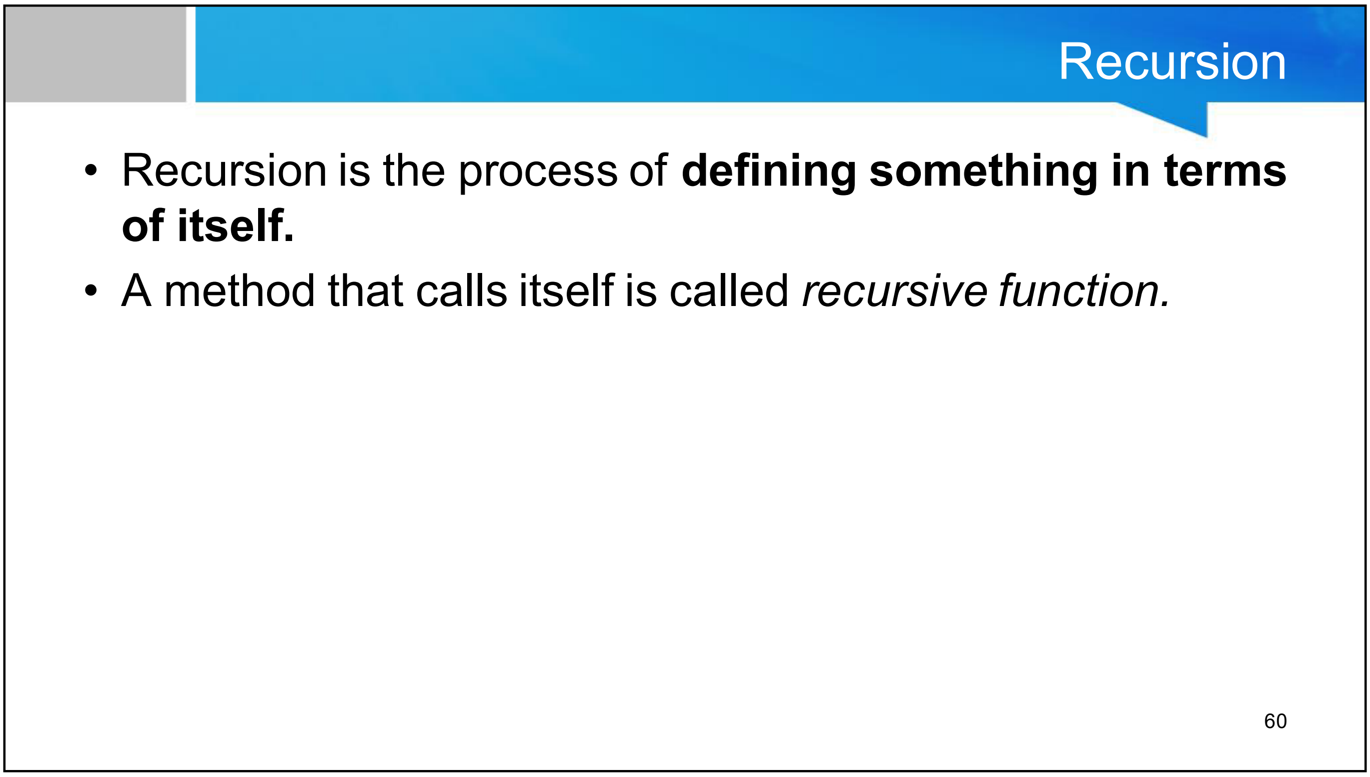


```
// Returning an object.
class Test {
    int a;
    Test(int i)
    {
    a = i;
    }
Test increase()
{
Test temp = new Test(a+10);
return temp;
}
}
```

```
class RetOb {
public static void main(String args[]) {
Test ob1 = new Test(2);
Test ob2;
ob2 = ob1.increase();
System.out.println("ob1.a:  " + ob1.a);
System.out.println("ob2.a:  " + ob2.a);
ob2 = ob2.increase ();
System.out.println(“increase ob2.a: ”
   +ob2.a);
  }
}
```

```
OUTPUT
ob1.a: 2
ob2.a: 12
increase ob2.a: 22
```

ob1
a 2
ob2
temp
a 12

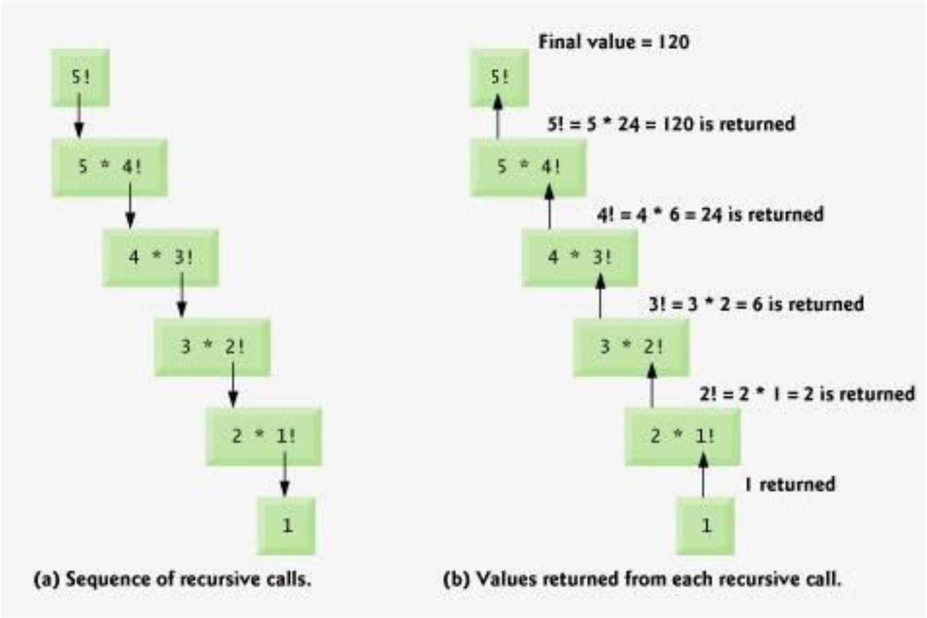


Recursion

- Recursion is the process of **defining something in terms of itself.**
- A method that calls itself is called *recursive function.*

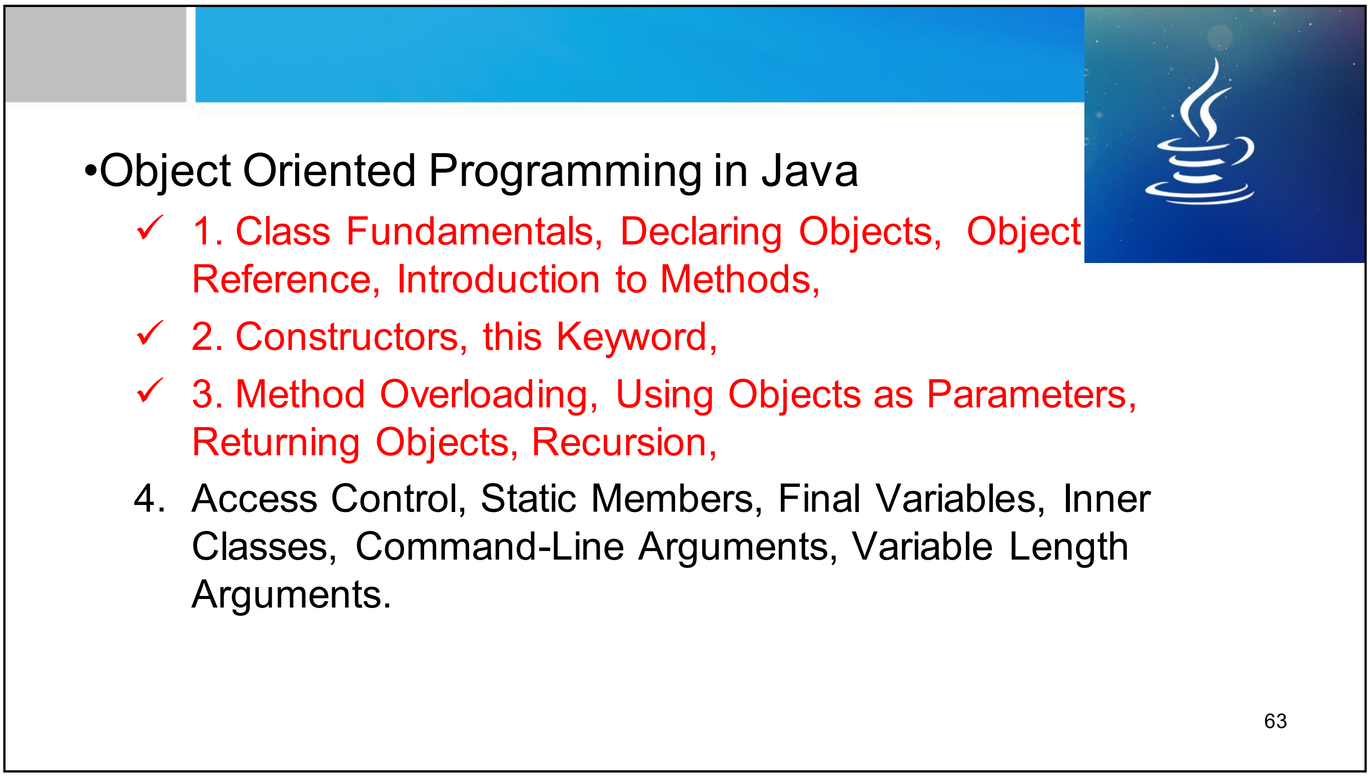
```
// A simple example of recursion.
class Factorial {
    int fact(int n)
     {
        int result;
        if(n==1)
          return 1;
        result = n* fact(n-1) ;
   return result;
}
}
```

```
class Recursion {
public static void main(String
   args[]) {
Factorial f = new Factorial();
int s= f.fact(5)
System.out.println("Factorial of 5
   is " + s);
}
}
```

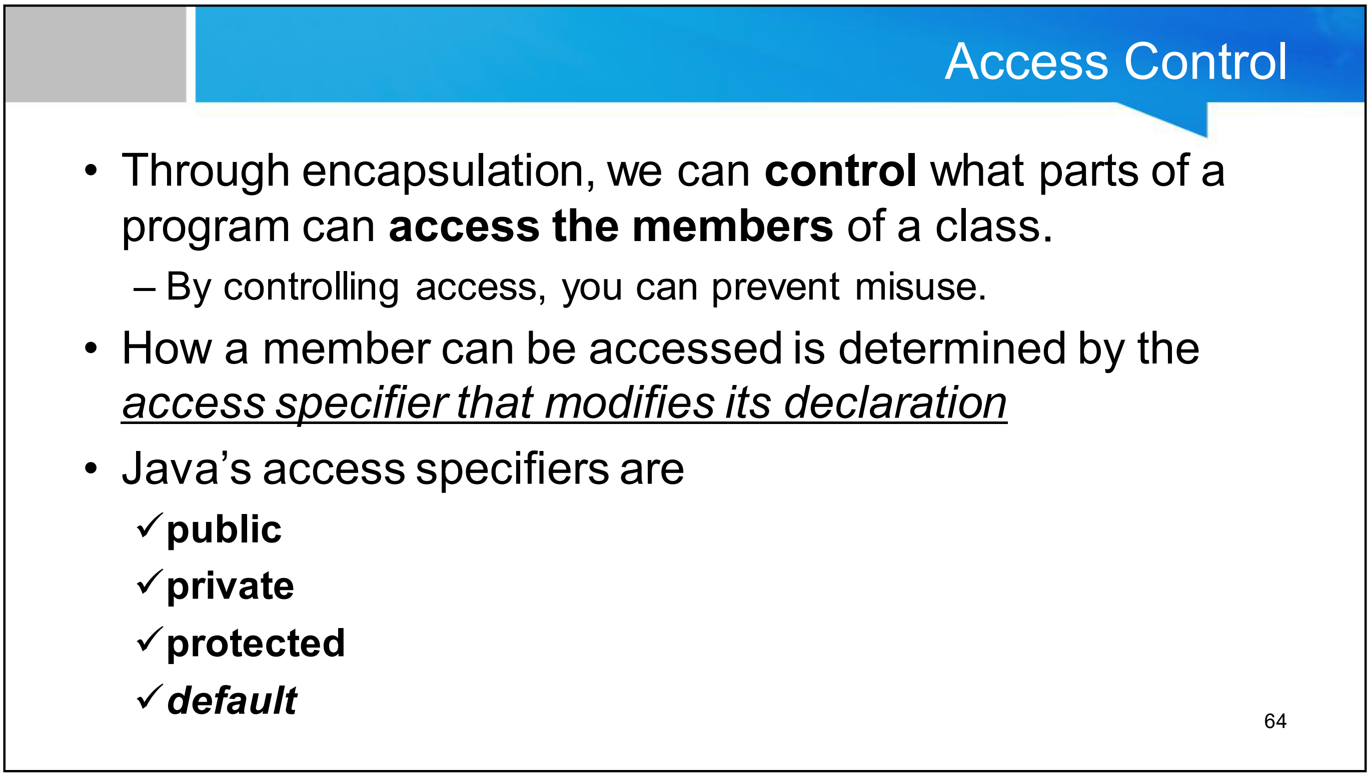

5!
5 * 4!
4 * 3!
3 * 2!
2 * 1!
1
Final value = 120
5!
5! = 5 * 24 = 120 is returned
5 * 4!
4! = 4 * 6 = 24 is returned
4 * 3!
3! = 3 * 2 = 6 is returned
3 * 2!
2! = 2 * 1 = 2 is returned
2 * 1!
1 returned
1

(a) Sequence of recursive calls.

(b) Values returned from each recursive call.



- Object Oriented Programming in Java
  - ✓ 1. Class Fundamentals, Declaring Objects, Object Reference, Introduction to Methods,
  - ✓ 2. Constructors, this Keyword,
  - ✓ 3. Method Overloading, Using Objects as Parameters, Returning Objects, Recursion,
  - 4. Access Control, Static Members, Final Variables, Inner Classes, Command-Line Arguments, Variable Length Arguments.



Access Control

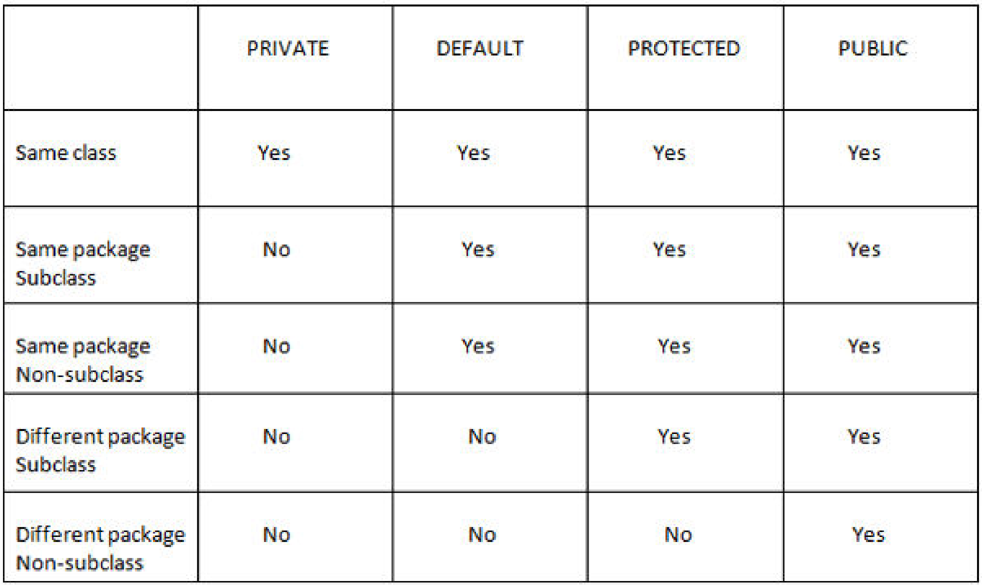
- Through encapsulation, we can **control** what parts of a program can **access the members** of a class.
  - – By controlling access, you can prevent misuse.
- How a member can be accessed is determined by the *access specifier that modifies its declaration*
- Java’s access specifiers are
  - ✓**public**
  - ✓**private**
  - ✓**protected**
  - ✓***default***

- When a member of a class is modified by the public specifier, then that member can be accessed by any other code. (ACCESSIBLE TO ALL)
  - public int i;
- When a member of a class is specified as private, then that member can only be accessed by any members of the same class.
  - private int a;

- When a member of a class is specified as protected , then that member can be accessed within the package and by any of its subclasses.
  protected char c;
- When no access specifier is there, then its access specifier is default.
  - It can be accessed within its own package, but cannot be accessed outside of its package

  int c;

```
class A{
    public int i;
    private double j;
    protected char c;
    float f;//default access
    public int myMethod(int a, char b) //public method
    {   //..
    }
}
```



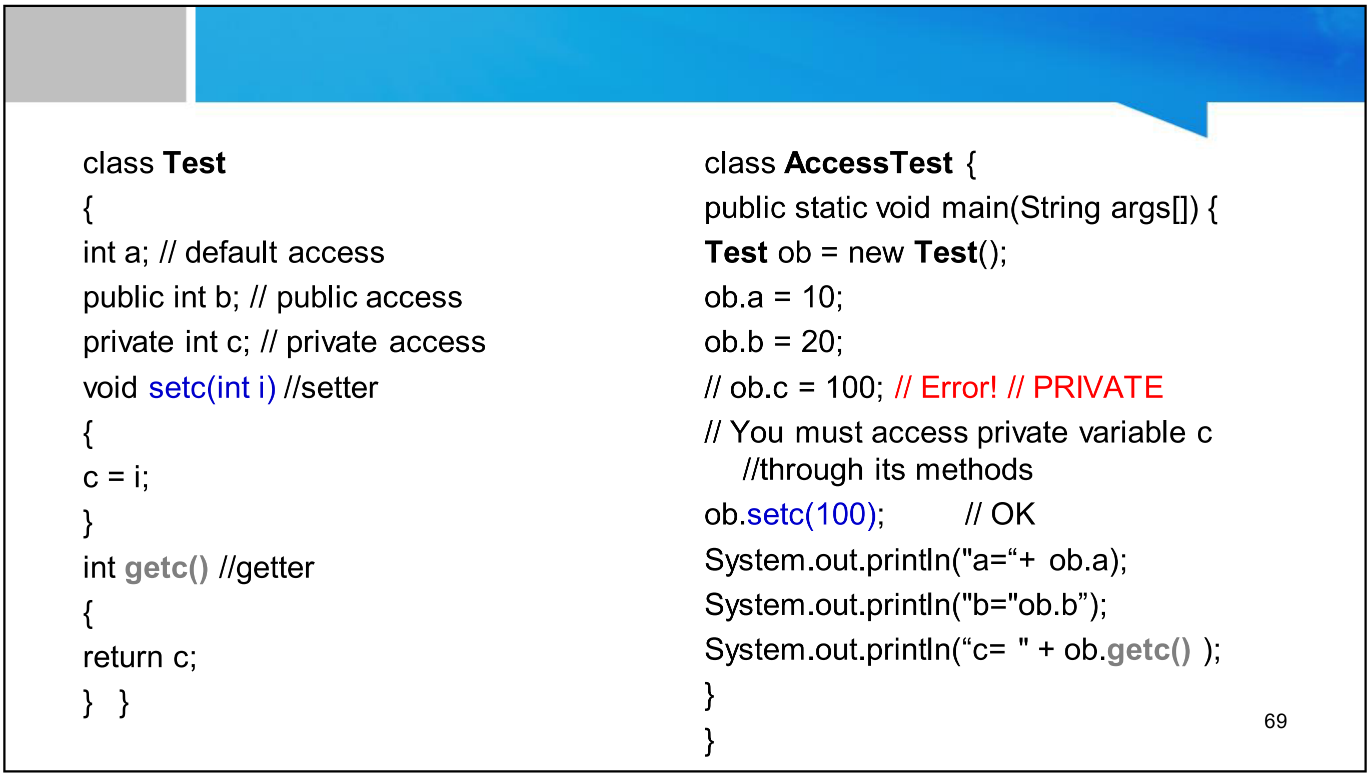
	PRIVATE	DEFAULT	PROTECTED	PUBLIC
Same class	Yes	Yes	Yes	Yes
Same package Subclass	No	Yes	Yes	Yes
Same package Non-subclass	No	Yes	Yes	Yes
Different package Subclass	No	No	Yes	Yes
Different package Non-subclass	No	No	No	Yes

SAME CLASS

SAME PACKAGE,

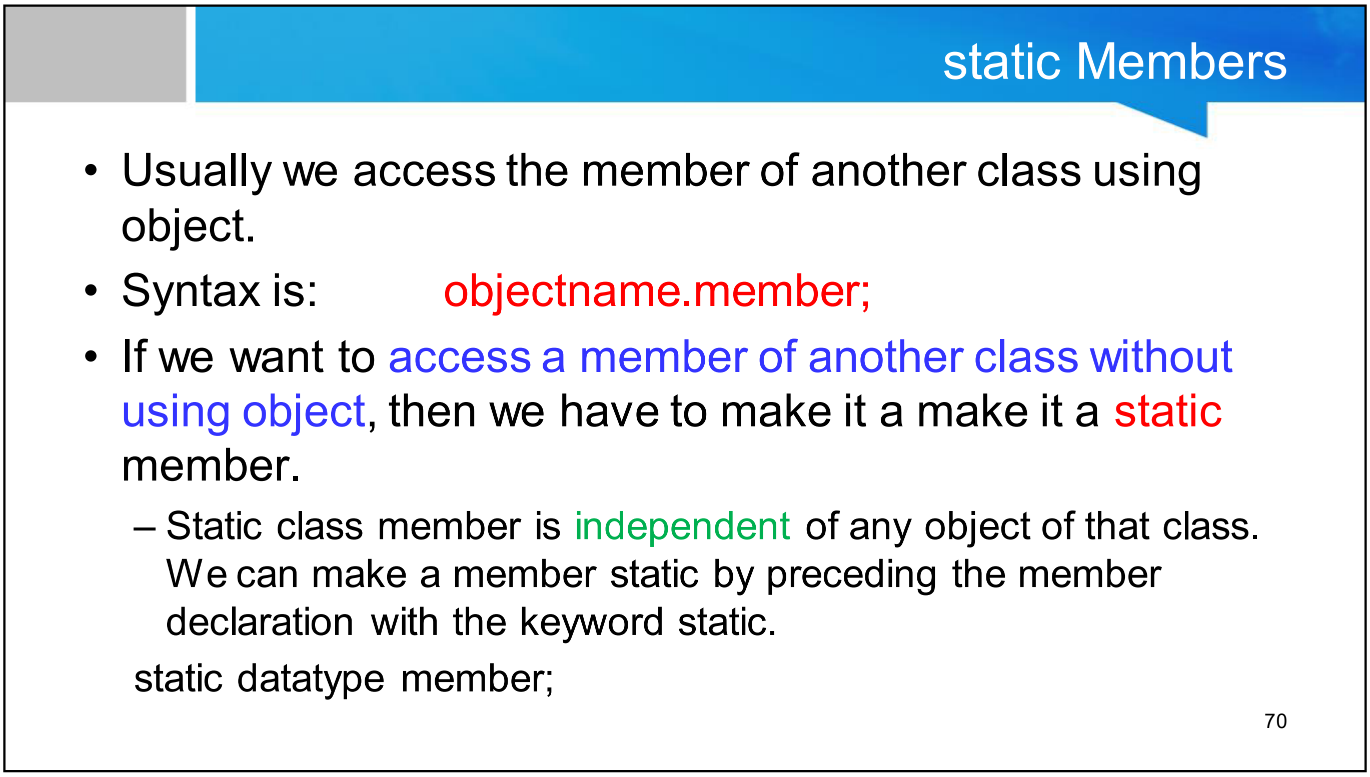
SAME PACKAGE
,
ANY SUBCLASS

ALL



```
class Test
{
int a; // default access
public int b; // public access
private int c; // private access
void setc(int i) //setter
{
c = i;
}
int getc() //getter
{
return c;
}  }
```

```
class AccessTest {
public static void main(String args[]) {
Test ob = new Test();
ob.a = 10;
ob.b = 20;
// ob.c = 100; // Error! // PRIVATE
// You must access private variable c
    //through its methods
ob.setc(100);        // OK
System.out.println("a="+ ob.a);
System.out.println("b="ob.b");
System.out.println("c= " + ob.getc() );
}
}
```



static Members

- Usually we access the member of another class using object.
- Syntax is: objectname.member;
- If we want to access a member of another class without using object, then we have to make it a make it a static member.
  - Static class member is independent of any object of that class. We can make a member static by preceding the member declaration with the keyword static.

static datatype member;

- When a member is declared **static,** it can be accessed before any objects of its class are created, and without reference to any object.
- Static member can be accessed using

```
classname.member;
```

- The most common example of a static member is main function.
  - main( ) is declared as static because it must be called before any objects is created.
- Instance variables declared as static are global variables.
- When objects of its class are declared, separate copy of a static variable is NOT made.
- All instances(objects) of the class share the same static variable.

- Methods declared as static(static methods) have several restrictions:
  - static methods can only call other static methods.
  - static methods must only access static data.
  - static methods cannot refer to this or super.
- If we need to do computation to initialize your static variables, we can declare a static block that gets executed exactly once, when the class is first loaded.

- // Demonstrate static variables, methods, and blocks.

```
class UseStatic {
static int a = 3;
static int b;
static void show(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[])
{show(42);
}
}
```

OUTPUT

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

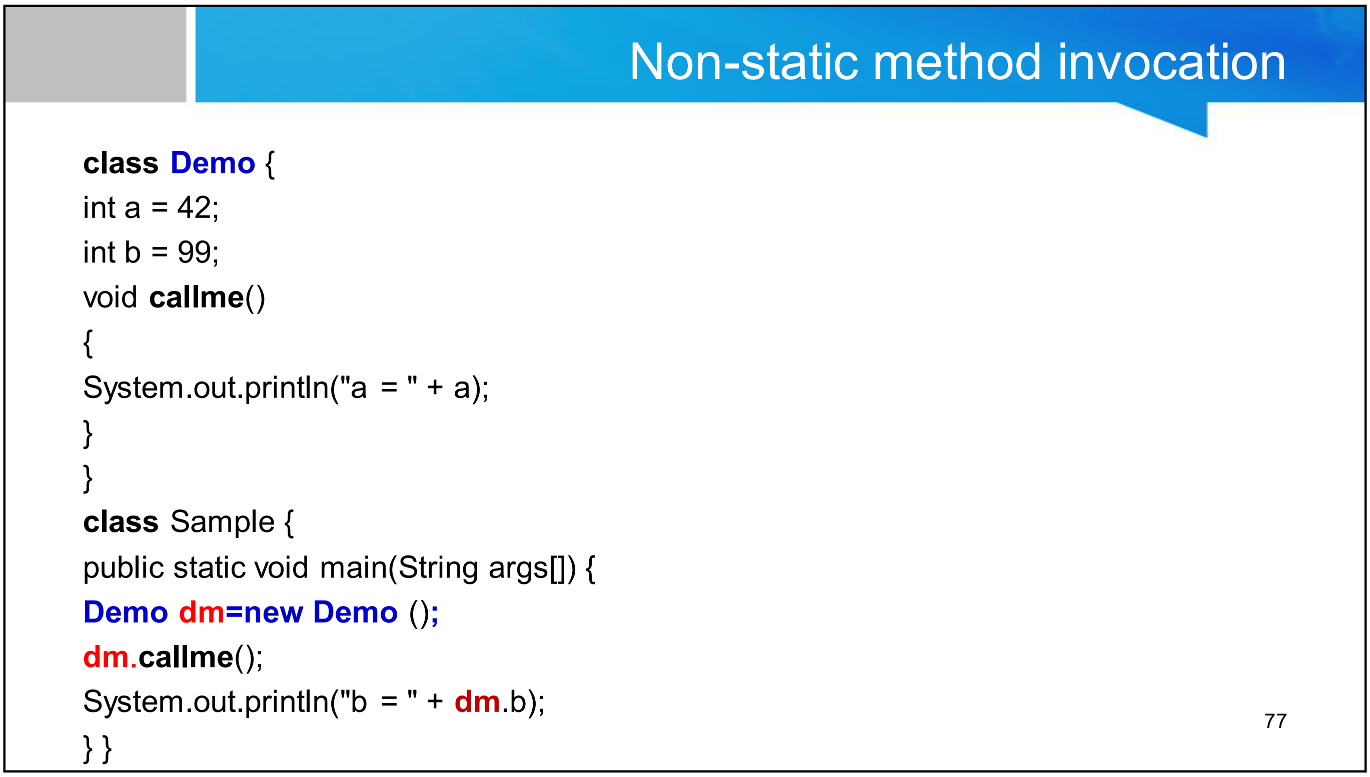
Working of e.g. code

- As soon as the UseStatic class is loaded, all of the static statements are run.
  - First, static member 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 show( ), passing 42 to x.
  - The three println( ) statements in show refer to the two static variables a and b, as well as to the local variable x.

- if we want to call a static method from outside its class, we can do so using the following general form:

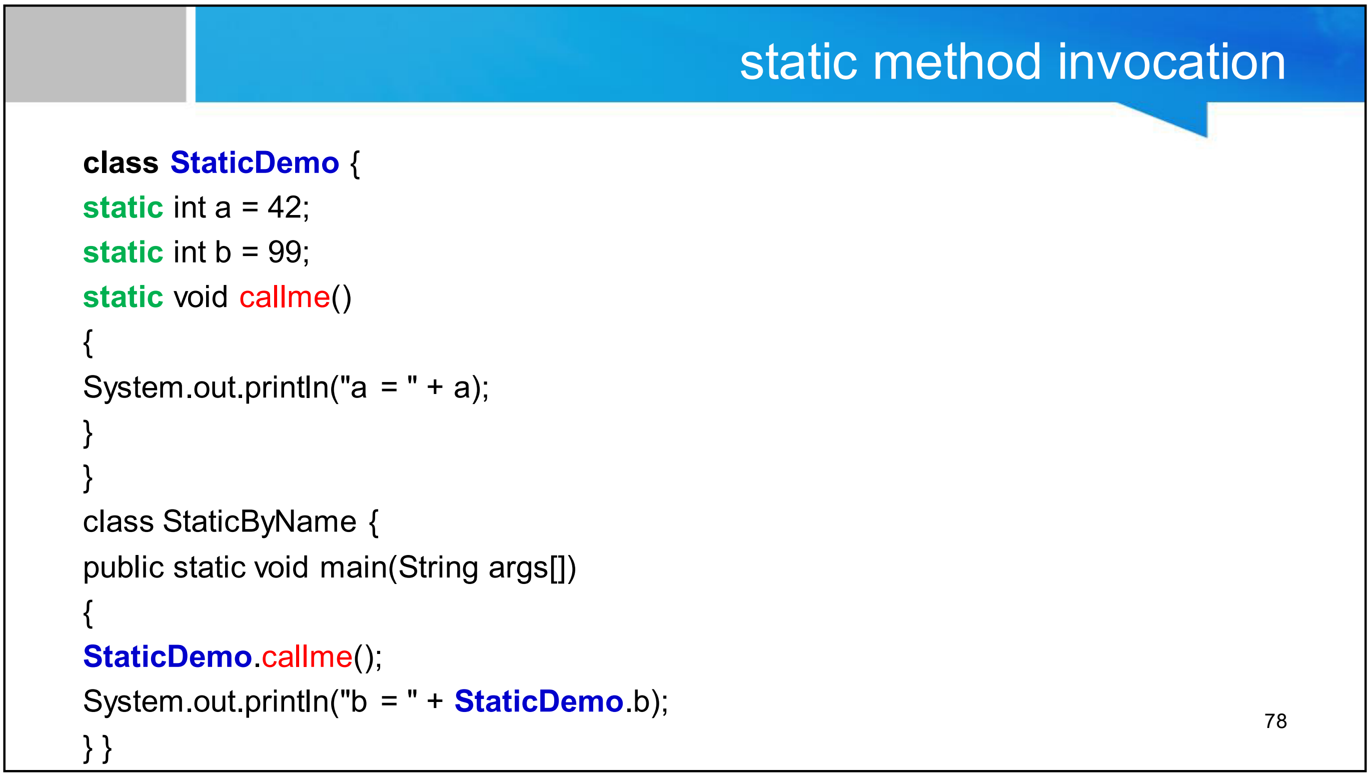
classname.method( );

- Here classname is the name of the class in which the static method is declared.



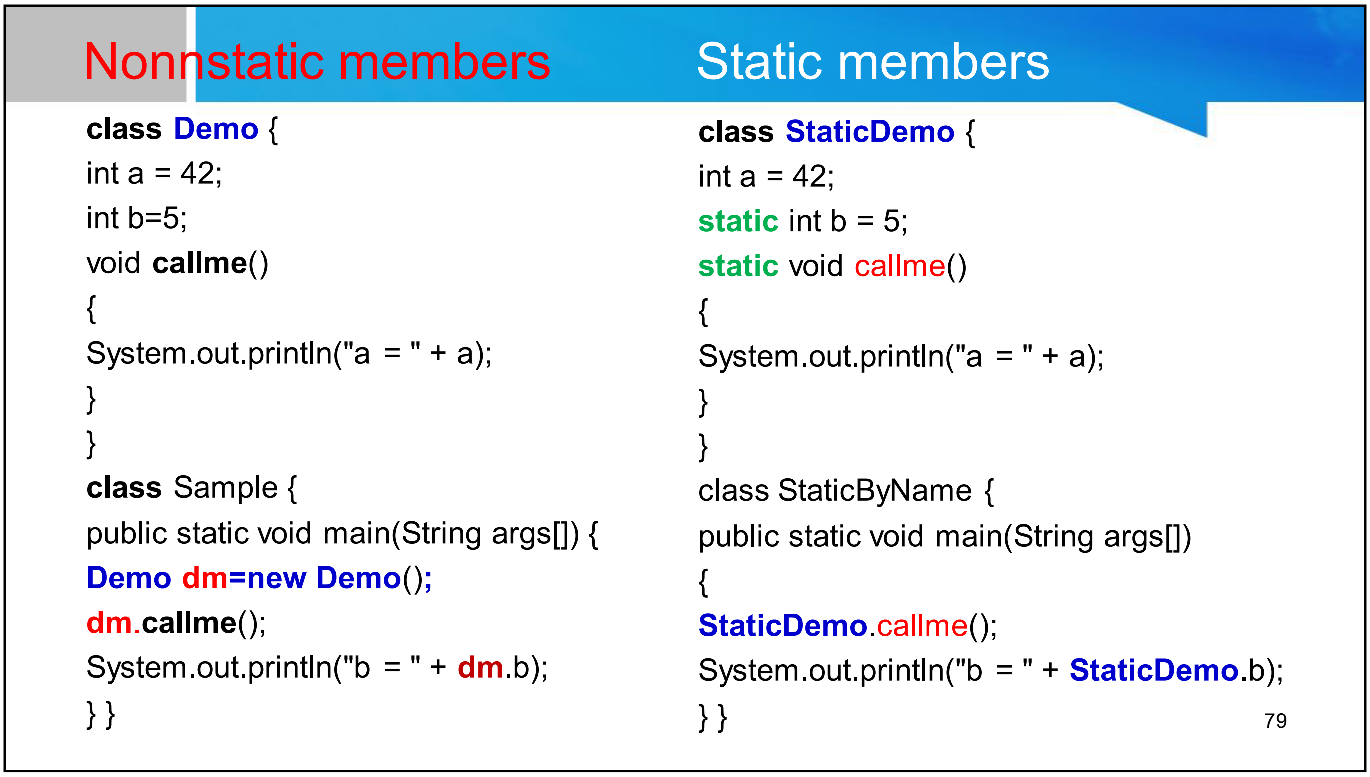
Non-static method invocation

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



static method invocation

```
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);
} }
```

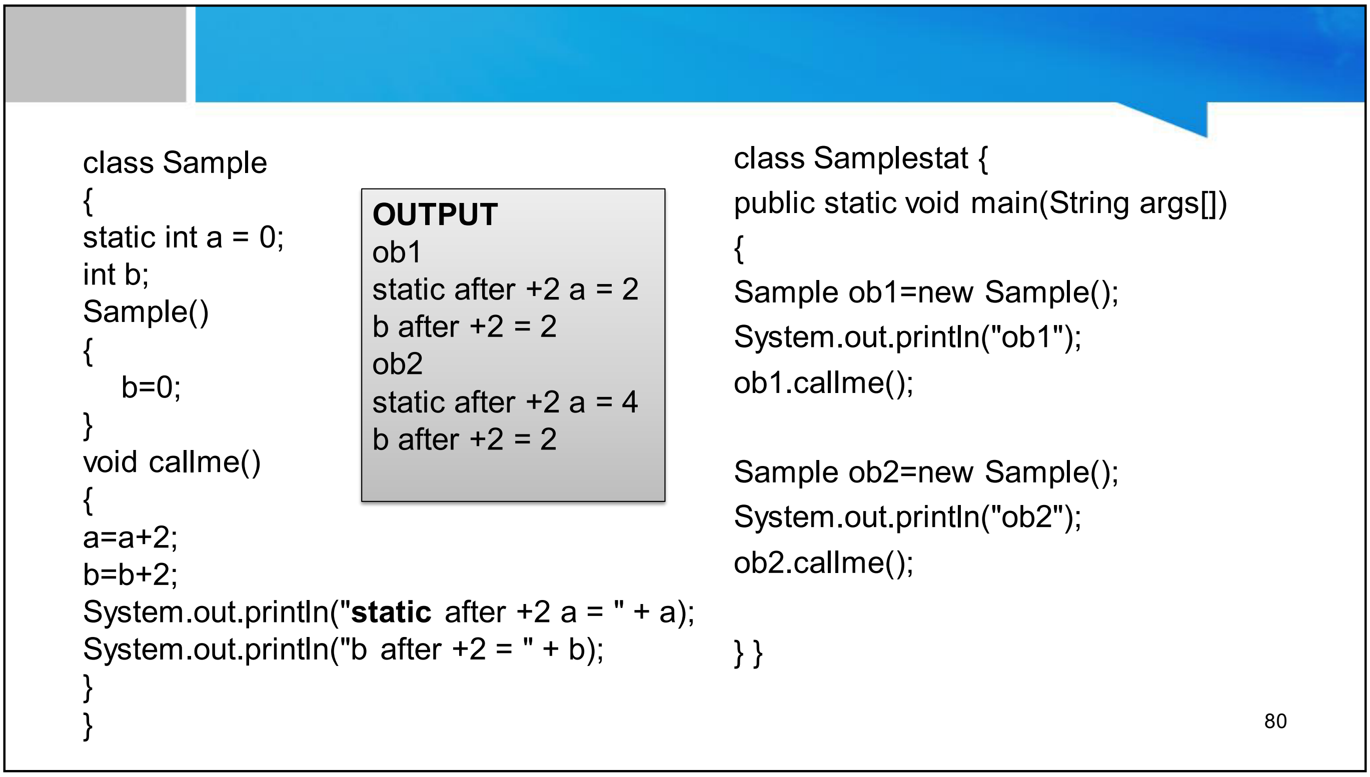


Nonstatic members

```
class Demo {
int a = 42;
int b=5;
void callme()
{
System.out.println("a  = " + a);
}
}
class Sample {
public static void main(String args[]) {
Demo dm=new Demo();
dm.callme();
System.out.println("b  = " + dm.b);
} }
```

Static members

```
class StaticDemo {
int a = 42;
static int b = 5;
static void callme()
{
System.out.println("a  = " + a);
}
}
class StaticByName {
public static void main(String args[])
{
StaticDemo.callme();
System.out.println("b  = " + StaticDemo.b);
} }
```

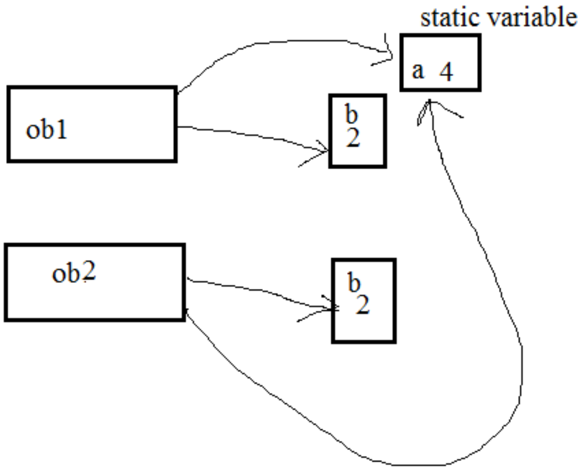


```
class Sample
{
static int a = 0;
int b;
Sample()
{
    b=0;
}
void callme()
{
a=a+2;
b=b+2;
System.out.println("static  after +2 a = " + a);
System.out.println("b  after +2 = " + b);
}
}
```

OUTPUT

```
ob1
static after +2 a = 2
b after +2 = 2
ob2
static after +2 a = 4
b after +2 = 2
```

```
class Samplestat {
public static void main(String args[])
{
Sample ob1=new Sample();
System.out.println("ob1");
ob1.callme();

Sample ob2=new Sample();
System.out.println("ob2");
ob2.callme();

} }
```



Final Variables

- A variable can be declared as final by prefixing final keyword.
- The contents of final variables cannot be modified.
- We must initialize a final variable when it is declared.

E.g.

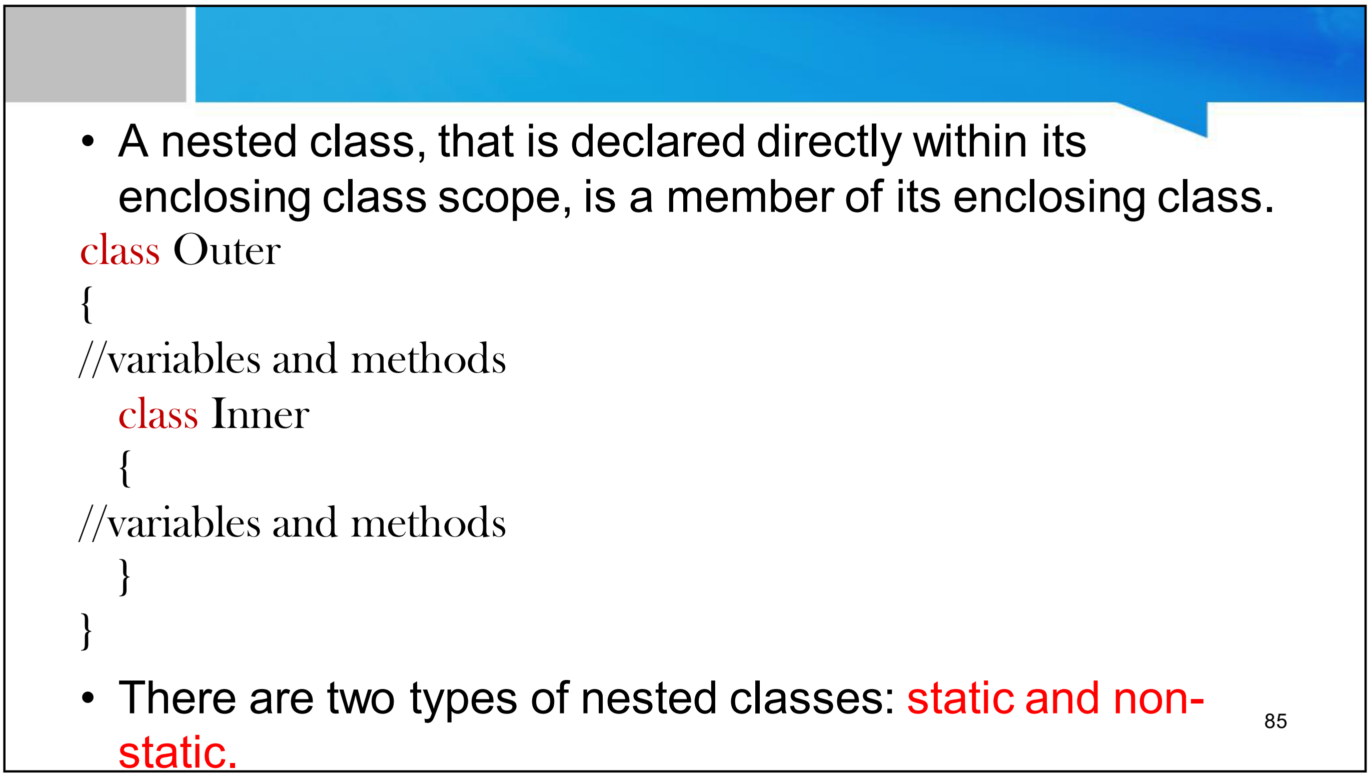
```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
```

- It is a convention to choose uppercase identifiers (CAPITAL LETTERS) for final variables. E.g. TOTAL
- We can use final variables as if they were constants, without fear that a value has been changed.
- Variables declared as final do not occupy memory on a per-instance basis.

Nested Class In Java

Nested Classes

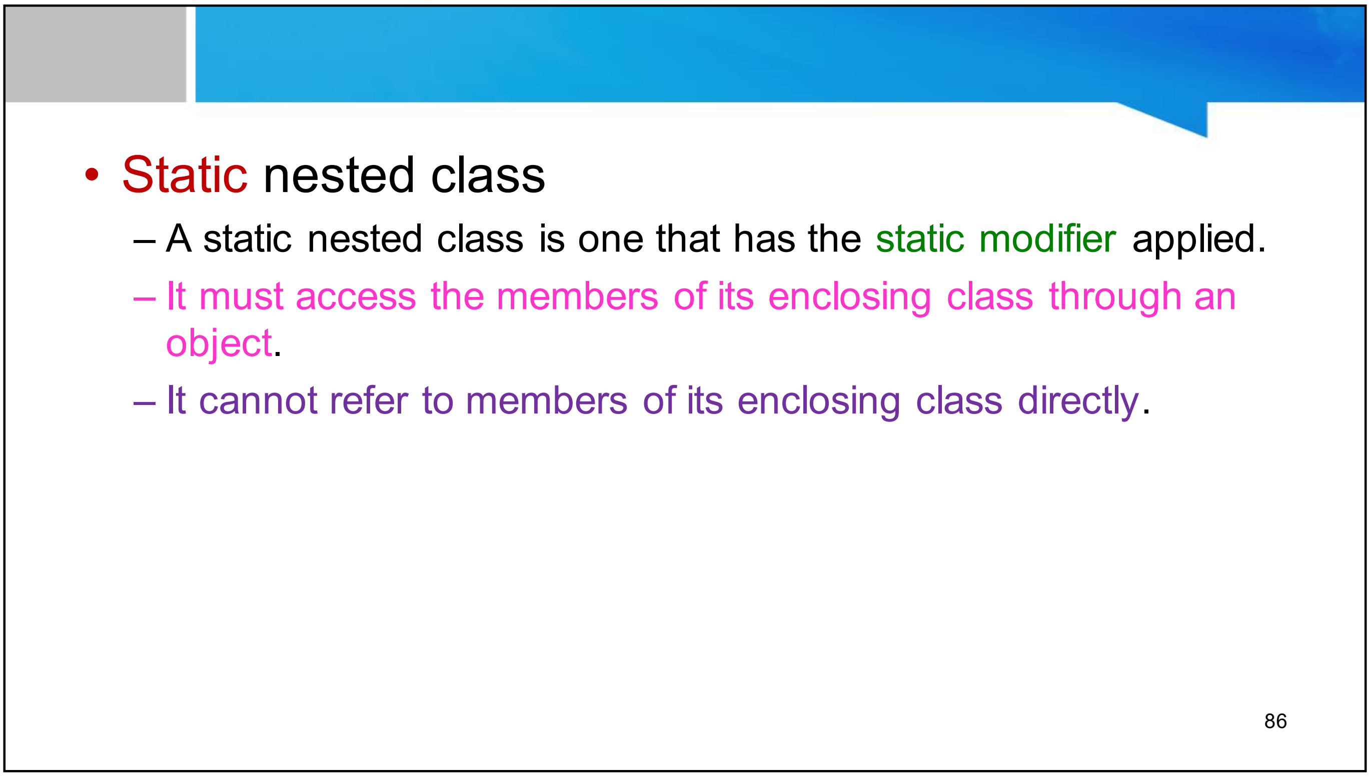
- It is possible to define a class within another class; such classes are known as nested classes.
- The scope of a nested class is bounded by the scope of its enclosing class(outer).
  - Thus, if class B is defined within class A, then B does not exist independently of A.
- A nested class has access to the members, including private members, of the enclosing(outer) class.
- 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.

```
class Outer
{
//variables and methods
   class Inner
   {
//variables and methods
   }
}
```

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



- Static nested class
  - A static nested class is one that has the static modifier applied.
  - It must access the members of its enclosing class through an object.
  - It cannot refer to members of its enclosing class directly.

- // Demonstrate a STATIC inner class.

```
class Outer
{
    int outer_x = 100;
    void test() {
        Nested nested= new Nested ();
        nested.display();
    }
    static class Nested {      //static nested class
      void display() {
          Outer obj = new Outer();
           System.out.println("display:  outer_x = " + obj.outer_x);
           }
      }
}
class NestedClassDemo {
    public static void main(String args[]) {
    Outer outer = new Outer();
    outer.test();
} }
```

```
OUTPUT
display:  outer_x = 100
```

Inner Class

- Non static class
  - A non-static nested class is called inner class.
  - An inner class has access to all of the variables and methods of its outer class.
  - It may refer to members of its enclosing class directly in the same way that other non-static members of the outer class do.

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

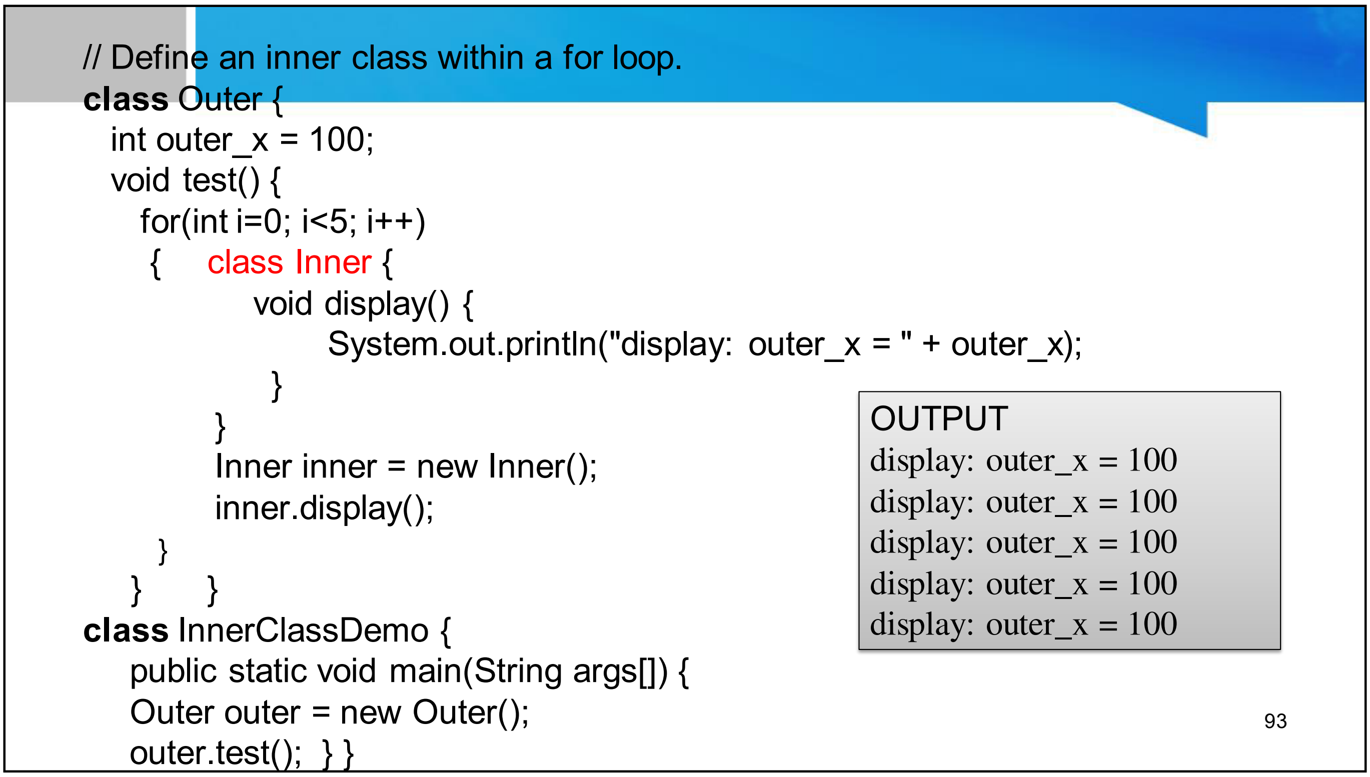
```
OUTPUT
display:  outer_x = 100
```

- In the program, an inner class named Inner is defined within the scope of class Outer.
- Therefore, any code in class Inner can directly access the variable outer_x in Outer class.
- An instance method named display( ) is defined inside Inner.
  - This method displays outer_x on the standard output stream.
- The main( ) method of InnerClassDemo creates an instance of class Outer and invokes its test( ) method.
- That method creates an instance of class Inner and the display() method is called.

- An instance(object) of Inner can be created only within the scope of class Outer.
- We can create an instance of Inner class outside of Outer class by qualifying its name with Outer classname, as in

  Outer.Inner ob=outerobject.new Inner();
- An inner class can access all of the members of its enclosing class, but the reverse is not true.
- Members of the inner class are known only within the scope of the inner class and may not be used by the outer class.

- We can define a nested class within the block defined by a method or even within the body of a **for loop**



```
// Define an inner class within a for loop.
class Outer {
  int outer_x = 100;
  void test() {
    for(int i=0; i<5; 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();  } }
```

OUTPUT

```
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
```

Command-Line & Variable Length Arguments In Java

Command-Line Arguments

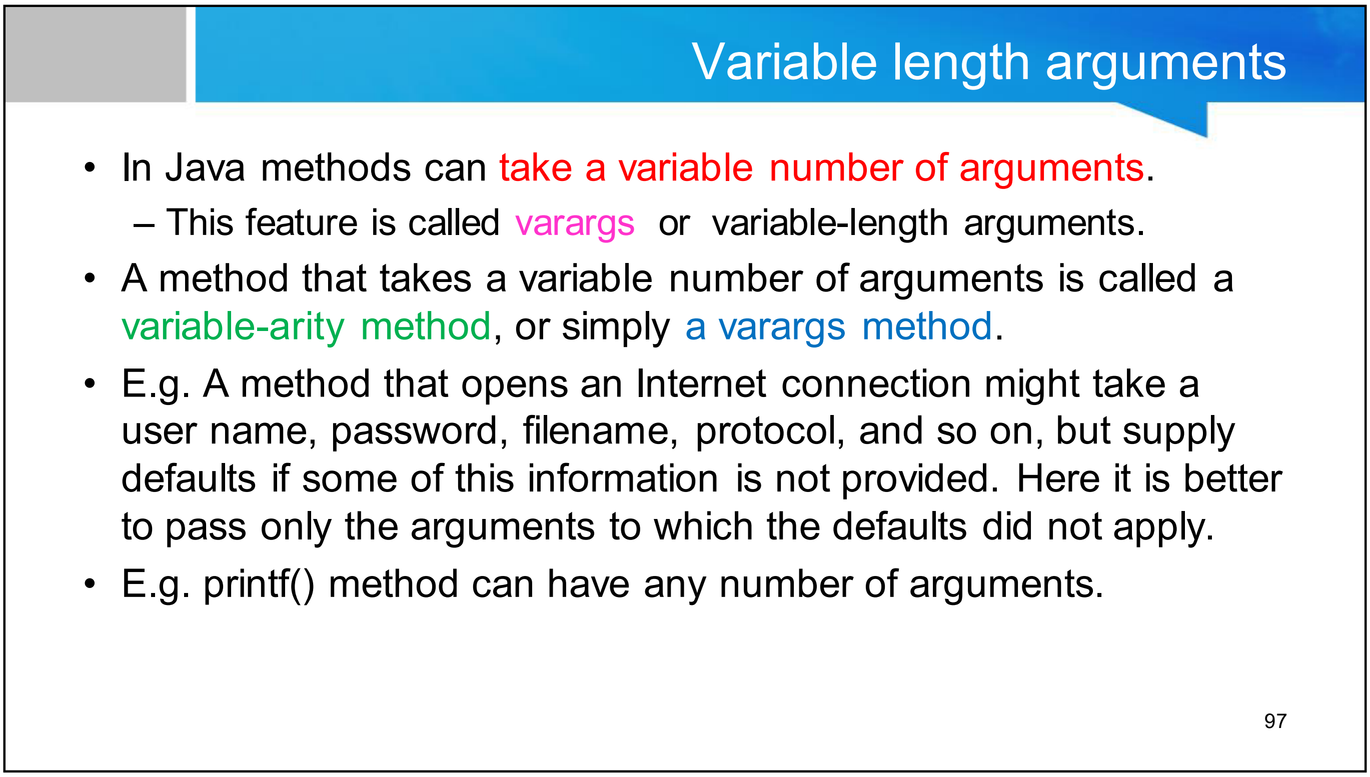
- If we want to pass information into a program when you run it, then you can do this by passing command-line arguments to main( ).
- A command-line argument is the information that follows program's name on the command line when it is executed.
- Command-line arguments are stored as strings in a String array passed to the args parameter of main( ).
  - The first command-line argument is stored at args[0]
  - the second at args[1]
  - so on.

```
// Display all 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]);
   }
}
```

```
Output
args[0]: this
args[1]: is
args[2]: a
args[3]: test
args[4]: 100
args[5]: -1
```

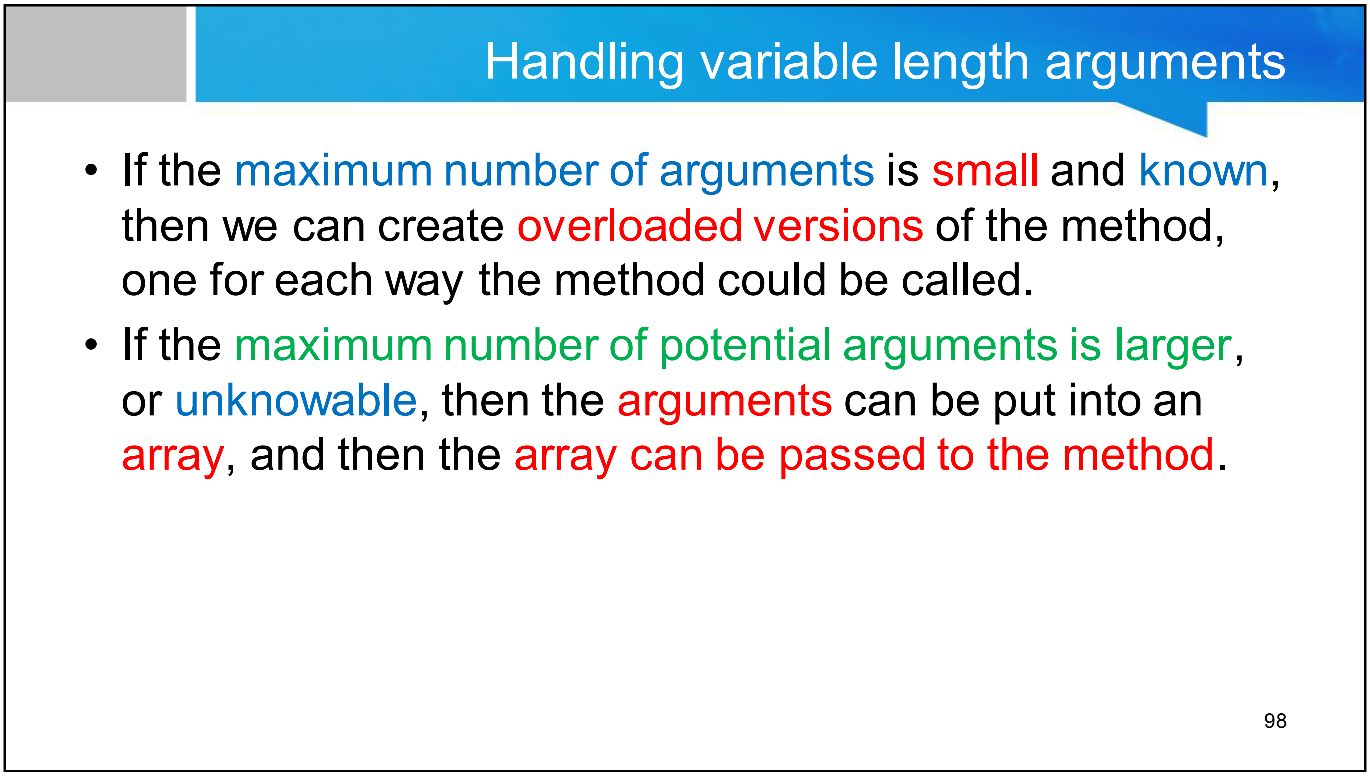
- Compile this using javac and execute this program as:-

  `java CommandLine this is a test 100 -1`



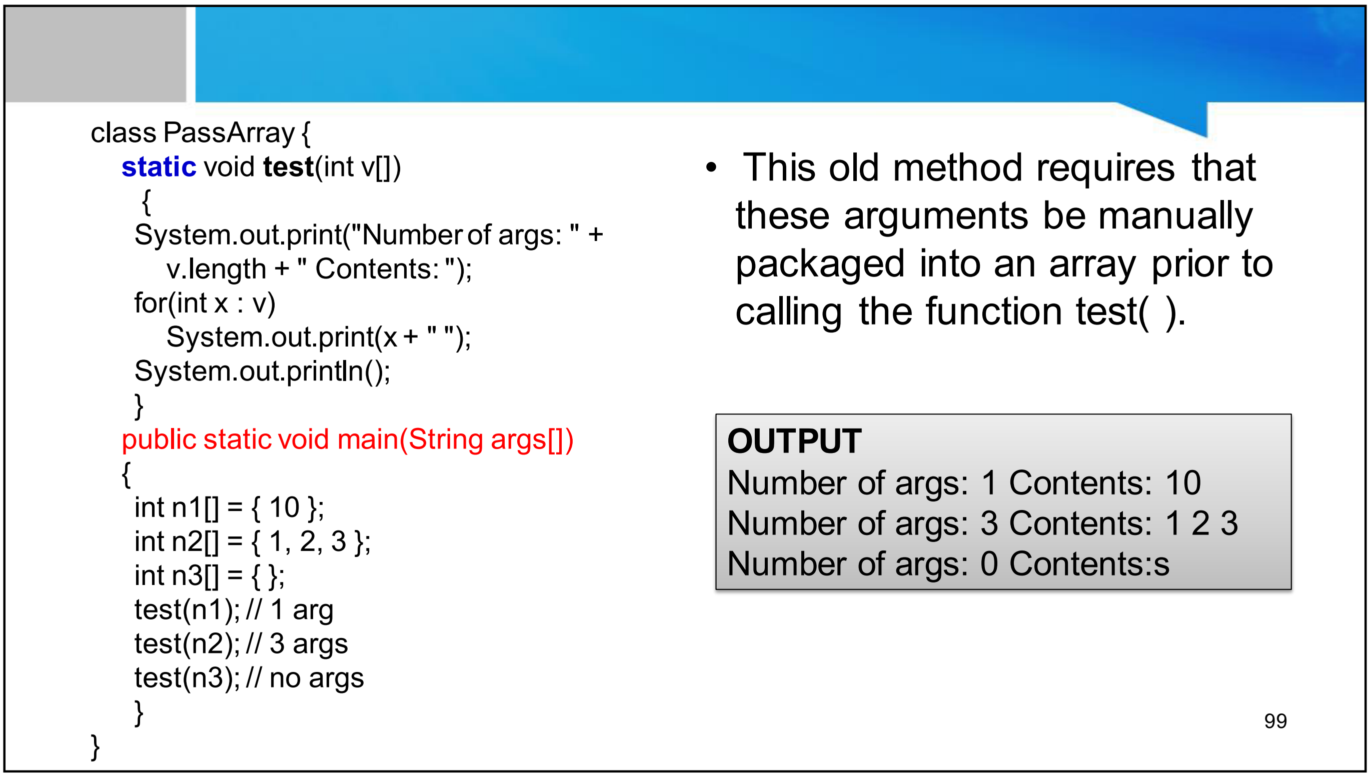
Variable length arguments

- In Java methods can take a variable number of arguments.
  - This feature is called varargs or variable-length arguments.
- A method that takes a variable number of arguments is called a variable-arity method, or simply a varargs method.
- E.g. 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. Here it is better to pass only the arguments to which the defaults did not apply.
- E.g. printf() method can have any number of arguments.



Handling variable length arguments

- If the maximum number of arguments is small and known, then we can create overloaded versions of the method, one for each way the method could be called.
- If the maximum number of potential arguments is larger, or unknowable, then the arguments can be put into an array, and then the array can be passed to the method.

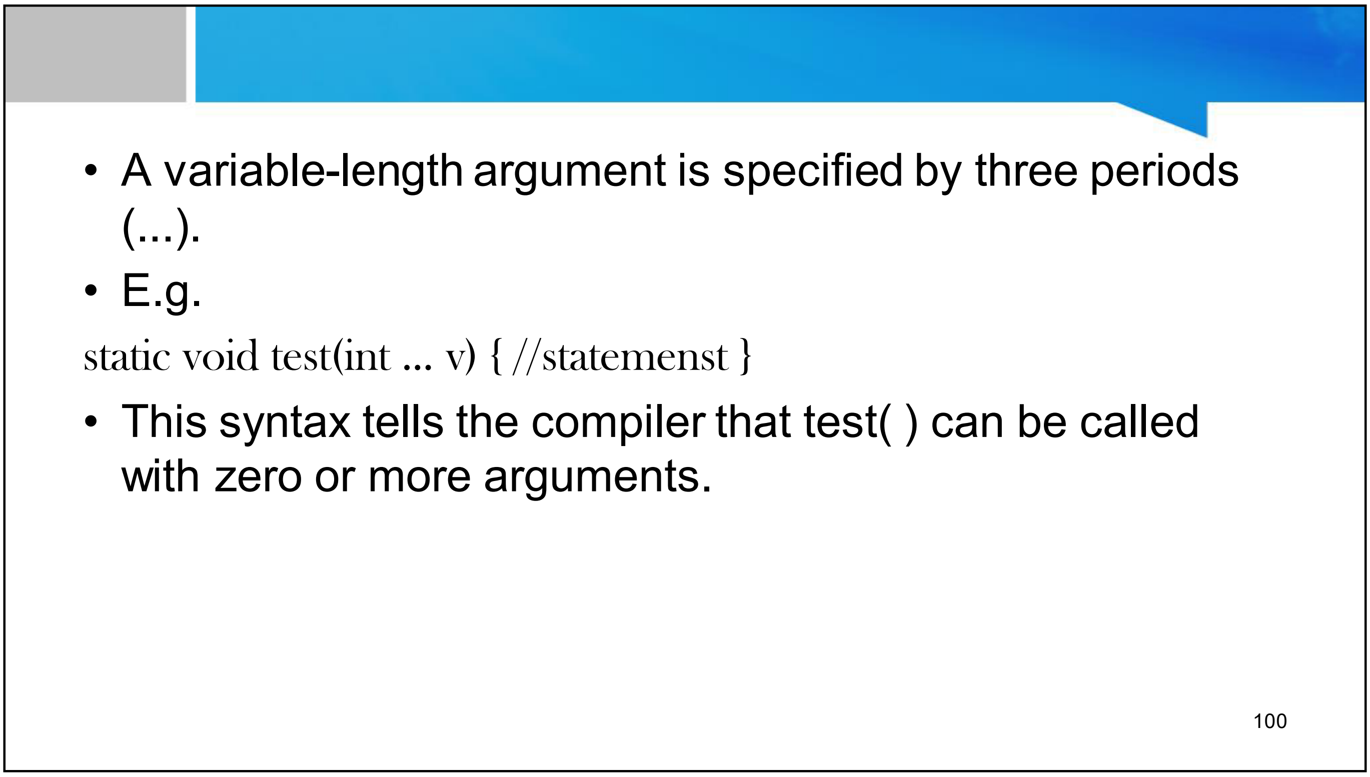


```
class PassArray {
   static void test(int v[])
    {
    System.out.print("Number of args: " +
       v.length + " Contents: ");
    for(int x : v)
       System.out.print(x + " ");
    System.out.println();
    }
   public static void main(String args[])
   {
    int n1[] = { 10 };
    int n2[] = { 1, 2, 3 };
    int n3[] = { };
    test(n1); // 1 arg
    test(n2); // 3 args
    test(n3); // no args
    }
}
```

- This old method requires that these arguments be manually packaged into an array prior to calling the function test( ).

OUTPUT

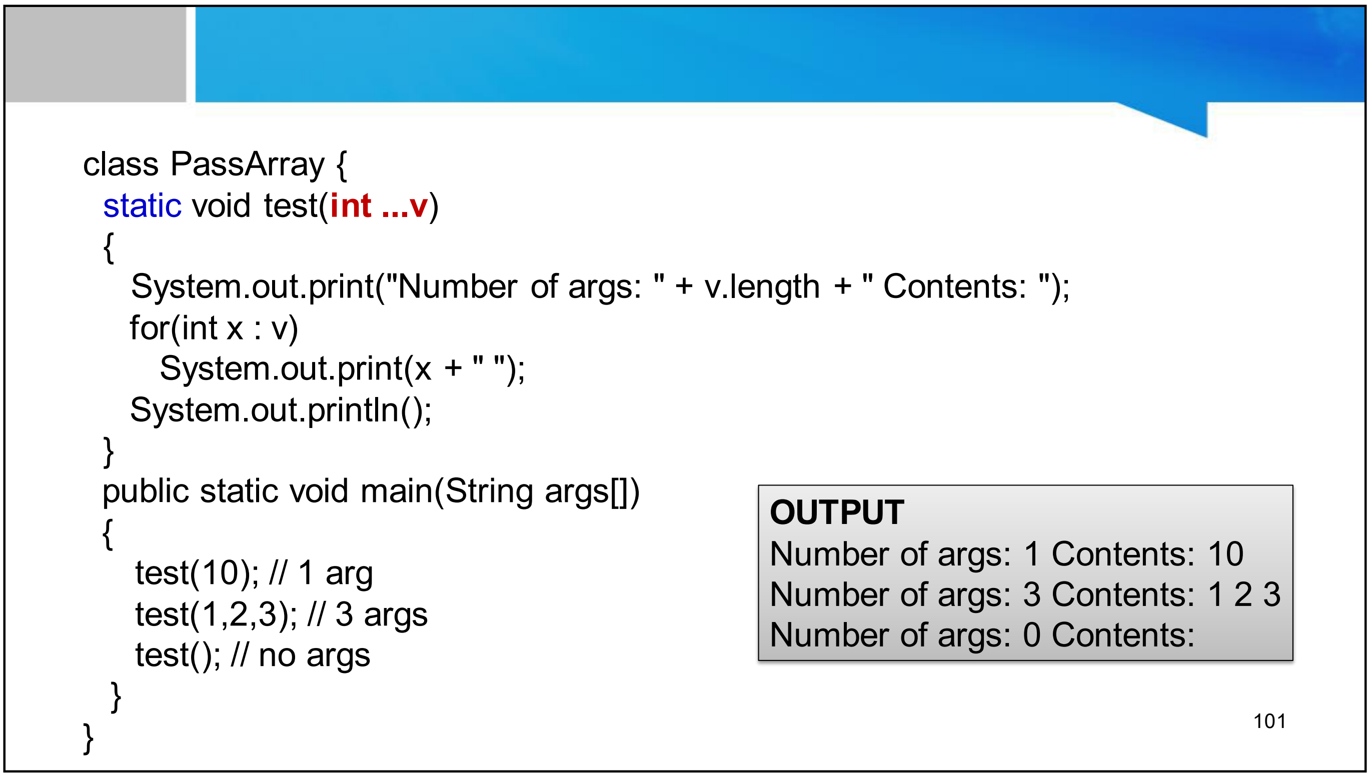
```
Number of args: 1 Contents: 10
Number of args: 3 Contents: 1 2 3
Number of args: 0 Contents:s
```



- A variable-length argument is specified by three periods (...).
- E.g.

static void test(int ... v) { //statemenst }

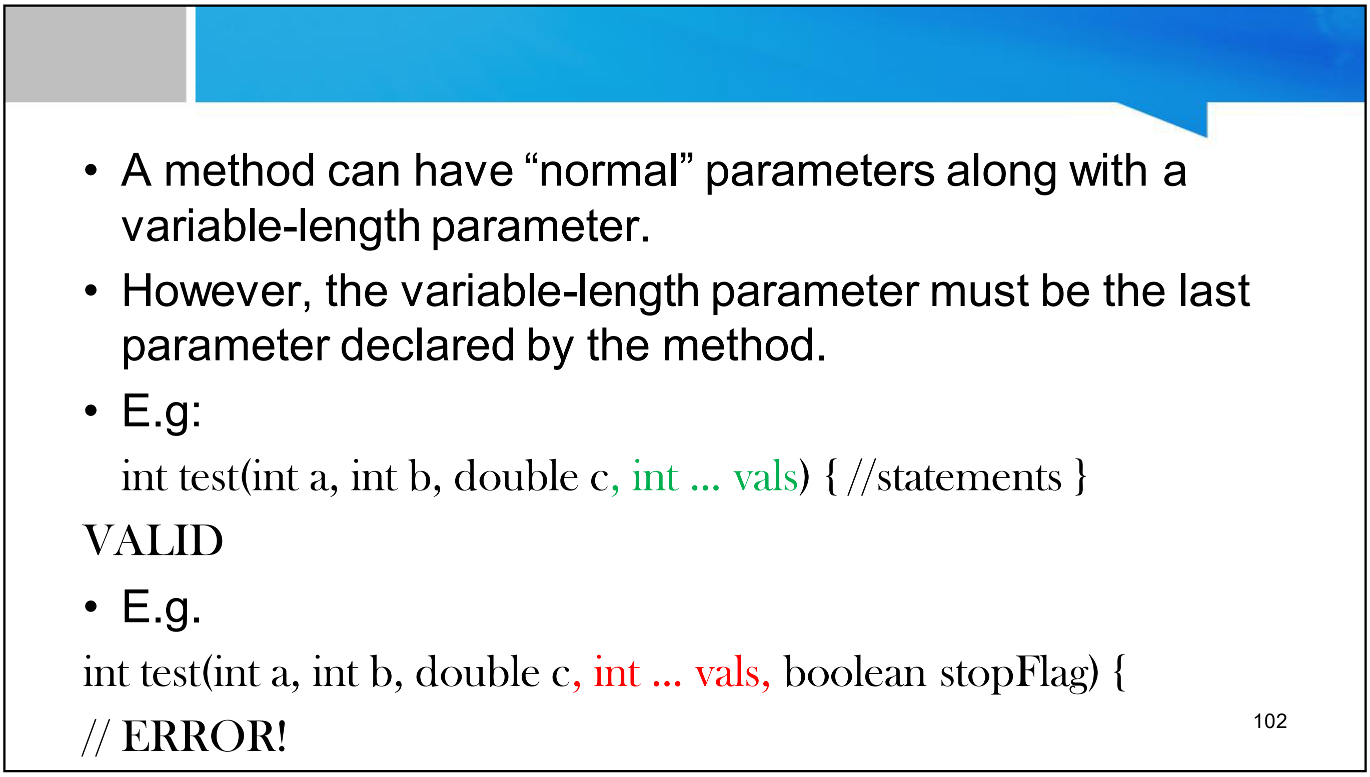
- This syntax tells the compiler that test( ) can be called with zero or more arguments.



```
class PassArray {
  static void test(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[])
  {
     test(10); // 1 arg
     test(1,2,3); // 3 args
     test(); // no args
   }
}
```

OUTPUT

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



- 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.
- E.g:

  int test(int a, int b, double c, int ... vals) { //statements }

VALID

- E.g.

int test(int a, int b, double c, int ... vals, boolean stopFlag) {

// ERROR!

MODULE 2 ENDS