A Closer Look at Methods and Classes (Chapter 7 of Schilit)

Object Oriented Programming BS (AI) II

By

Abdul Ghafoor

Revision of Methods/Functions

- What is a method/function?
- Parameter vs Argument
- Void?
  - Why do we use these methods

Polymorphism

OOP Principle

When an object of class shows multiple behaviors.

Overloading Methods

Two or more methods in same class:

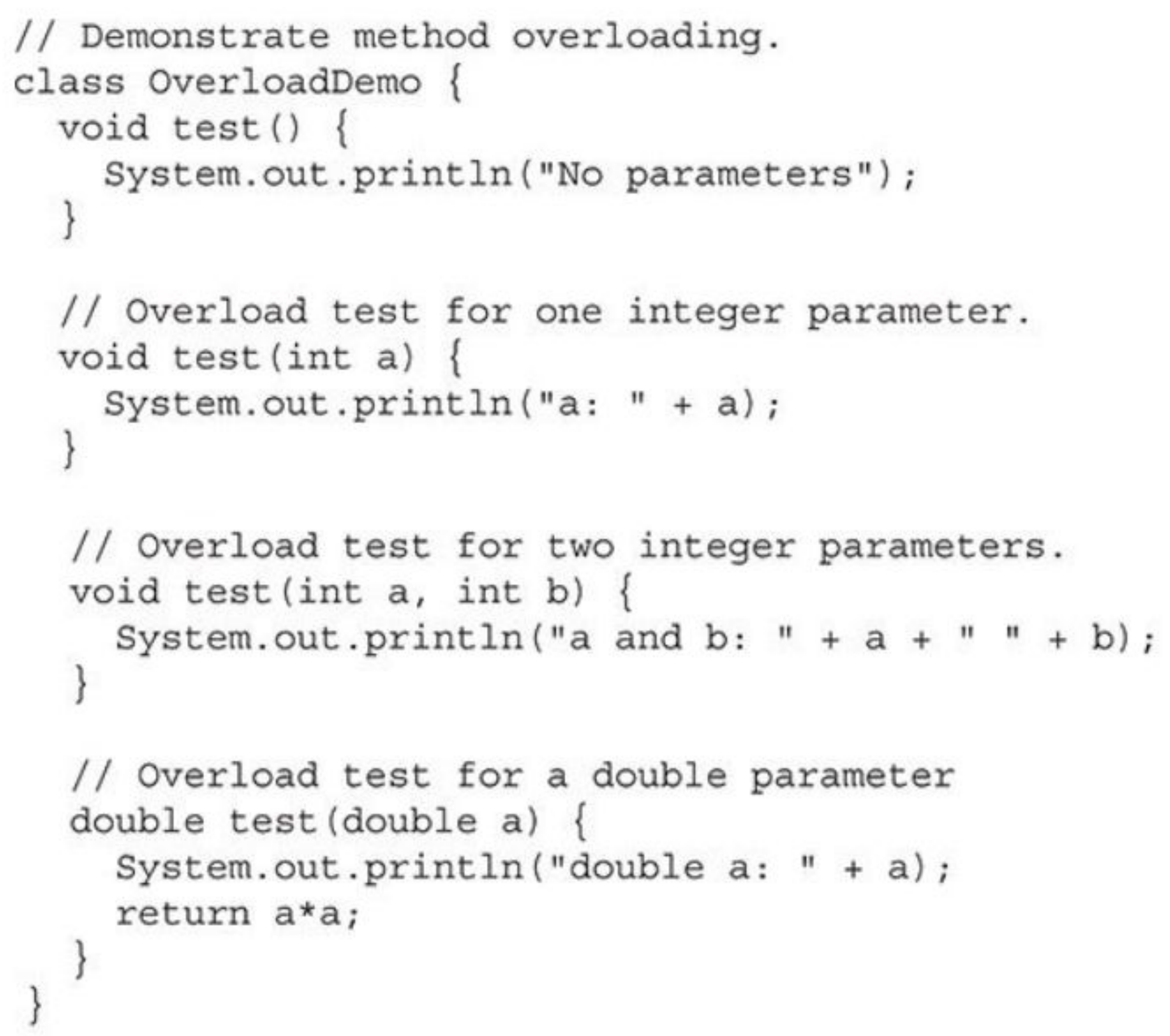
- Having same name
- Different type/number/Sequence of arguments

Type or number of argument determine the method to be called

Overloading Methods

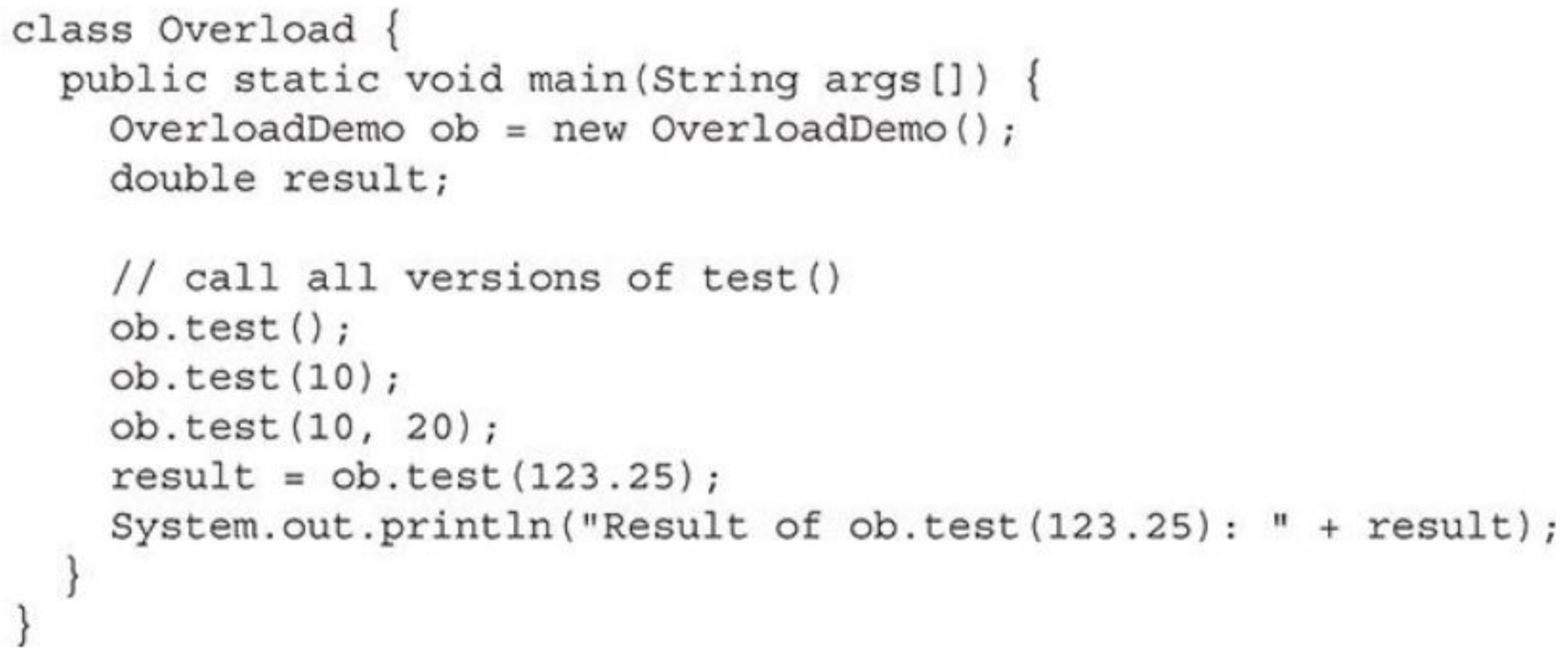
- Return type alone is not sufficient during the call of method
    - Java matches the arguments with the parameters to call a particular method

Overloading Methods



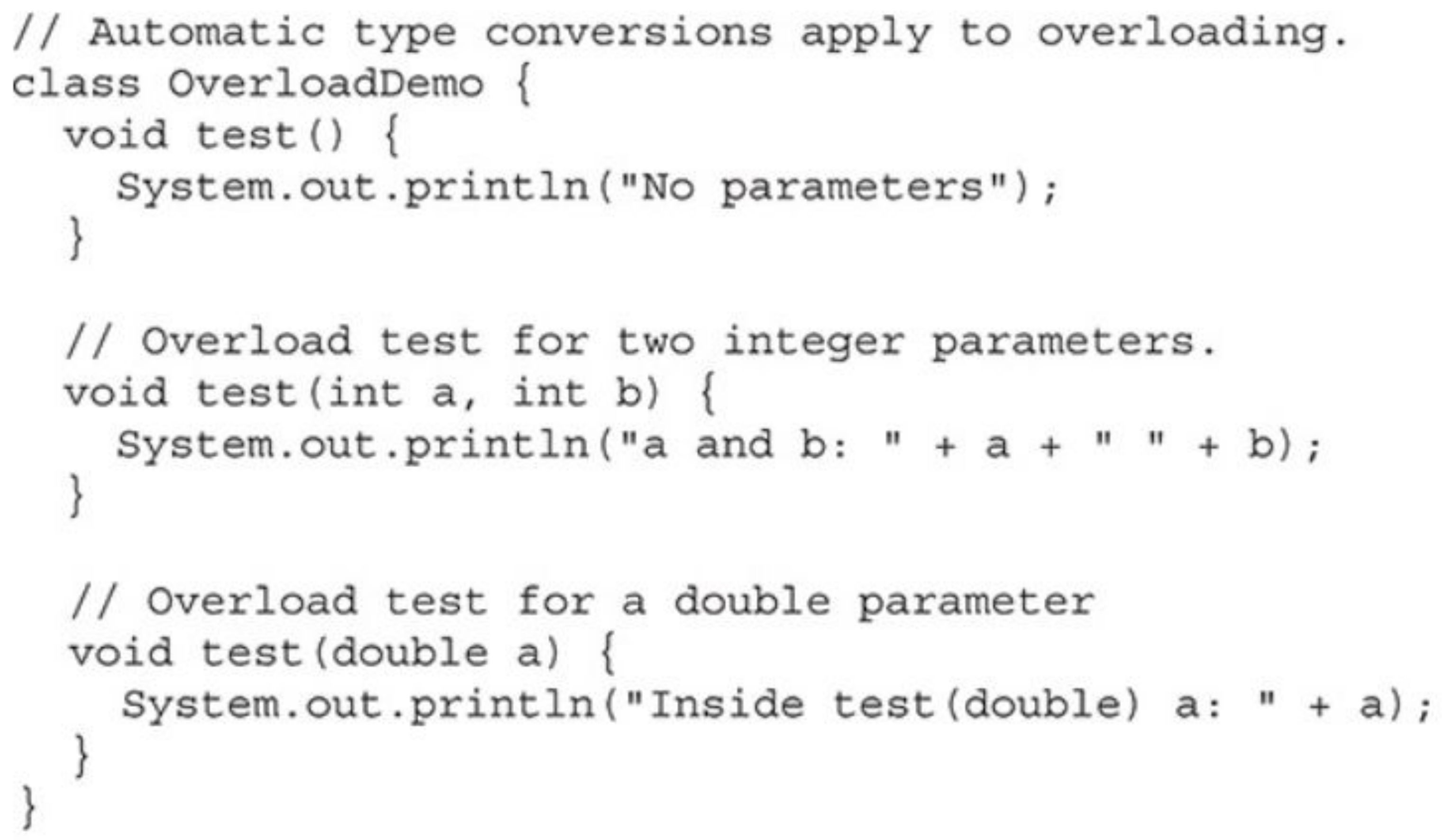
```
// Demonstrate method overloading.
class OverloadDemo {
  void test() {
    System.out.println("No parameters");
  }

  // Overload test for one integer parameter.
  void test(int a) {
    System.out.println("a: " + a);
  }

  // Overload test for two integer parameters.
  void test(int a, int b) {
    System.out.println("a and b: " + a + " " + b);
  }

  // Overload test for a double parameter
  double test(double a) {
    System.out.println("double a: " + a);
    return a*a;
  }
}
```

What should be the output now???



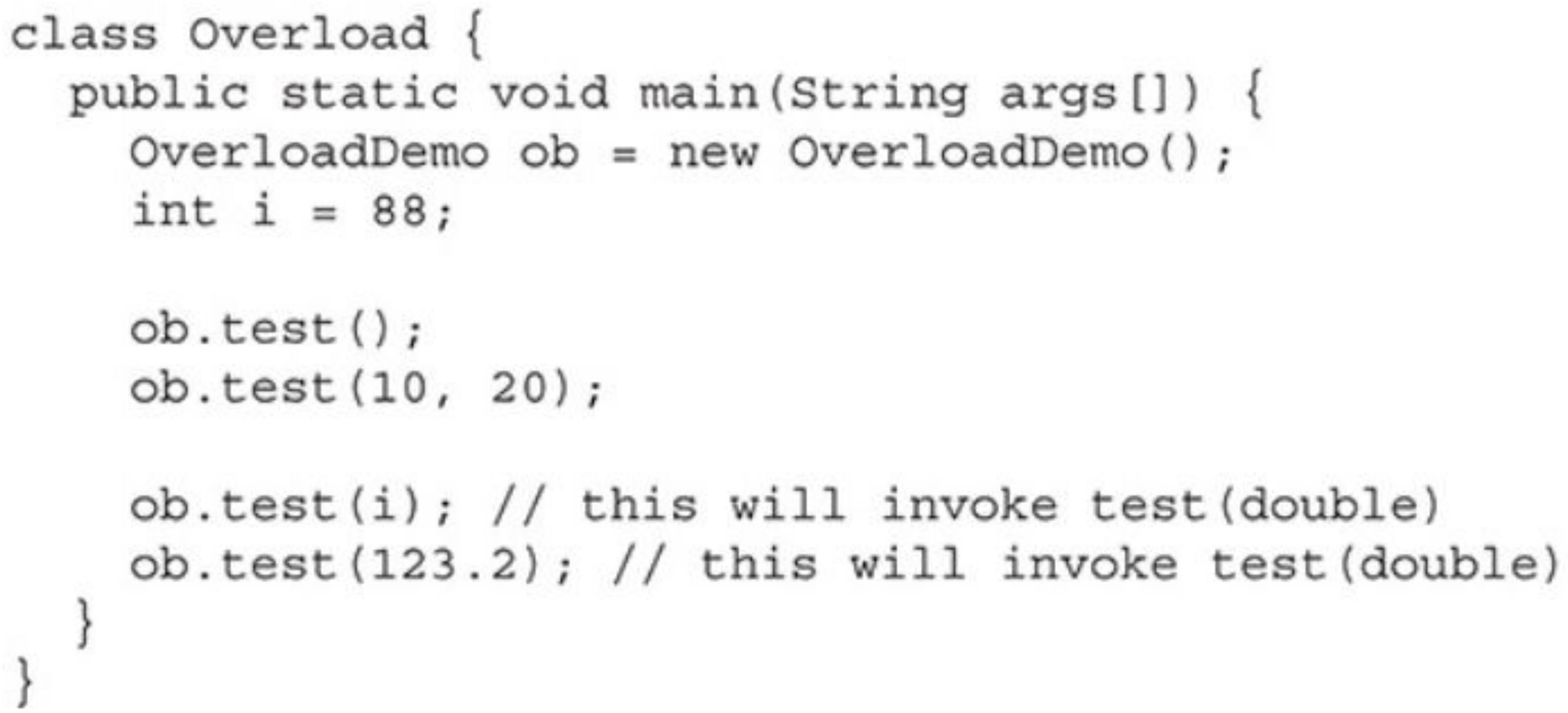
```
class Overload {
  public static void main(String args[]) {
    OverloadDemo ob = new OverloadDemo();
    double result;

    // call all versions of test()
    ob.test();
    ob.test(10);
    ob.test(10, 20);
    result = ob.test(123.25);
    System.out.println("Result of ob.test(123.25): " + result);
  }
}
```

Example of Automatic Type conversion



```java
// Automatic type conversions apply to overloading.
class OverloadDemo {
  void test() {
    System.out.println("No parameters");
  }

  // Overload test for two integer parameters.
  void test(int a, int b) {
    System.out.println("a and b: " + a + " " + b);
  }

  // Overload test for a double parameter
  void test(double a) {
    System.out.println("Inside test(double) a: " + a);
  }
}
```

Example of Automatic Type conversion



```
class Overload {
  public static void main(String args[]) {
    OverloadDemo ob = new OverloadDemo();
    int i = 88;

    ob.test();
    ob.test(10, 20);

    ob.test(i); // this will invoke test(double)
    ob.test(123.2); // this will invoke test(double)
  }
}
```

Overloading Methods

Advantage is common name methods for different activities of same nature, exp: Addition

Those languages which don’t support overloading(like C):

Give unique names for methods

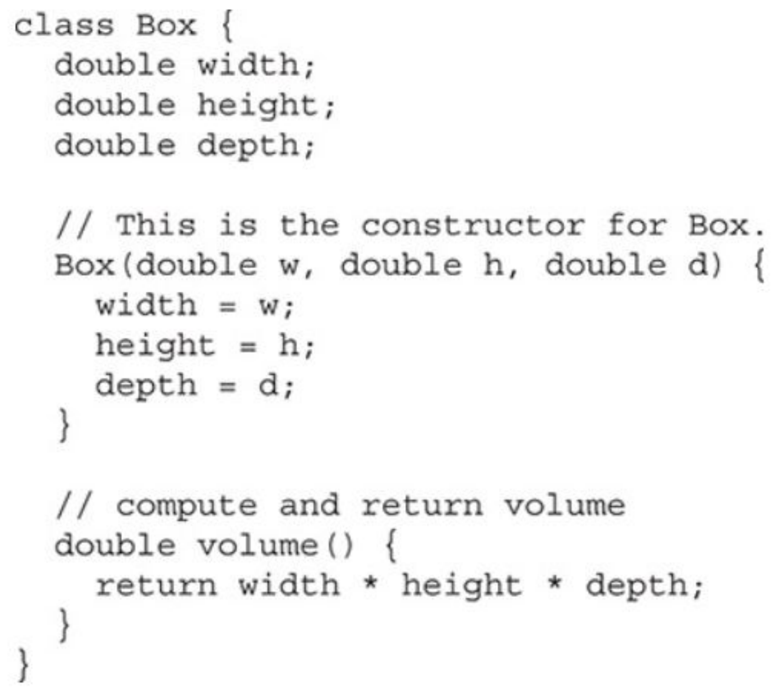
abs() for int and labs() for long int

It is one of the ways in which java can achieve polymorphism:

One object shows multiple behaviors

Overloading Constructors

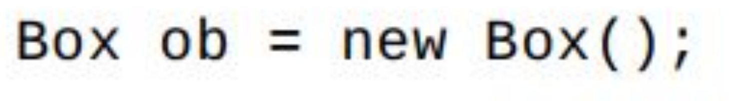
- Requires three parameters



```java
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;
  }
}
```

Overloading Constructors

- This is invalid right now



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

- What if you wanted a box without parameters

Some Questions?

What if you wanted a Box and didn’t care about initial dimensions?

What if you wanted to Initialize a cube by specifying only one value that will be used for all other dimensions

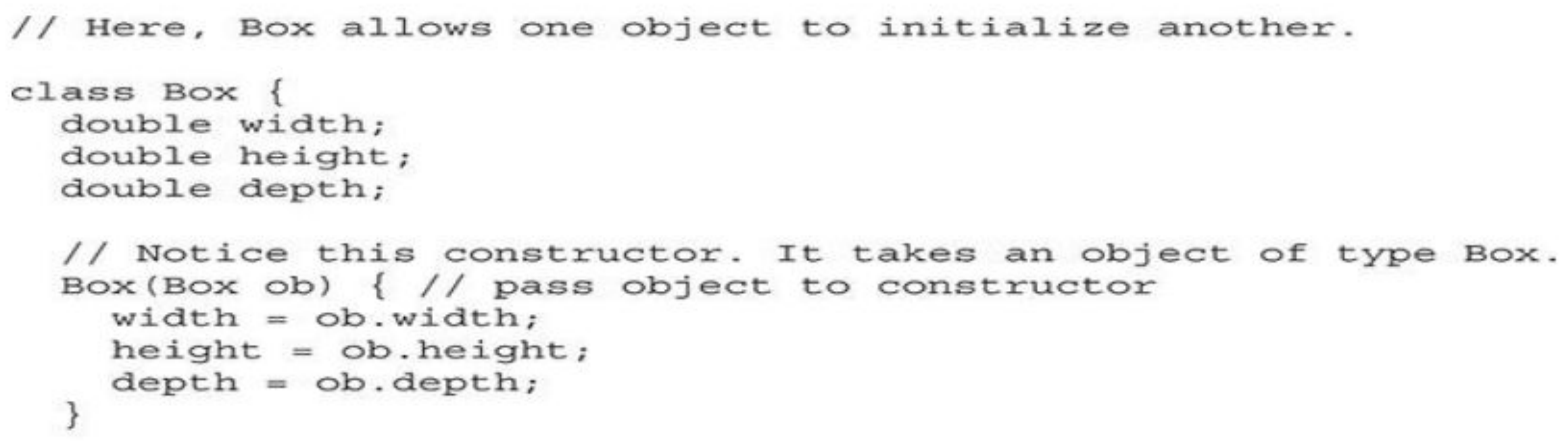
```
/* Here, Box defines three constructors to initialize
   the dimensions of a box various ways.
*/
class Box {
  double width;
  double height;
  double depth;

  // constructor used when all dimensions specified
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // constructor used when no dimensions specified
  Box() {
    width = -1;  // use -1 to indicate
    height = -1; // an uninitialized
    depth = -1;  // box
  }

  // constructor used when cube is created
  Box(double len) {
    width = height = depth = len;
  }

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

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

    double vol;

    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume of mybox1 is " + vol);

    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume of mybox2 is " + vol);

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

Passing Objects as Parameters to Methods

```java
// 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 equalTo(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.equalTo(ob2));
    System.out.println("ob1 == ob3: " + ob1.equalTo(ob3));
  }
}
```

Also used to copy values of one object to other with the help of constructors



```
// Here, Box allows one object to initialize another.

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;
  }
```

Encapsulation

Process of wrapping up code and data in a single unit

Exp: Capsule mixed from several herbs

We can create a fully encapsulated class in Java by making all the data members of the class private.

How to access and modify the values then?

Getter and Setter Methods

Setter:

- Assign/Modify the values of instance variables
- Also known as Mutator
- Return type is void

Getter:

- Used to retrieve or get the values of instance variables
- Also known as accessor
- Return type is similar to the datatype of instance variable

Why getter/setter, when we have constructor

Constructor just for Initialization

Setter for initializing, as well as modifying/changing the values

Getter for retrieving

Advantages of Encapsulation

Class can be read-only/write-only

It provides you the **control over the data:**

Value of Marks should be greater than zero

Salary shouldn’t be negative

Data Hiding is also achieved, because private member can only be accessed inside the methods of same class

Code becomes more easy to understand and readable

Can we make constructor private?

- Yes you can but on one cost:
  - You can not create the object outside of the class

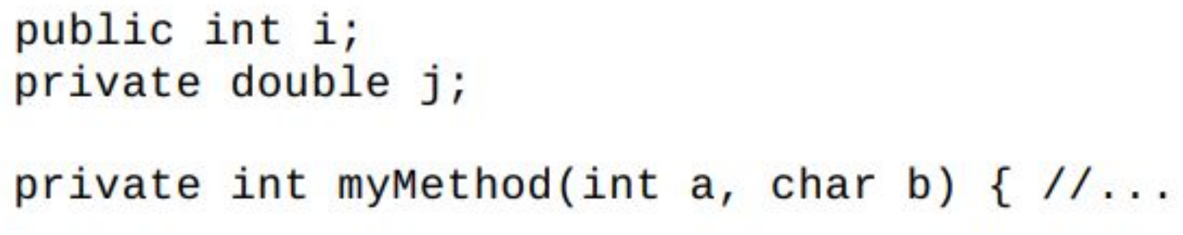
Demo

Task

- Create a class Student, a Student has Student_id, Student_name, and Program of study (Like CS,BBA etc), Make Five Objects of Student class
- Make Sure this class is a Encapsulated Class

Access Controls in Java

- Access Modifiers:
  - How a member(instance variable/method) can be accessed
    - Public
    - Private
    - Protected (Applies only when Inheritance is involved)
    - Default



```
public int i;
private double j;

private int myMethod(int a, char b) { //...
```

Access Modifiers

Private: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.

Default: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.

Protected: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.

Public: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Understanding *static* member variables

- static member variable of class
  - Associated with class
  - Independent of object
  - One copy for all objects
  - Accessed without any object reference (Main Method)

Understanding *static* member variables

- Instance variables which are static: Global Variables:
    - One copy is shared among all objects
- Change the value of static variable of one object:
    - Change will be visible to all objects, because there is only one copy for all of them

Understanding *static* member methods

- Can only call other static methods:
  - Let’s Try to call Non-Static method
- Can access only static member variables of class:
  - Let’s Try to access non-static members
- Can be called on class and the object reference:
  - Let’s try both
- Can’t refer *this* and *super* keywords
- Example

How to access them

- Outside the class in which they are defined:
    - Classname.methodname();
    - Classname.datamember;

Understanding *static* code block

- Gets executed exactly once at the start

- Example (static member variable, method and block)

Introducing keyword *final*

Usage with variable:

- In Java, the final keyword is used to define constants or variables whose value cannot be changed once assigned.

Introducing keyword *final*

Usage with Methods:

- In Java, the final keyword can also be used with **methods** to prevent them from being overridden by subclasses.
- This is particularly useful when you want to ensure that a method's behavior remains consistent and cannot be altered by any subclass

Nested and Inner Classes

In Java, **nested classes** are classes that are defined within another class. There are two types of nested classes:

- **Inner Classes**: These are non-static nested classes that are associated with an instance of the outer class. They can access all the members (including private members) of the outer class.
- **Static Nested Classes**: These are static nested classes that do not require an instance of the outer class to be created. They can only access static members of the outer class.

Types of Nested Classes:

Inner Classes:

- Non-static nested classes are called **inner classes**.
- Inner classes have access to all the members (fields and methods) of the outer class, even the private ones.
- An inner class is associated with an instance of the outer class and can be instantiated only through an instance of the outer class.

Types of Nested Classes:

Static Nested Classes:

- A **static nested class** is a nested class marked with the static keyword. This means it is not tied to an instance of the outer class, and can be instantiated independently of the outer class.
- A static nested class can only access the static members (fields and methods) of the outer class.

Local Inner Classes:

- A **local inner class** is a class defined within a method or a block of code. It is only accessible within the method or block where it is defined.
- Local inner classes can access local variables and parameters of the method, but only if they are declared as final or effectively final (i.e., their value is not changed after initialization).

Nested and Inner Classes

- Static class:
  - Declared inside a class
  - Using static keyword
  - Can have static/non-static data members
  - Can have static/non-static methods
  - Can not access the non-static members of outer class directly, will have to use an object

Nested and Inner Classes

- Inner classes/ Non Static:
  - Nested/inner has access to member variables/methods including private of outer
  - Outer has no direct access of inner class methods/variables, an object of inner needs to be created

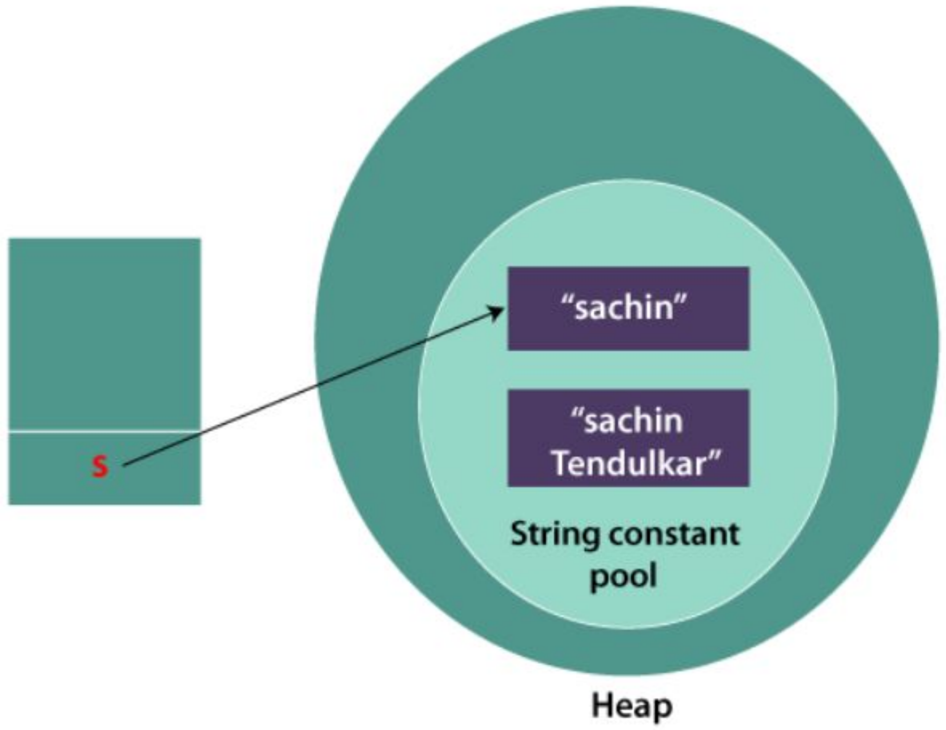
String class

- Class in java.lang
- Even string constants inside System.out.print() are string objects
- String objects are immutable
- StringBuffer and StringBuilder are mutatable strings

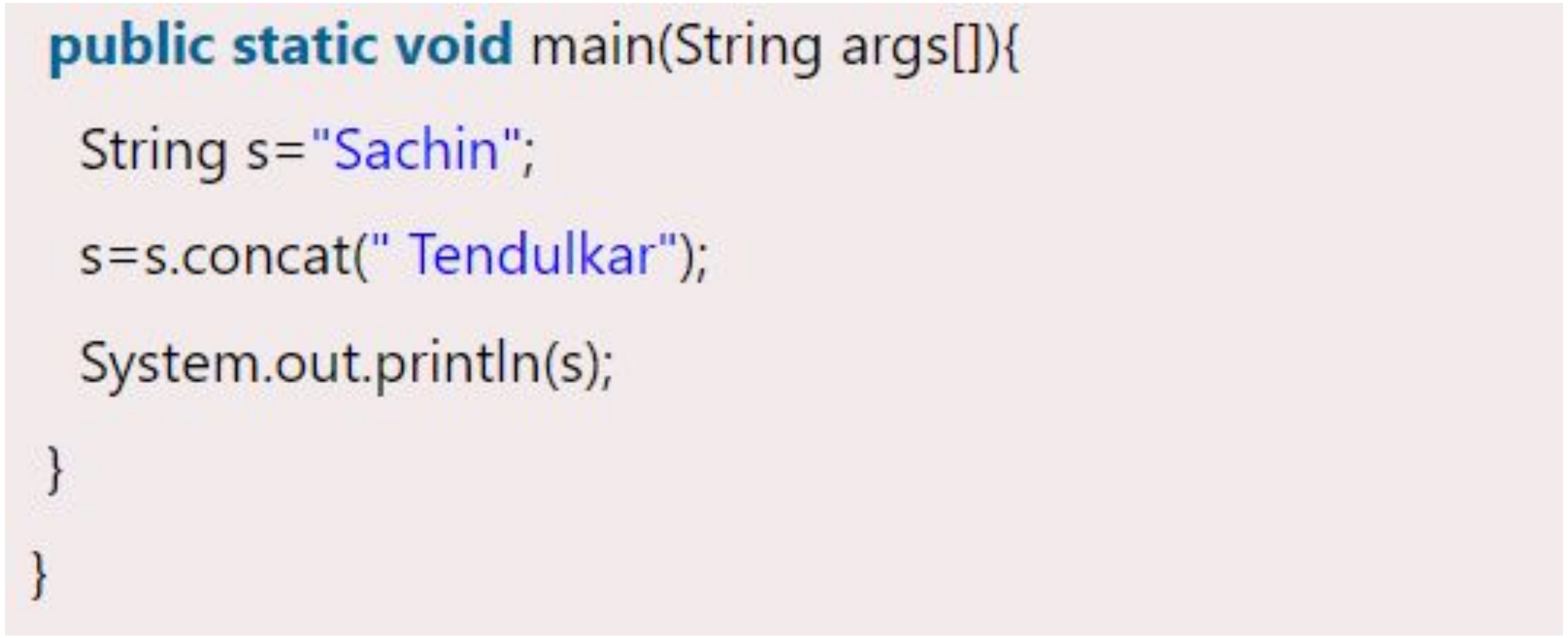
Strings are Immutable!

- Once you create a string, you can not change it’s value
- If you try to change the value a new string object is created
- But Why?
  - Strings are objects

```
String s="Sachin";
s.concat(" Tendulkar");//concat() method appends the string at the end
System.out.println(s);//will print Sachin because strings are immutable objects
```



Solution: Assign Reference to new string Object explicitly

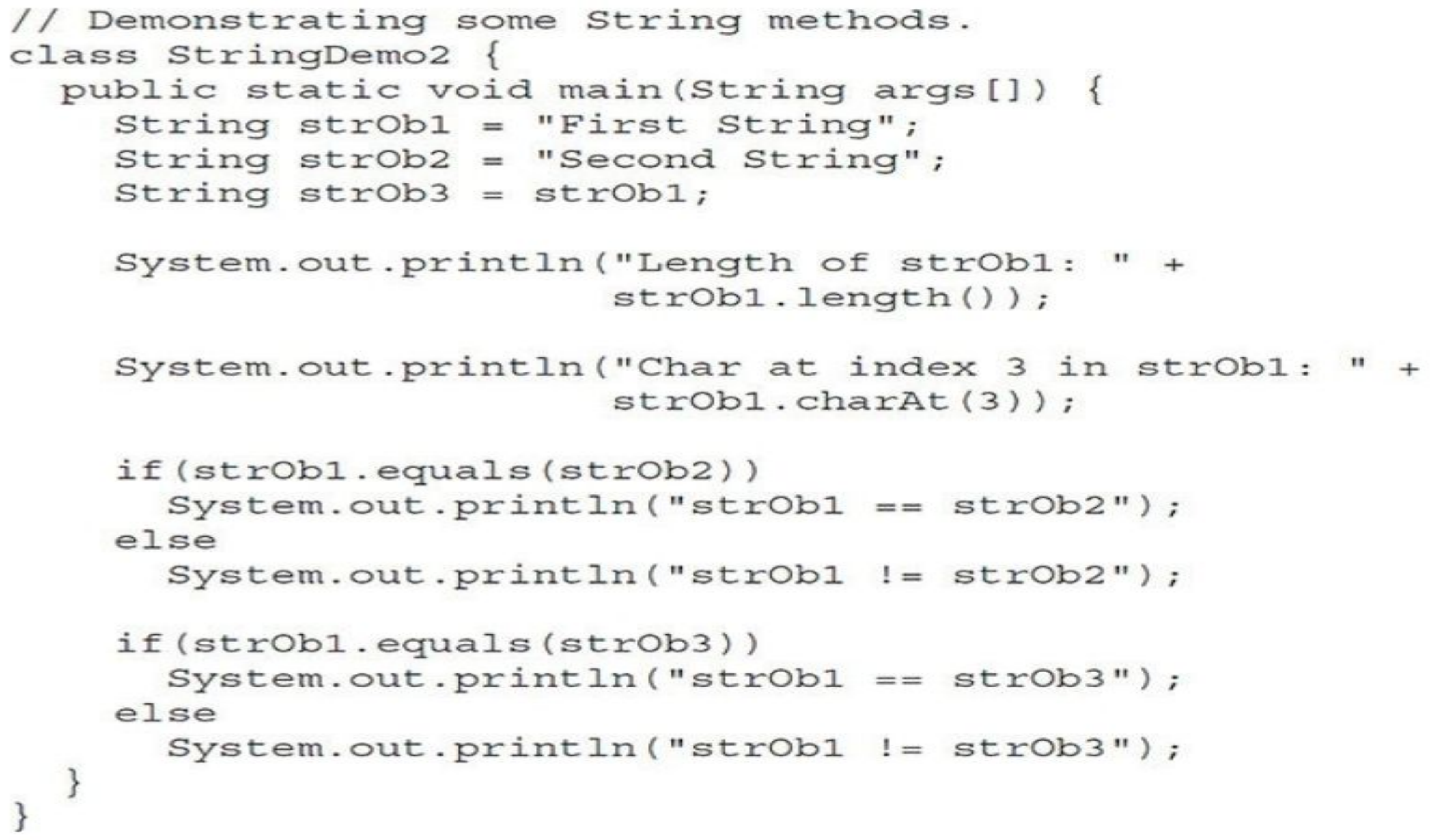


```
public static void main(String args[]){
  String s="Sachin";
  s=s.concat(" Tendulkar");
  System.out.println(s);
 }
}
```

String class

- String methods
  - equals()
  - equalsIgnoreCase()
  - length()
  - charAt()

Demo



```
// Demonstrating some String methods.
class StringDemo2 {
  public static void main(String args[]) {
    String strOb1 = "First String";
    String strOb2 = "Second String";
    String strOb3 = strOb1;

    System.out.println("Length of strOb1: " +
                       strOb1.length());

    System.out.println("Char at index 3 in strOb1: " +
                       strOb1.charAt(3));

    if(strOb1.equals(strOb2))
      System.out.println("strOb1 == strOb2");
    else
      System.out.println("strOb1 != strOb2");

    if(strOb1.equals(strOb3))
      System.out.println("strOb1 == strOb3");
    else
      System.out.println("strOb1 != strOb3");
  }
}
```

String class

- equals() method vs == operator
  - equals() checks value
  - == checks reference

Random

- A class in java.util
- Methods for generating random numbers
- nextInt()
- nextInt(int n): generates random number between 0 and n
- nextInt(-value): gives error

Demo

```
import java.util.Random;
int randomInt = random.nextInt(100);

// Generate a random integer between 10 and 100 (inclusive)
        int randomInt = 10 + random.nextInt(91); // 100 - 10 + 1 = 91
```

:Simulate the roll of a dice (1 to 6)