COURSE:- Java Programming

Sem:- I

Module: 2

By:
Dr. Ritu Jain,
Associate Professor

Class

- A class is a user-defined blueprint or template from which objects are created. It represents the set of properties or methods that are common to all objects of one type.
- Class defines a new data type. Once defined, this new type can be used to create objects of that type.
- A class is a template for an object and an object is an instance of a class.
- When you define a class, you specify the
  - ✓ data it contains and
  - ✓ the code that operates on that data.

Created by Dr. Ritu Jain

Object

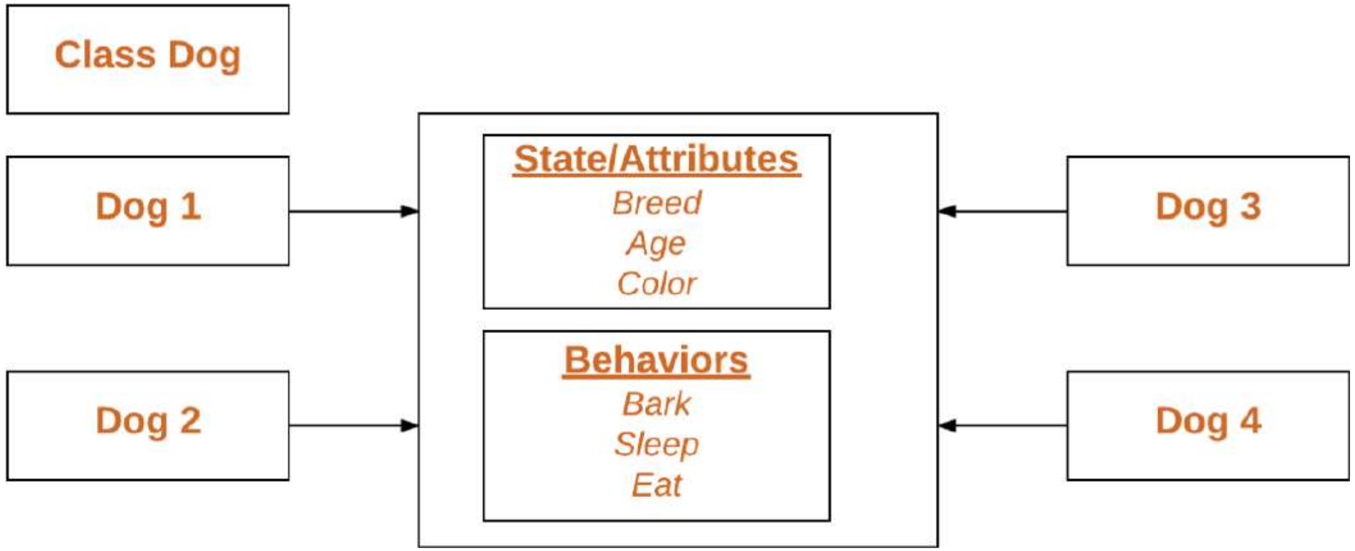
- It represents real-life entity.
- Real-life entities share two characteristics: they all have attributes and behavior.

An object consists of:

- **State:** It is represented by *attributes* of an object. It also shows properties of an object.
- **Behavior:** It is represented by *methods* of an object. It shows response of an object with other objects.
- **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

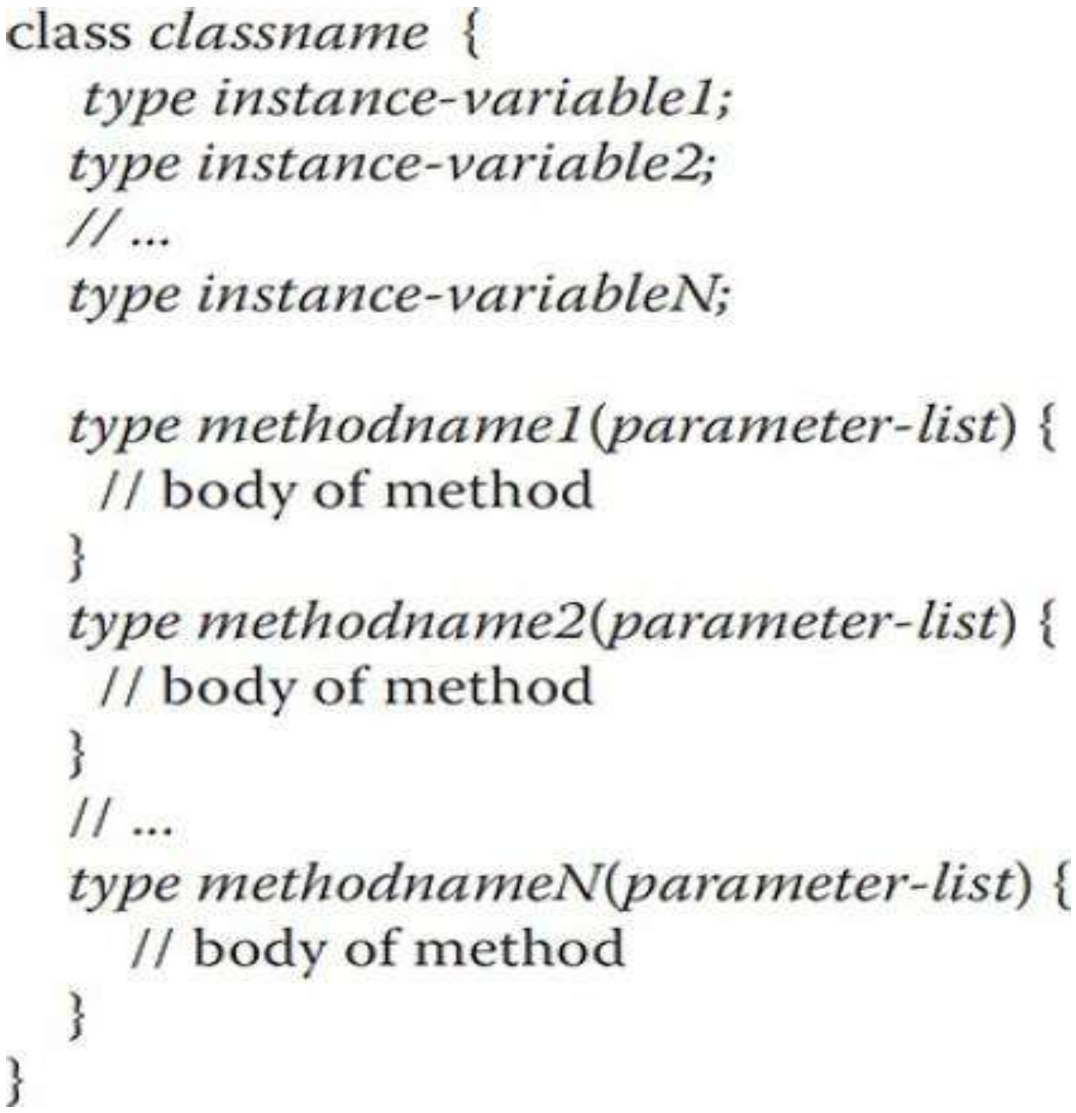
Declaring Objects

Declaring Objects (Also called instantiating a class)
When an object of a class is created, the class is said to be **instantiated**. All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances.



-

✓

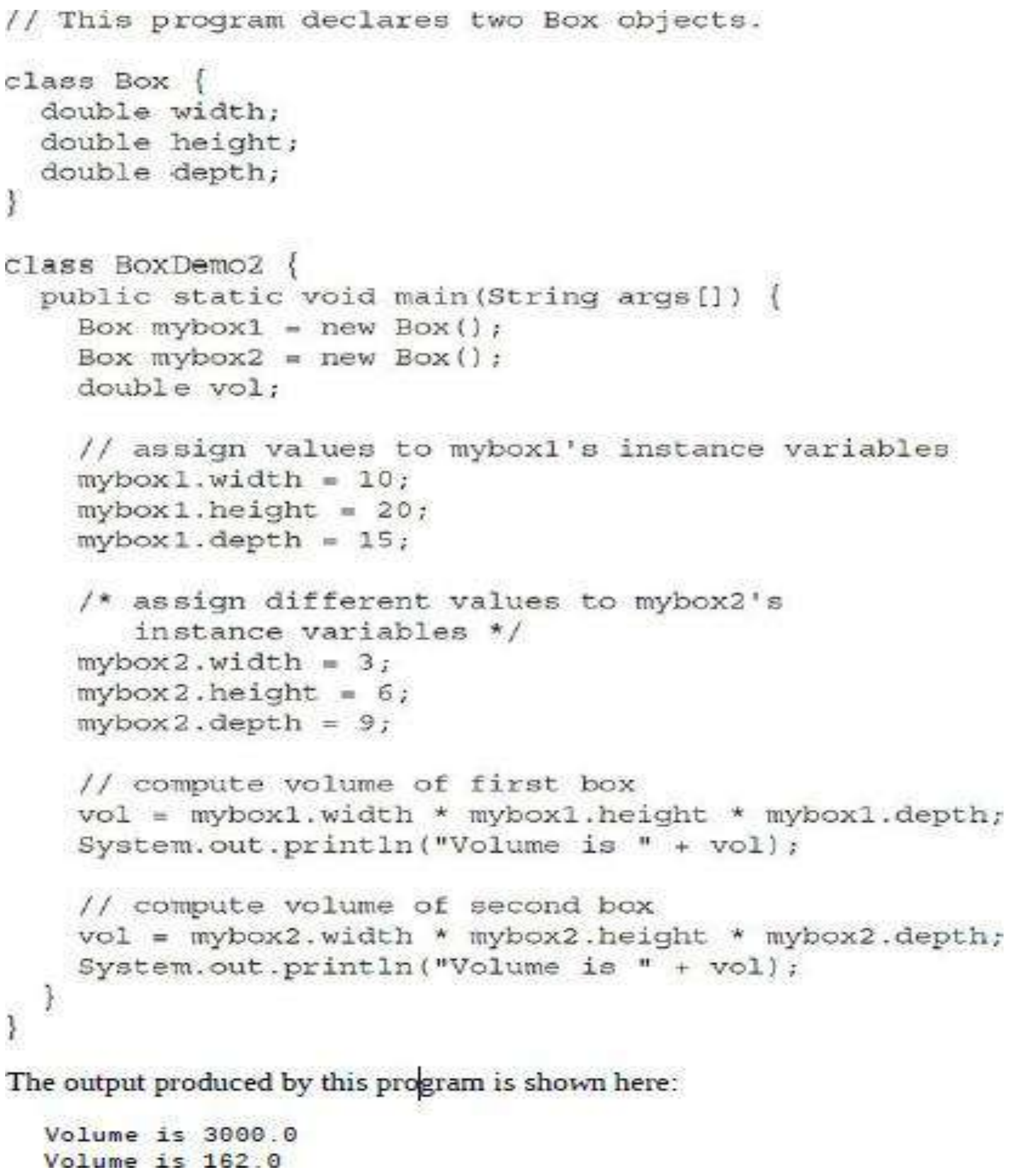


```
class classname {
    type instance-variable1;
    type instance-variable2;
    // ...
    type instance-variableN;

    type methodname1(parameter-list) {
      // body of method
    }
    type methodname2(parameter-list) {
      // body of method
    }
    // ...
    type methodnameN(parameter-list) {
        // body of method
    }
}
```

Class

- The 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 code is contained within *methods*.
- The variables and the methods within a class are called as *members* of the class.
- Each instance of a class contains its own copy of the instance variables. This implies that the data for one object is separate and unique from the data of another object.

- Remember that all Java classes do not need to have a **main()** method. You only specify a **main()** method in a class if that class is the starting point of the program.



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

The output produced by this program is shown here:

```
  Volume is 3000.0
  Volume is 162.0
```

How to create objects

- A class declaration only creates a template; it does not create an actual object.
- General form to create object:
  - *Class_name class-var* = new *classname* ( );

- Example:
  - Box mybox = new Box();

OR

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

- The **new** operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is then stored in the variable.
- The class name followed by parentheses specifies the *constructor* for the class. A constructor defines what occurs when an object of a class is created.
- If no explicit constructor is specified, then Java will automatically supply a default constructor. This is the case with **Box**. For now, we will use the default constructor.

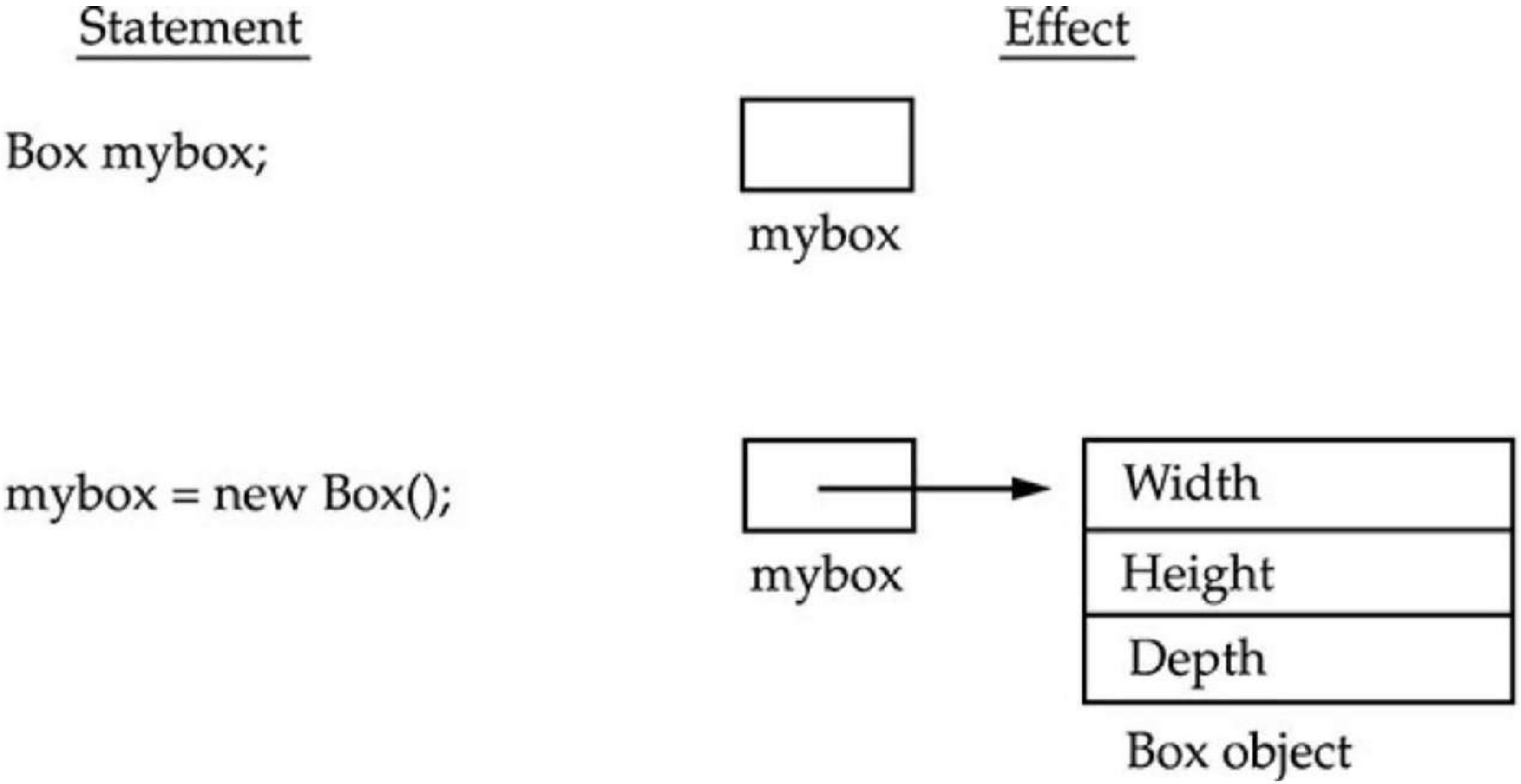


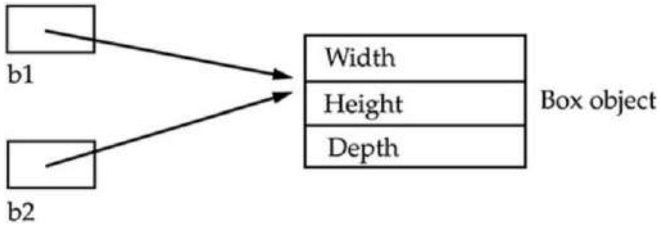
Figure 6-1 Declaring an object of type **Box**

Distinction between class and object

- ✓ A class creates a new data type that can be used to create objects. That is, a class creates a logical framework that defines the relationship between its members.
- ✓ When you declare an object of a class, you are creating an instance of that class.
- ✓ Thus, a class is a logical construct. An object has physical reality. (That is, an object occupies space in memory.) It is important to keep this distinction clearly in mind.

Assigning Object Reference Variables

- Box b1 = new Box(); Box b2 = b1;
- It simply makes **b2** refer to the same object as does **b1**. Thus, any changes made to the object through **b2** will affect the object to which **b1** is referring, since they are the same object.



- •Although **b1** and **b2** both refer to the same object, they are not linked in any other way.
-

Methods in class

- Classes usually consist of two things: instance variables and methods.
- general form of a method:

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

- *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*;
- Here, *value* is the value returned.

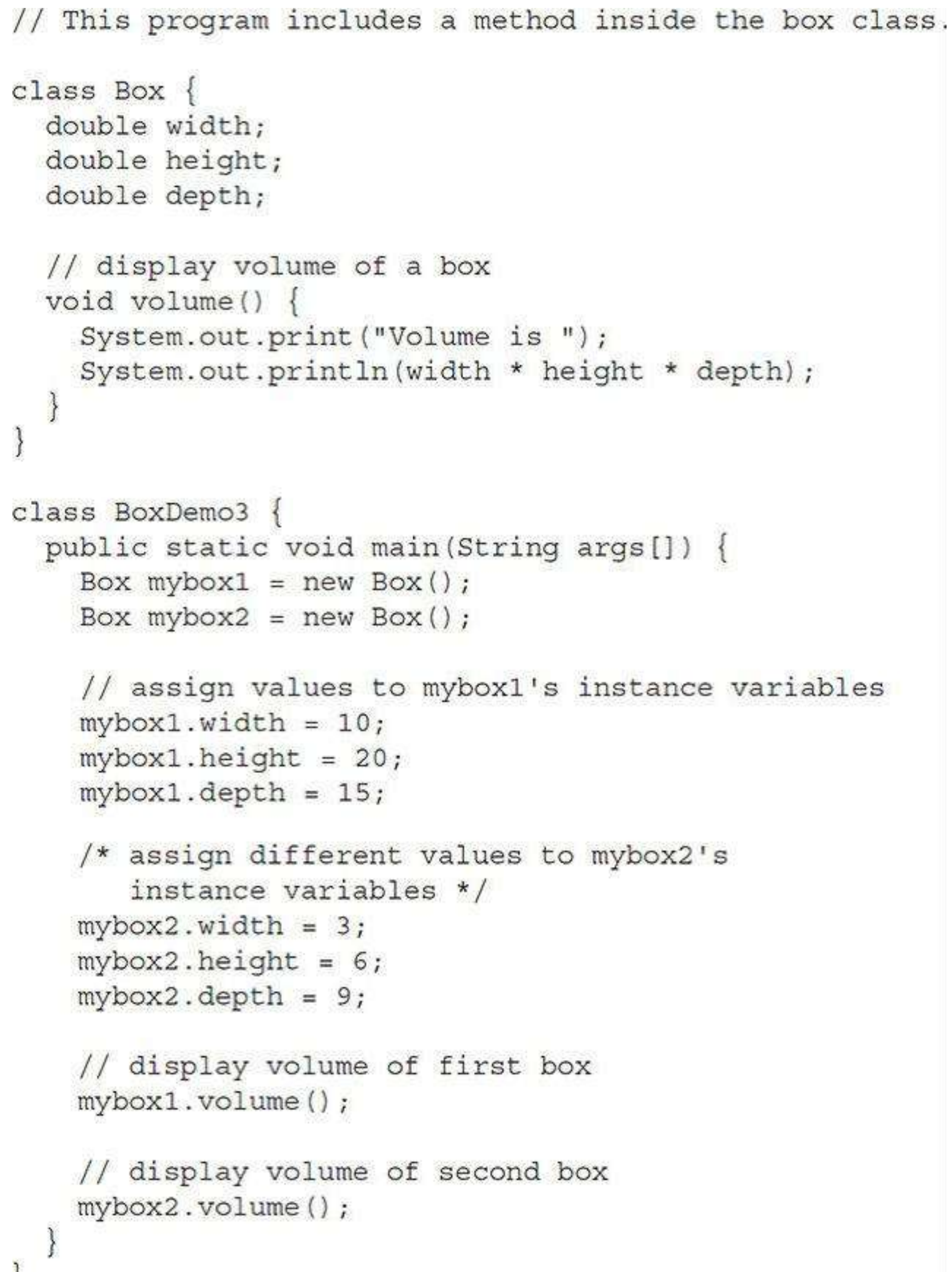
Methods in class

- Methods define the interface to most classes. This allows the class implementor to hide the specific layout of internal data structures behind cleaner method abstractions.
- In addition to defining methods that provide access to data, you can also define methods that are used internally by the class itself.

Accessing class members

- *dot (.) Operator is used to access instance variable as well as methods of a class.*
- Every object have its own copy of instance variable. Changes to the instance variables of one object have no effect on the instance variables of another.
- Call a variable as follows:
  - objectname.variablename;
  - Ex: mybox1.width=10;
- Call a class method as follows
  - objectname.methodname();
  - Ex: mybox1.volume();

Example 1 to show how to define classes, instance variable and methods in the class. Create objects and access instance variables and call instance methods of a class through objects



```
// This program includes a method inside the box class.

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();
  }
}
```

Accessing class members

- When an instance variable is accessed by code that is not part of the class in which that instance variable is defined, it must be done through an object, by use of the dot operator.
- However, when an instance variable is accessed by code that is part of the same class as the instance variable, that variable can be referred to directly.
- The same thing applies to methods.

```
class Employee{
  int emp_id;
  String name;
  String dept;
  float   salary;
 void add_info (int id, String n, String d, float sal) {
    emp_id = id;
    name = n;
    dept = d;
    salary = sal;
  }
void display() {
   System.out.println("Employee id: " + emp_id );
   System.out.println("Employee name: " + name );
   System.out.println("Employee department: " + dept );
   System.out.println("Employee salary: " + salary );
}}
```

```
public class EmployeeClass {
  public static void main(String[] args) {
    Employee Employee e1 = new Employee();
    Employee e2 = new Employee();
    Employee e3 = new Employee();
    e1.add_info (101, "Naman", "Salesforce", 45000);
    e2.add_info (102, "Riya", "Tax", 25000);
    e3.add_info (103, "Anu", "Development", 55000);
    e1.display();
    e2.display();
    e3.display();
  }
}
```

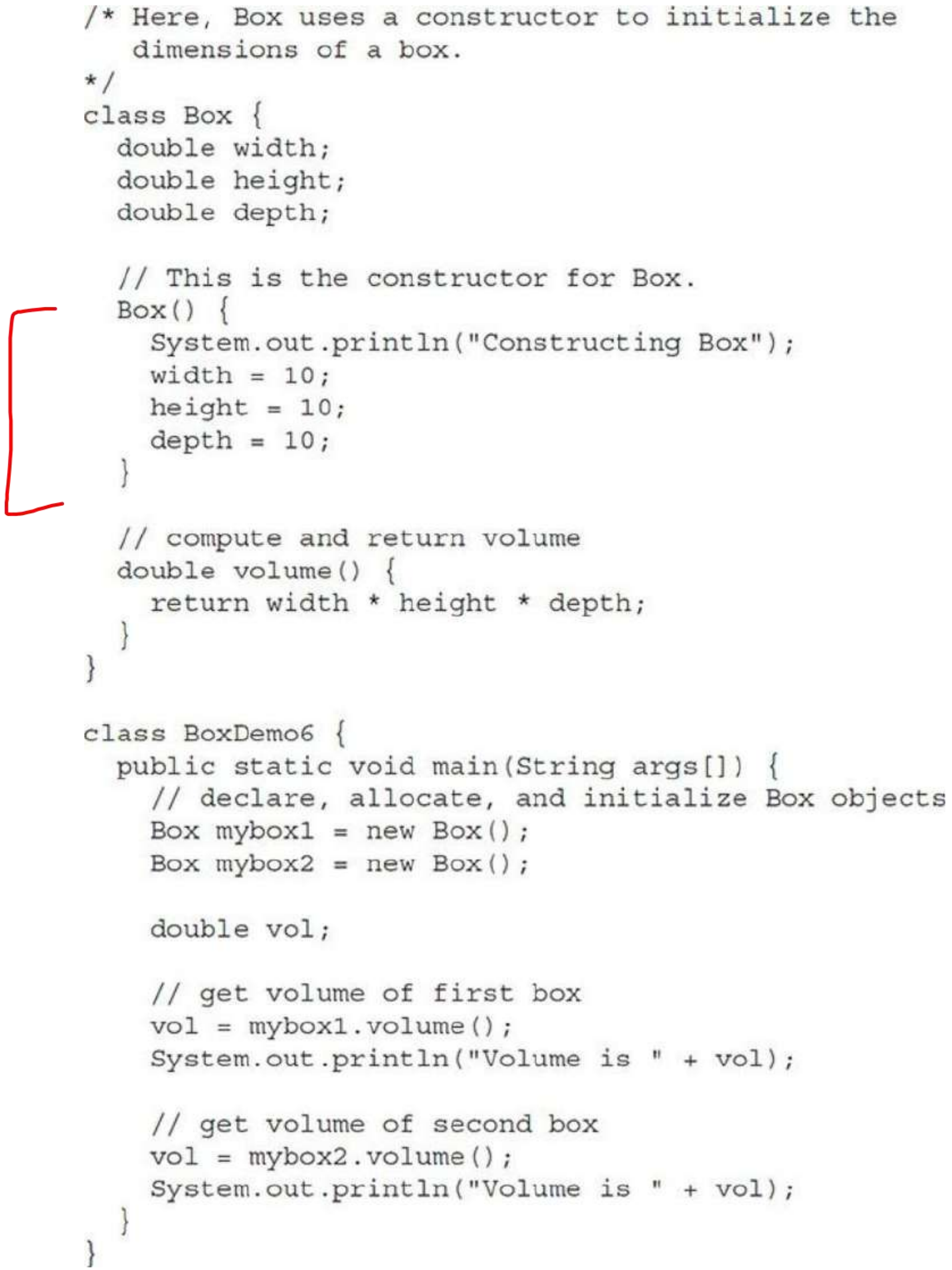
Constructor

- Java allows objects to initialize themselves when they are created.
- This automatic initialisation is performed through the use of a constructor. A constructor initialises an object as soon as it is created.
- **Rules for constructor:**
  - *It has the same name as the name of the class in which it resides.*
  - *Constructors do not have a return type, not even **void**.*
  - This is because the implicit return type of the class' constructor is the class type itself.
  - It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.

Default Constructor

- A default constructor is one that takes no arguments.
- It is used to initialize objects with default values.
- If you don’t explicitly define any constructor, Java automatically provides a default constructor.

Example of Default Constructor (with no parameters)



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

If Constructor is not defined?

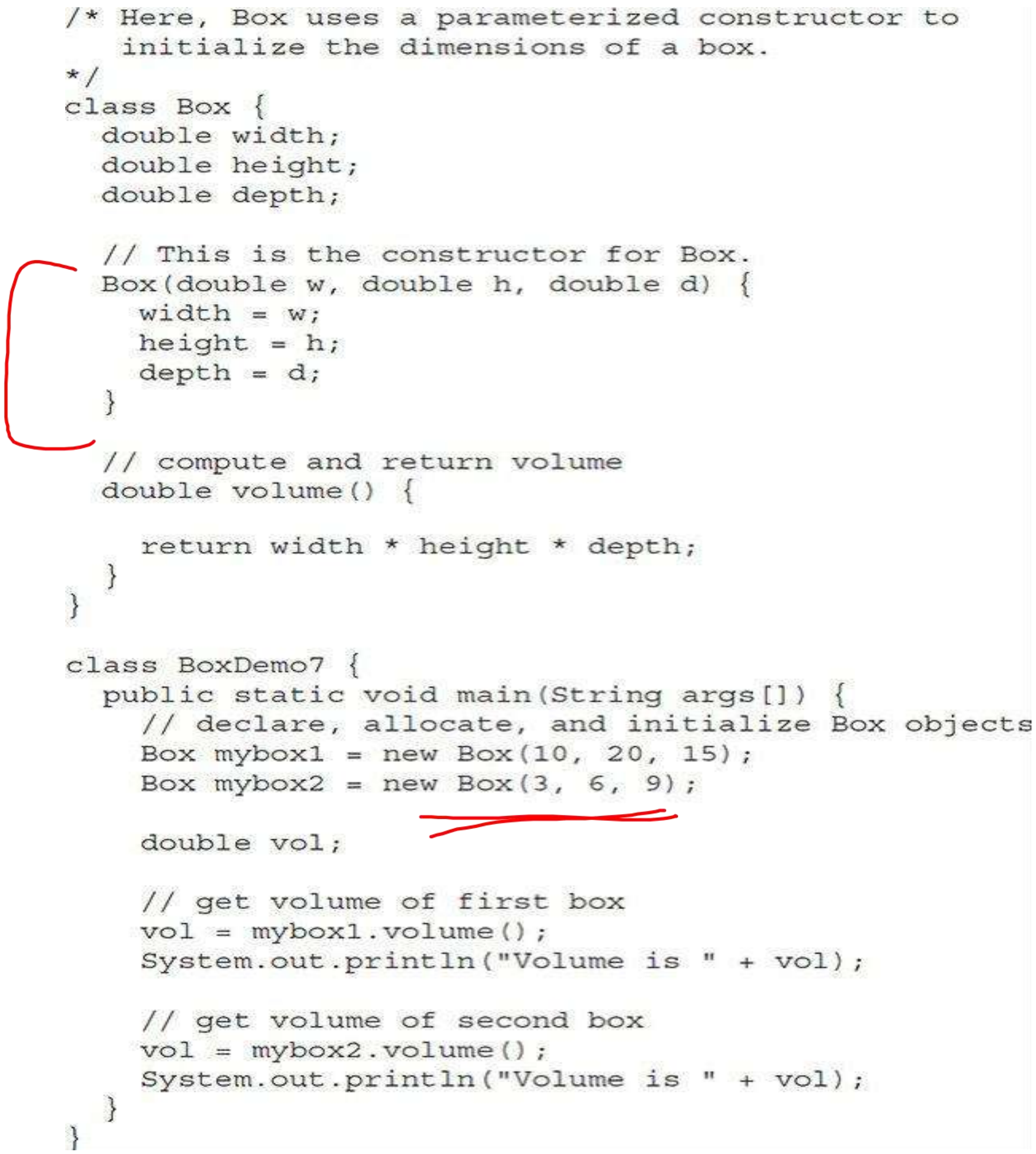
- ✓ When you do not explicitly define a constructor for a class, then Java creates a default constructor for the class. This is why the preceding line of code worked in earlier versions of **Box** that did not define a constructor.
- ✓ When using the default constructor, all non- initialized instance variables will have their default values, which are zero, **null**, and **false**, for numeric types, reference types, and **boolean**, respectively.

Parameterized Constructors

- ✓ We saw that the above constructor initialized all the objects with the same values.
- ✓ But actually what is needed in practice is that we should be able to set different initial values.
- ✓ For this purpose, we make use of parameterized constructors and each object gets initialized with the set of values specified in the parameters to its constructor.
- ✓ The following modification in the above program will illustrate :

- **Created by Dr. Ritu Jain**

Example of Parameterized Constructor



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

Copy Constructor

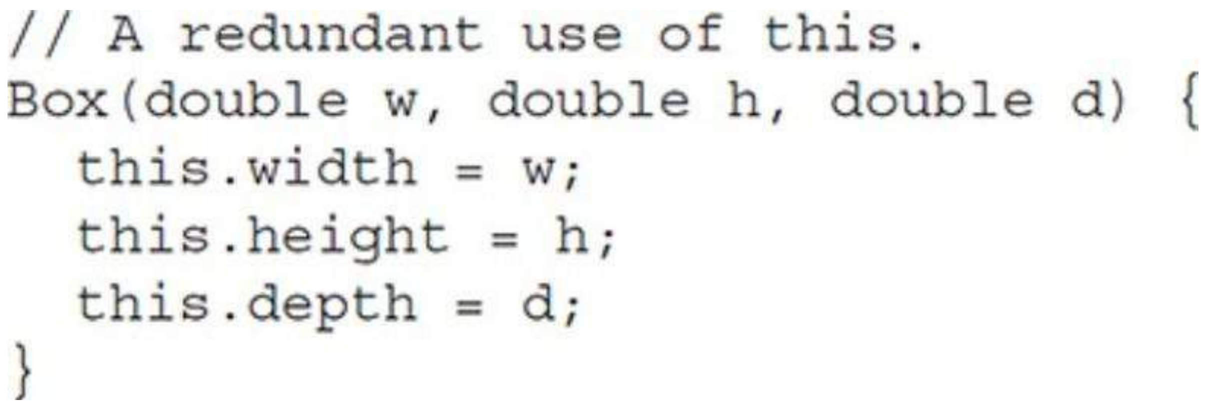
- Java does not have a built-in or automatic copy constructor like C++ does, but developers can define their own copy constructor manually in a class.
- A copy constructor in Java is simply a constructor that takes an object of the same class as a parameter and copies its field values to produce a new, independent object.
- The following example creates a new Student object by copying the values from an existing Student object, demonstrating the concept and use of a copy constructor in Java

Example of Copy Constructor

```java
public class Student {
    private String name;
    private int age;

    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Copy constructor
    public Student(Student student) {
        this.name = student.name;
        this.age = student.age;
    }

    public String toString() {
        return "Name: " + name + ", Age: " + age;
    }

  public static void main(String[] args) {
        Student student1 = new Student("John", 20);      // Original object
        Student student2 = new Student(student1);        // Copy using copy constructor

        System.out.println(student1); // Output: Name: John, Age: 20
        System.out.println(student2); // Output: Name: John, Age: 20
    }
}
```

this Keyword

- **this** is a keyword that can be used inside any method to refer to the current object i.e **this** is always a reference to the object on which the method was invoked. This is useful when a method needs to refer to the object that invoked it.



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

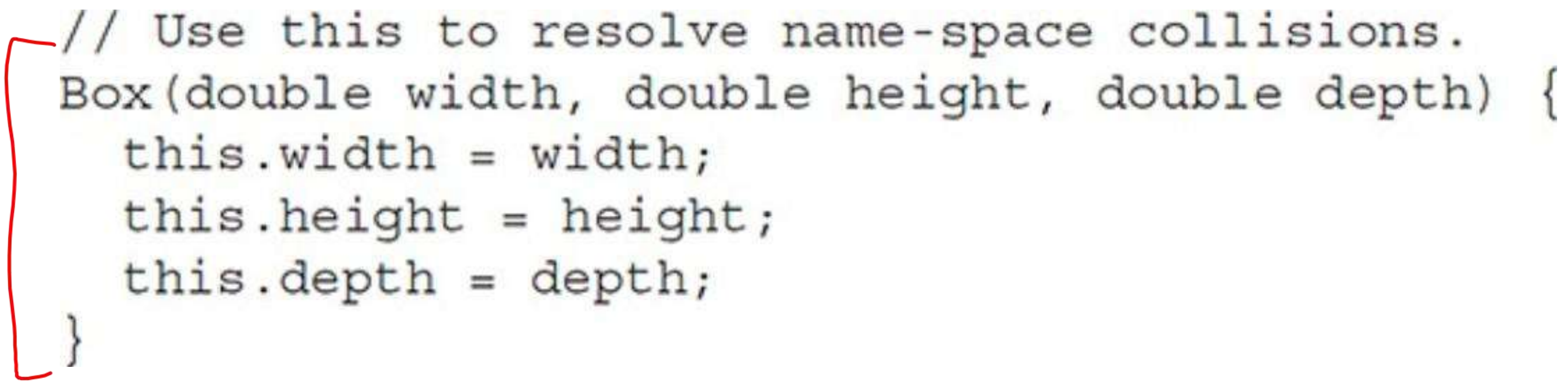
//use of **this** is redundant, but perfectly correct.

Created by Dr. Ritu Jain

Instance Variable hiding

- In Java, it is illegal to declare two local variables with the same name within the same or enclosing scopes.
- However, you can have local variables, including formal parameters to methods which overlap with the names of the class' instance variables.
- But when a local variable has the same name as the instance variable, then the local variable hides the instance variable.
- We can use the **this** keyword to refer directly to the object and thus resolve the name space collisions that might occur between the instance variables and the local variables.

Using *this*



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

Types of Constructor

Type	Description	Example

1. Default Constructor	A constructor **with no parameters**. It initializes objects with **default values**. If no constructor is defined, Java provides one automatically.	Person()
		{}

2. Parameterized Constructor	A constructor that **accepts parameters** to initialize objects with **specific values**.	Person(String n, int a) {
		name = n;
		age = a; }

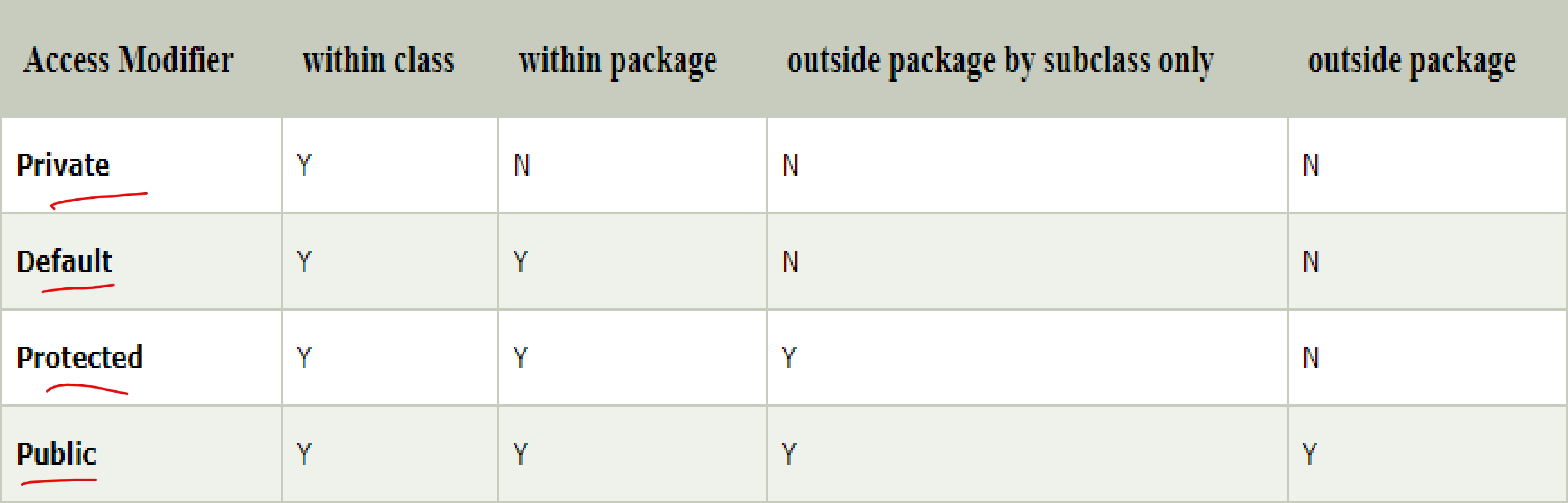
3. Copy Constructor (User-defined)	Java doesn't provide a built-in copy constructor (unlike C++), but you can **create one manually** to **copy values** from another object.	Person(Person p) {
		name = p.name;
		age = p.age; }

Getters and Setters

- In Java, Getter and Setter are methods used to protect your data and make your code more secure. Getter and Setter make the programmer convenient in setting and getting the value for a particular data type.
- **Getter in Java:** Getter returns the value (accessors), it returns the value of data type int, String, double, float, etc. For the program's convenience, the getter starts with the word "get" followed by the variable name.
  - Eg: public int getNum(){return num;}
- Getter method begin with "is" if return type is boolean.
  - Eg. public boolean isHappy(){ return happy;}
- **Setter in Java:** While Setter sets or updates the value (mutators). It sets the value for any variable used in a class's programs. and starts with the word "set" followed by the variable name.
  - Eg. public void setNum(int n){ num=n;}

Access Control through Access modifiers

- **public:** Member can be accessed by any other code. It can be accessed from within the class, outside the class, within the package and outside the package.
- **private:** member cannot be accessed from outside the class.
- **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.
- **Default access level:** 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.



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

```java
// Example for concept of class:  DogClass
public class DogClass {
    private String name;
    private String breed;
    private int age;
    private String color;
   public DogClass(){}

    // Constructor Declaration of Class
    public DogClass(String name, String breed, int age,
               String color)
    {
        this.name = name;
        this.breed = breed;
        this.age = age;
        this.color = color;
    }
    public String getName() { return name; }
    public String getBreed() { return breed; }
    public int getAge() { return age; }
    public String getColor() { return color; }

    public String toString()
    {
        return ("Hi my name is "+name
                + ".\nMy breed,age and color are "
                + breed + "," + age
                + "," + color);
    }

    public static void main(String[] args)
    {
        DogClass tuffy
                = new DogClass("tuffy", "papillon", 5, "white");
        System.out.println(tuffy.toString());
        System.out.println(tuffy);
    }
}
```

getters

Binding

- In Java, **binding** is the process of linking a **method call** with the **method to be executed.**
- If this linking happens **at compile time**, it is called **Static Binding (Early Binding)**.
- If this linking happens **at runtime**, it is called **Dynamic Binding (Late Binding)**.

Compile Time Polymorphism: Method Overloading

Compile Time Polymorphism: Method Overloading

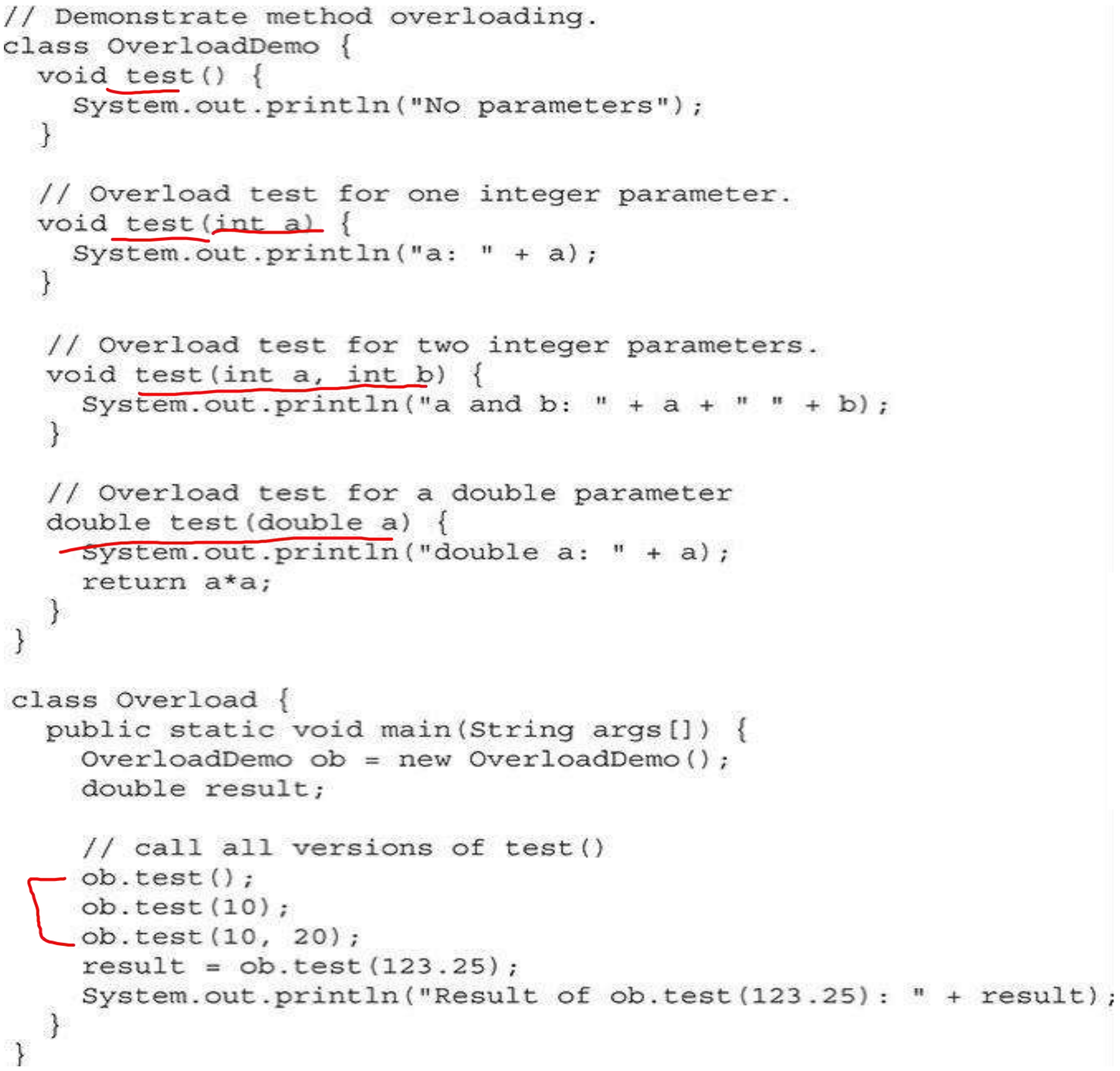
- **Compile-time polymorphism in Java** is the type of polymorphism that is resolved **at compile time**. The compiler decides which method to invoke based on the method signature during compilation.
- It is achieved by **method overloading.**
- Method Overloading allows different methods to have the same name, but different signatures.
- Signature can differ by either:
  - ✓ number of parameters or
  - ✓ type of parameters or
  - number of parameters and type of parameters

Compile Time Polymorphism: Method Overloading

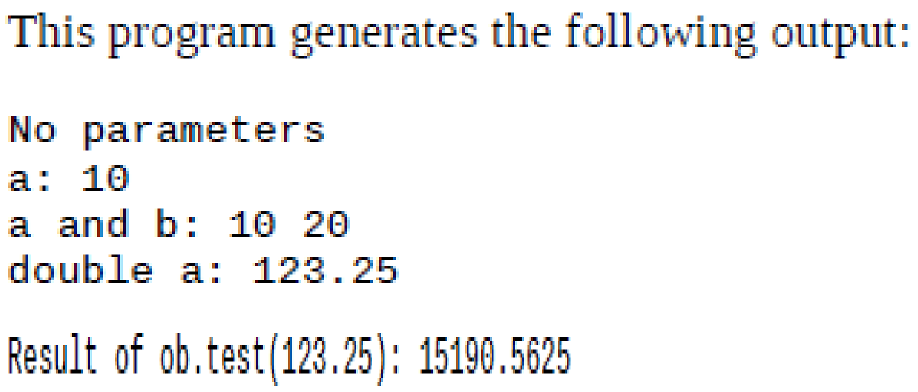
- Java supports overloading. It is one of the ways that Java implements the concept of polymorphism.
- Method overloading supports polymorphism because it is one way that Java implements "one interface, multiple methods" paradigm.
- **The purpose of method overloading is that it allows related methods to be accessed by use of a common name.**
- Programmer need only remember the general operation being performed.
- *While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method.*

Created by Dr. Ritu Jain

Example1: Method Overloading



```
// 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;
  }
}

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);
  }
}
```



This program generates the following output:

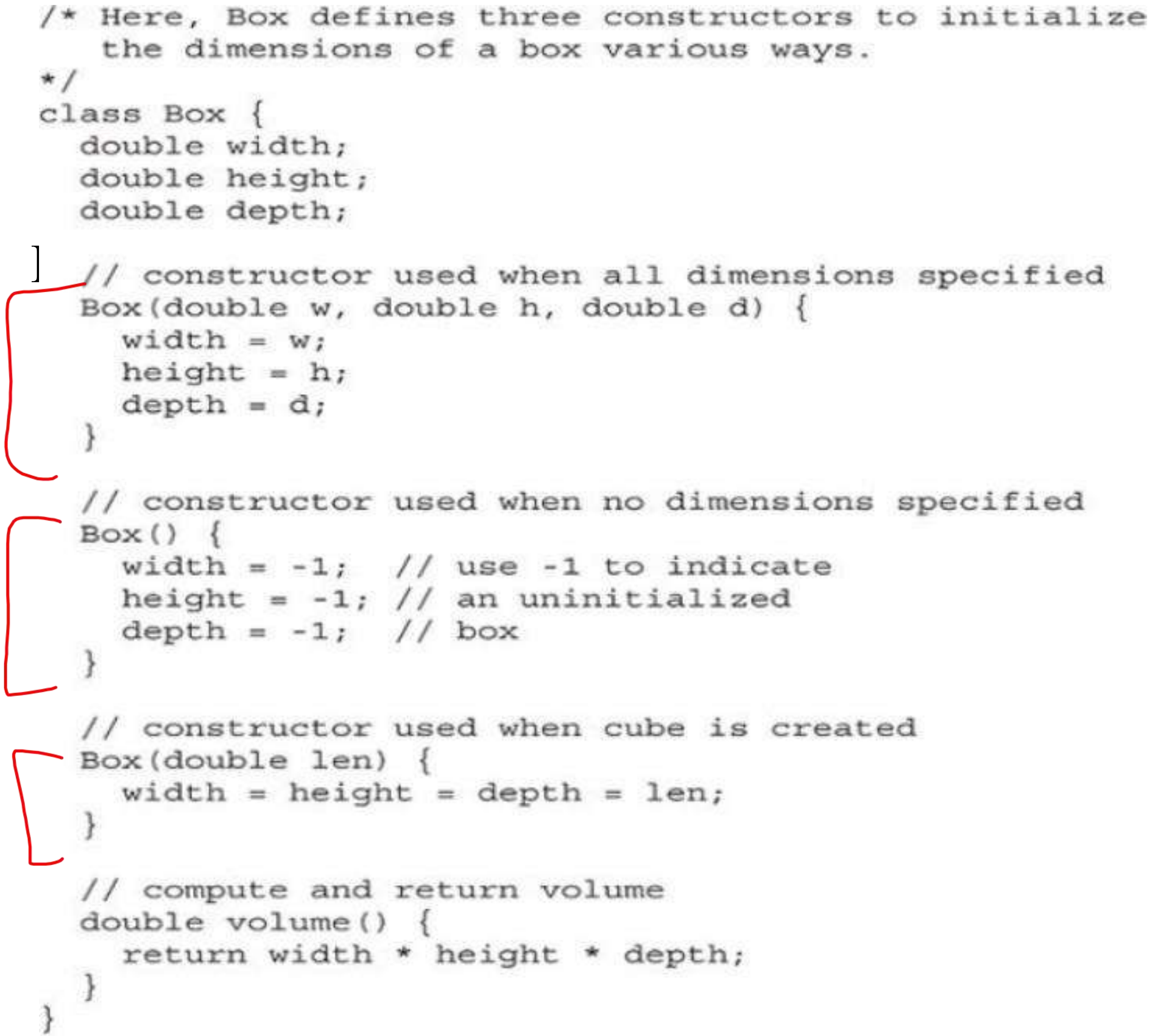
```
No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625
```

Overloading Constructors

- In addition to overloading normal methods, you can also overload constructor methods. In fact, for most real-world classes that you create, overloaded constructors will be the norm, not the exception.

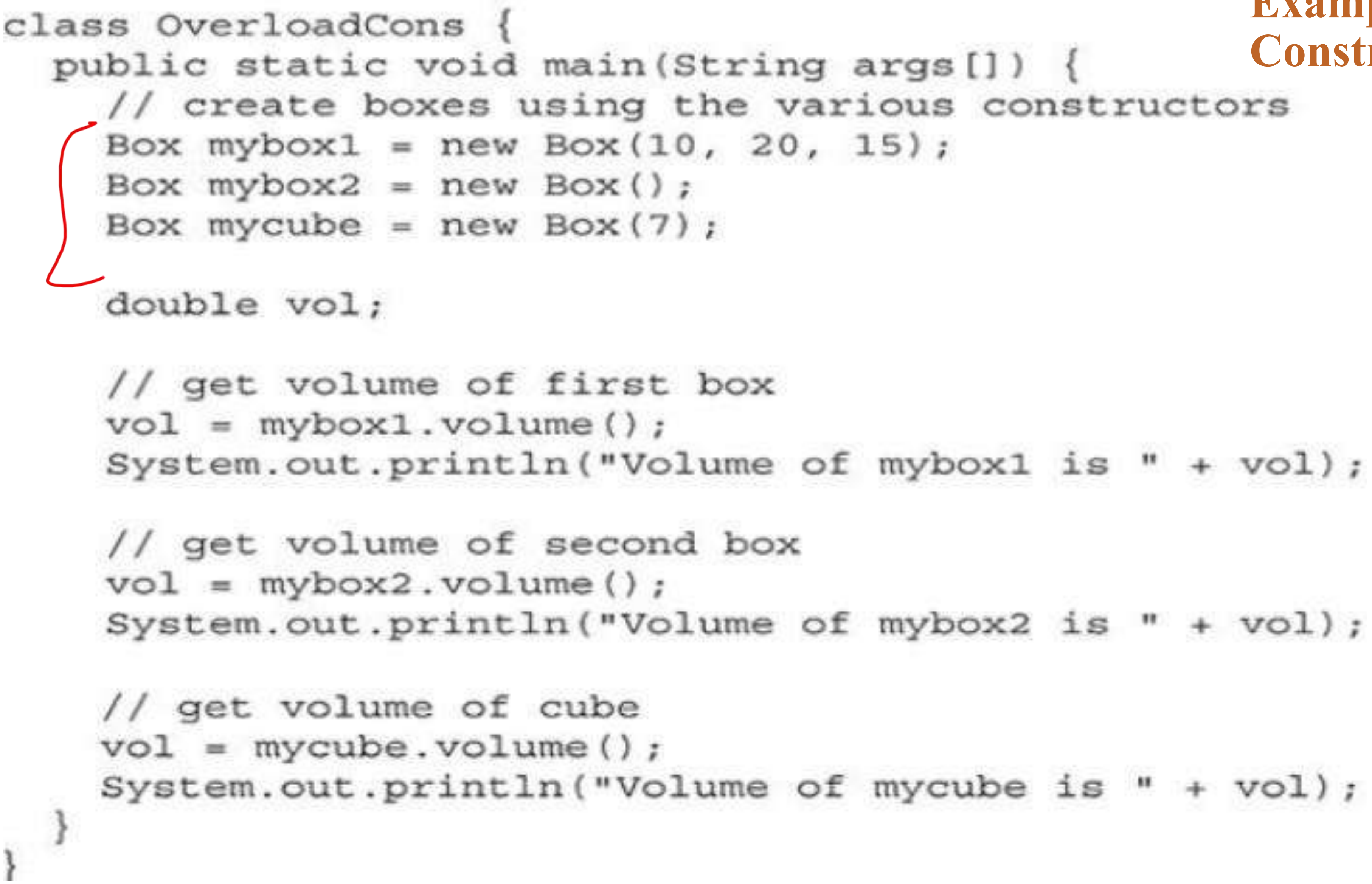
Example (slide 1): Constructor Overloading

OverloadCons.java



```
/* 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;
  }
}
```

Example (slide 2): Constructor Overloading

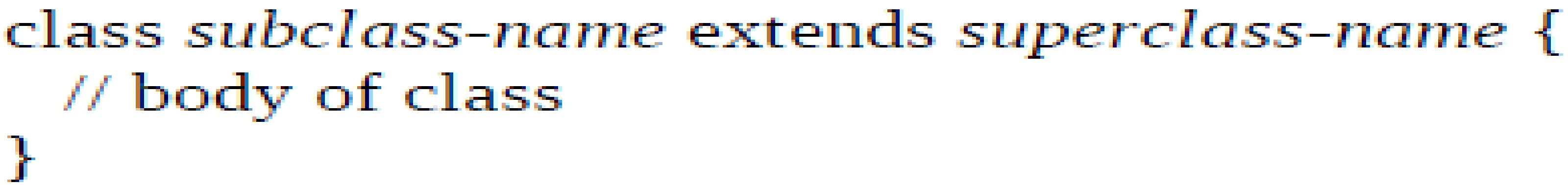


```
class OverloadCons {
  public static void main(String args[]) {
    // create boxes using the various constructors
    Box mybox1 = new Box(10, 20, 15);
    Box mybox2 = new Box();
    Box mycube = new Box(7);

    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);
  }
}
```

Inheritance

Inheritance

- **Idea of inheritance:** When you want to create a new class and there is already a class that includes some of the code that you want, you can derive your new class from the existing class. In doing this, you can reuse the fields and methods of the existing class without having to write (and debug!) them yourself.
- To inherit a class, use the **extends** keyword in subclass definition.
- **Syntax:**



```
class subclass-name extends superclass-name {
  // body of class
}
```

- You can only inherit **one superclass** for any subclass.
- Java **does not support the concept of multiple inheritance through class.**
- **Multilevel Inheritance:** You can create a hierarchy of inheritance in which a subclass becomes a superclass of another subclass .
- No class can be a superclass of itself.
- **Private members** of a class are not inherited.
- Every class in java implicitly extends **java.lang.Object** class.

Types of inheritance

- Different types of inheritance which are supported in Java through **classes.**
  - Single Inheritance
  - Multilevel Inheritance
  - Hierarchical Inheritance
  - Hybrid Inheritance

Single Inheritance

- In single inheritance, a sub-class is derived from only one super class. It inherits the properties and behavior of a single-parent class.

Example of single inheritance: InheritEx1.java

```
 class Animal {
   String name;
   public void eat() {
        System.out.println("I can eat");
   }
}

class Dog extends Animal {
  public void bark() {
        System.out.println(name+”barks”);
  }
}

class InheritEx1 {
     public static void main(String[] args) {
          Dog labrador = new Dog();
          labrador.name = "Rohu";
          labrador.bark();
          labrador.eat();

  }
}
```

Multilevel Inheritance

- **Multilevel Inheritance**
- In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also acts as the base class for other classes.

```
class Animal {// Base class
    void eat() {
        System.out.println("Animal eats food");
    }
}
// Derived class (inherits from Animal)
class Mammal extends Animal {
    void walk() {
        System.out.println("Mammal walks on land");
    }
}
// Further derived class (inherits from Mammal, which inherits from Animal)
class Dog extends Mammal {
    void bark() {
        System.out.println("Dog barks");
    }

    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat();  // inherited from Animal
        dog.walk(); // inherited from Mammal
        dog.bark(); // defined in Dog
    }
}
```

```
Animal eats food
Mammal walks on land
 Dog barks
```

Hierarchical Inheritance

- In hierarchical inheritance, more than one subclass is inherited from a single base class. i.e. more than one derived class is created from a single base class.
- For example, cars and buses both are vehicle

Hierarchical Inheritance

```
class Animal {
    void eat() {
        System.out.println("Animal eats food");
    }
}
class Dog extends Animal {
    void bark() {
        System.out.println("Dog barks");
    }
}
class Cat extends Animal {
    void meow() {
        System.out.println("Cat meows");
    }
}
public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.eat();   // inherited from Animal
        dog.bark();  // defined in Dog
        Cat cat = new Cat();
        cat.eat();   // inherited from Animal
        cat.meow();  // defined in Cat
    }
}
```

Using super keyword

- Whenever a subclass needs to refer to its immediate superclass, it can do so by use of **super**.

- **super** has two uses:
  - to call superclass' constructor.
  - used to access a member (variables as well as methods) of the superclass that has been hidden by a member of a subclass.

- How to use super:
  - super() calls the parent class constructor.
  - super.methodName() calls a method from the parent class.
  - super.variableName accesses a variable from the parent class.

Using super() to Call Superclass Constructors

- A subclass can call a constructor defined by its superclass by use of the following form of super: **super(arg-list);**
  - Here, arg-list specifies any arguments needed by the constructor in the superclass.
- super( ) must always be the first statement executed inside a subclass' constructor.
- When a subclass calls **super()**, it is calling the constructor of its immediate superclass.

Ex: Using super() to Call Superclass Constructors

```
class Vehicle {
    String brand;
    Vehicle(String brand) {
        this.brand = brand;
        System.out.println("Vehicle brand: " + brand);
    }
    void displayBrand() {
        System.out.println("Brand: " + brand);
    }
}
```

```
class Car extends Vehicle {
    int year;
    Car(String brand, int year) {
        super(brand);
        this.year = year;
        System.out.println("Car year: " + year);
    }
    void displayInfo() {
        displayBrand();
        System.out.println("Year: " + year);
    }

    public static void main(String[] args) {
        Car myCar = new Car("Toyota", 2022);
        myCar.displayInfo();
    }
}
```

Using super to overcome name hiding

- The second form of **super** acts somewhat like **this**, except that it always refers to the superclass of the subclass in which it is used.
- Syntax: **super.*member;***
- Here, *member* can be either a method or an instance variable.
- **This second form of super is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.**

Using super to access a parent class variable

```
class Parent {
    int num = 100;
}

class Child extends Parent {
    int num = 200;

    void show() {
        System.out.println(num);         // prints Child's num
        System.out.println(super.num);   // prints Parent's num
    }
}

public class Test {
    public static void main(String[] args) {
        Child c = new Child();
        c.show();
    }
}
```

Output:

```
200
100
```

Using super to access a parent class method

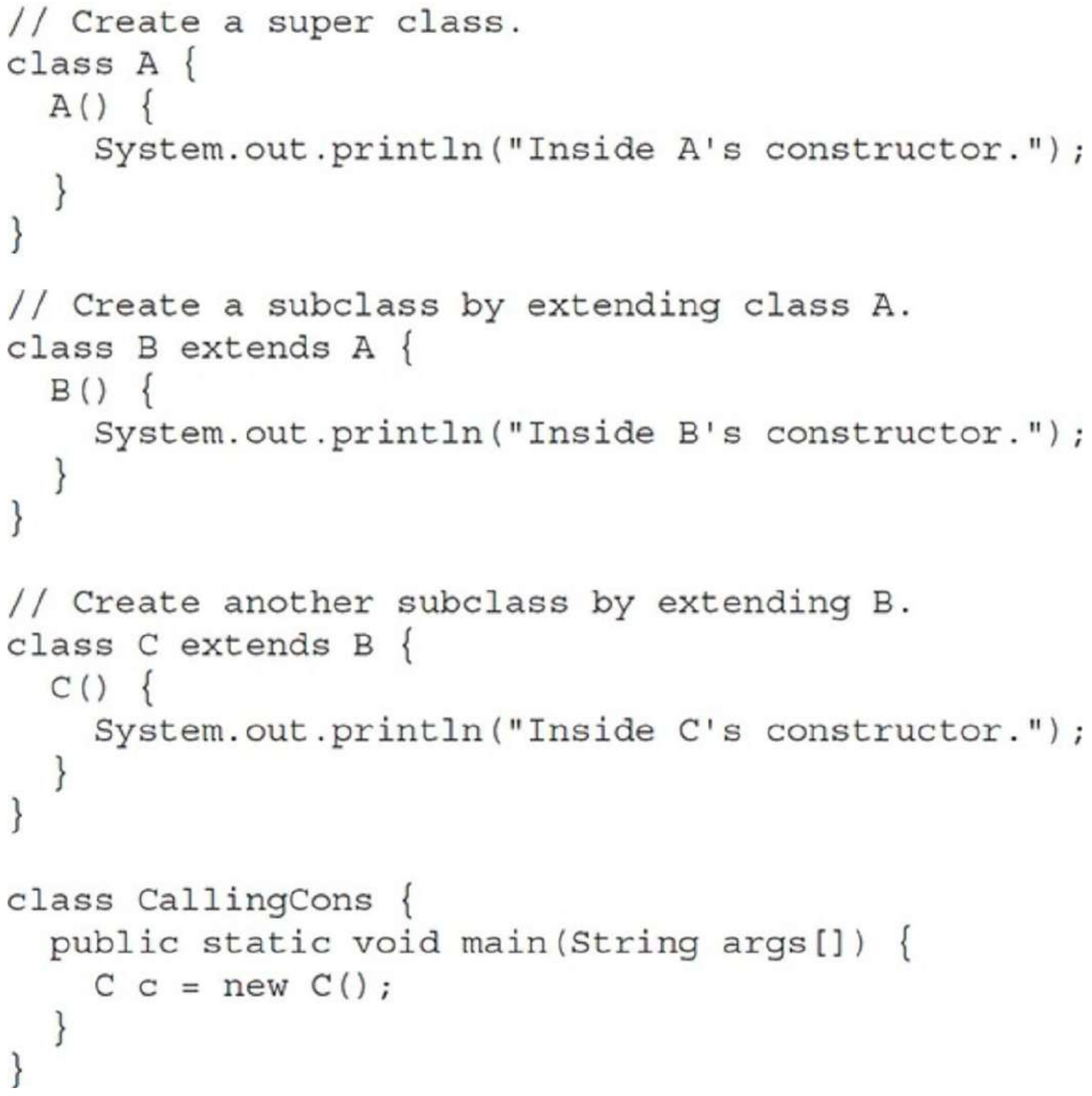
```
class Parent {
    void display() {
        System.out.println("Parent display method");
    }
}

class Child extends Parent {
    void display() {
        super.display();  // call Parent's display()
        System.out.println("Child display method");
    }
}

public class Test {
    public static void main(String[] args) {
        Child c = new Child();
        c.display();
    }
}
```

Output:

```
Parent display method
Child display method
```

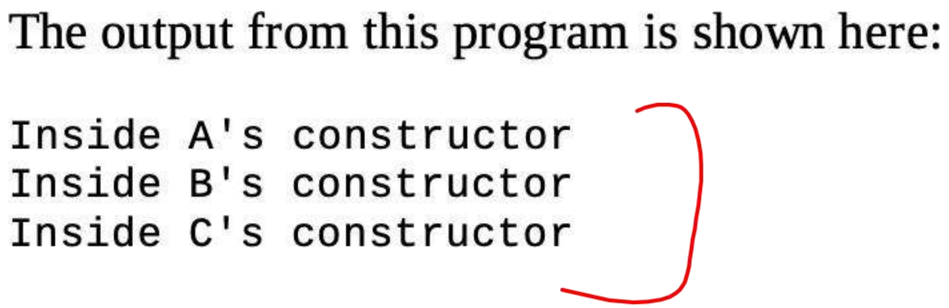
Order of execution of Constructors in Multilevel inheritance

- In a class hierarchy, constructors complete their execution in order of derivation, from superclass to subclass.
- Further, since **super( )** must be the first statement executed in a subclass' constructor.
- If **super( )** is not used, then compiler inserts super() to call parameterless constructor of superclass.



```
// Create a super class.
class A {
  A() {
    System.out.println("Inside A's constructor.");
  }
}

// Create a subclass by extending class A.
class B extends A {
  B() {
    System.out.println("Inside B's constructor.");
  }
}

// Create another subclass by extending B.
class C extends B {
  C() {
    System.out.println("Inside C's constructor.");
  }
}

class CallingCons {
  public static void main(String args[]) {
    C c = new C();
  }
}
```

Order of execution of Constructors in Multilevel inheritance



The output from this program is shown here:

```
Inside A's constructor
Inside B's constructor
Inside C's constructor
```

static

- The keyword **static** in Java means that **the member belongs to the class, not to any specific object** of that class.
- It can be used for creating static:
  - **static variables**
  - **static methods**
  - **static blocks**

- **Need of static:**
  - If you want to define a member in a class that will be used independently of any object of that class.
- **How to create static member:** To create such a member, precede its declaration with the keyword **static**.

Static variable

Static variables:

- A **static variable** (also called a **class variable**) is a variable that **belongs to the class**, not to any specific object.
- That means **only one copy** of the variable exists in memory — shared by **all objects** of that class.
- **Syntax to declare static variable:**

```
class ClassName {
        static dataType variableName;
}
```

Syntax to access static variable outside the class:

```
classname.staticvarname
```

Static variable

```
class Student {
    static String college = "ABC University"; // static variable
    String name; // instance variable
    Student(String n) {
        name = n;
    }
    void display() {
        System.out.println(name + " - " + college);
    }
}
public class StaticVarEx1 {
    public static void main(String[] args) {
        System.out.println(Student.college );
        Student s1 = new Student("Riya");
        Student s2 = new Student(“Ravi");
        s1.display();
        s2.display();
        Student.college = "XYZ Institute";
        s1.display();
        s2.display();
    }
}
```

Output:

```
ABC University
Riya - ABC University
Ravi - ABC University
Riya - XYZ Institute
Ravi – XYZ Institute
```

Exaplanation:

- college is declared **static**, so both s1 and s2 share it.
- When we change Student.college, it affects **all objects** — because there’s **only one copy** in memory.

Key Points About Static Variables

Feature	Description
Belongs to	The class, not to any individual object
Memory Allocation	Done only once, when the class is loaded
Shared by	All objects of the class
Accessed using	Class name (recommended) — classname.staticvariablename e.g., Student.college
Lifetime	Exists until the class is unloaded from memory

When to Use Static Variables

- Use static variables when:
  - You want to **share common data** across all objects (e.g., school name, company name).
  - You need a **counter** to track the number of objects created.

Object Counter using static variable

```
class Counter {
    static int count = 0; // static variable
    Counter() {
        count++; // increment when object is created
    }
}
public class Main {
    public static void main(String[] args) {
        new Counter();
        new Counter();
        new Counter();

        System.out.println("Number of objects created: " + Counter.count);
    }
}
```

Static methods

- **Static methods:** A **static method** is a method that **belongs to the class**, not to any specific object. You can call it **without creating an object** of the class.
- It's declared using the static keyword.
- **Syntax for creating static methods :**

```
 class ClassName {
     static returnType methodName() {
         // method body
     }
 }
```

- Methods declared as **static** have several restrictions:
  - They can only directly call other **static** methods of their class.
  - They can only directly access **static** variables of their class.
  - They cannot refer to **this** or **super** in any way.

Calling Static method of the same class

```
class StaticMethodEx1 {
    static void display() {
        System.out.println("This is a static method.");
    }

    public static void main(String[] args) {
            StaticMethodEx1.display();
            display();
    }
}
```

Calling Static method of the different class

```
class MathUtils {
    static int square(int x) {
        return x * x;
    }
}

public class StaticMethodEx2 {
    public static void main(String[] args) {
        System.out.println(MathUtils.square(5)); // called without object
    }
}
```

Key Points About Static Methods

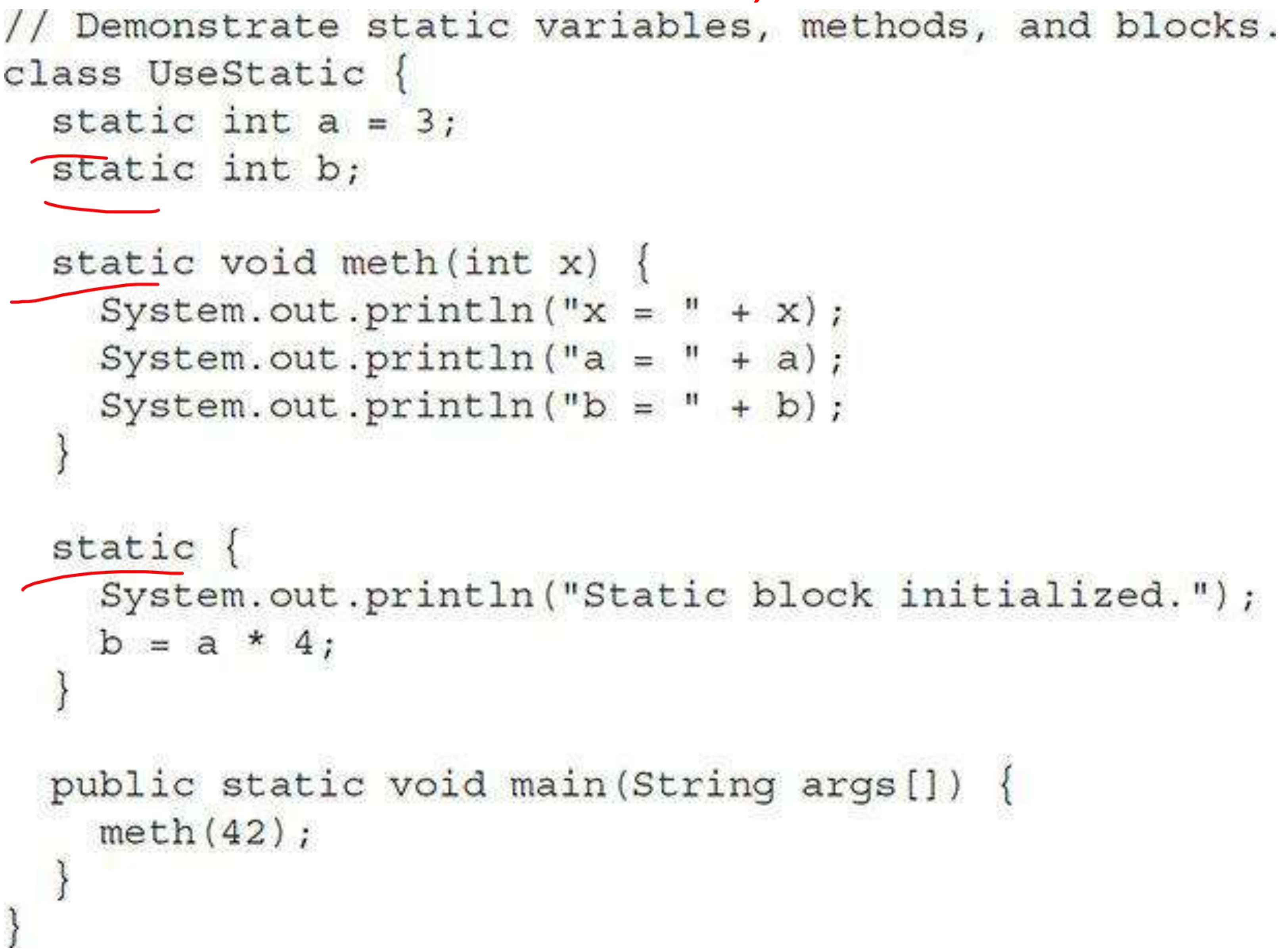
Feature	Description
Belongs to	The class (not to objects)
Accessed by	Class name (e.g., ClassName.methodName())
Called without object	Yes
Can access	Only static variables and other static methods directly
Cannot access	Instance variables or instance methods directly
Memory allocation	Done when the class is loaded
Use Case	Utility or helper methods (e.g., Math.sqrt(), Collections.sort())

Static Blocks

- **Static Blocks**
- Used to **initialize static variables**.
- Runs **once** when the class is loaded.

- **Need of static block:**
- 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.

Static Variable, static methods, static block



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

Output:

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

Example: Static vs Non-static Access

```
class Student {
    int rollNo;                 // instance variable
    static String college = "ABC College"; // static variable

    static void changeCollege() {
        college = "XYZ University"; // allowed (accessing static variable)
    }

    void display() {
        System.out.println(rollNo + " " + college);
    }
}
public class Main {
    public static void main(String[] args) {
        Student.changeCollege(); //  call static method without object

        Student s1 = new Student();
        s1.rollNo = 101;
        s1.display();
    }
}
```

Output:

```
101 XYZ University
```

Why Static Methods Can’t Access Instance Members Directly

- Because instance members belong to **objects**, and when a static method runs, **no object may exist** yet.

```
class Example {
    int x = 10;


    static void show() {
        // System.out.println(x);   Error: Cannot access non-static variable x
    }}
```

- **To fix it, you must create an object:**
  - Example e = new Example();
  - System.out.println(e.x);

- The main() method in Java is **always static** so the JVM can call it **without creating an object** of the class.

Array of Objects

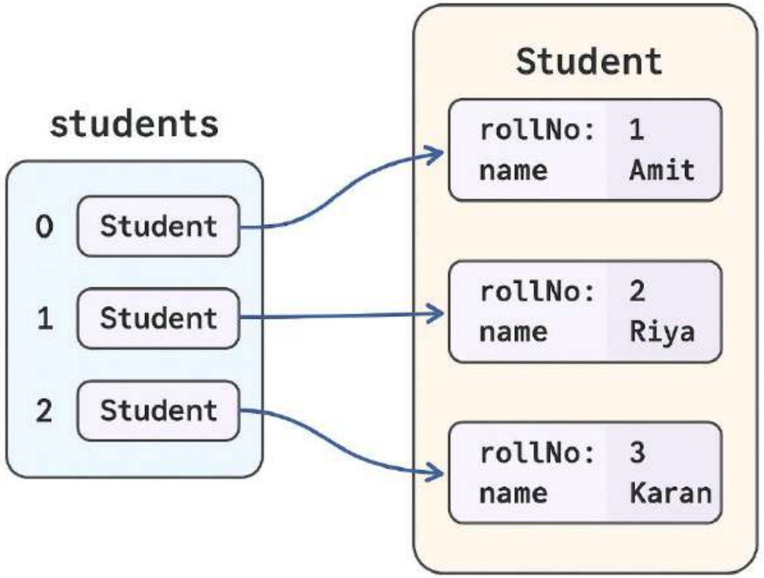
- **Definition:** An **array of objects** in Java is an array where **each element is an object** of a class, rather than a primitive data type. It allows you to store multiple objects of the same class in a single array structure.
- **Syntax of Array of Objects**
  - // Step 1: Declare an array of class type
  - ClassName[] arrayName;

  - // Step 2: Create the array (allocate memory for references)
  - arrayName = new ClassName[size];

  - // Step 3: Create individual objects and assign them to array elements
  - arrayName[index] = new ClassName(parameters);

- **Example Syntax with Explanation:**

```
Student[] students;            // Declaration of array of Student objects
students = new Student[3];     // Memory allocated for 3 Student references
//or Student[] students = new Student[3];
students[0] = new Student(1, "Amit");   // Object 1 created and assigned
students[1] = new Student(2, "Riya");   // Object 2 created and assigned
students[2] = new Student(3, "Karan");  // Object 3 created and assigned
```



Example Program for Array of Object

```
class Student {
    int rollNo;
    String name;
    Student(int rollNo, String name) {
        this.rollNo = rollNo;
        this.name = name;
    }
    void display() {
        System.out.println("Roll No: " + rollNo + ", Name: " + name);
    }
}
public class ArrayOfObjectsEx {
    public static void main(String[] args) {
        Student[] students = new Student[3]; // Creating an array of Student objects
        students[0] = new Student(1, "Amit");
        students[1] = new Student(2, "Riya");
        students[2] = new Student(3, "Karan");

        for (int i = 0; i < students.length; i++) {
            students[i].display();
        }
    }}
```

Arrays of objects

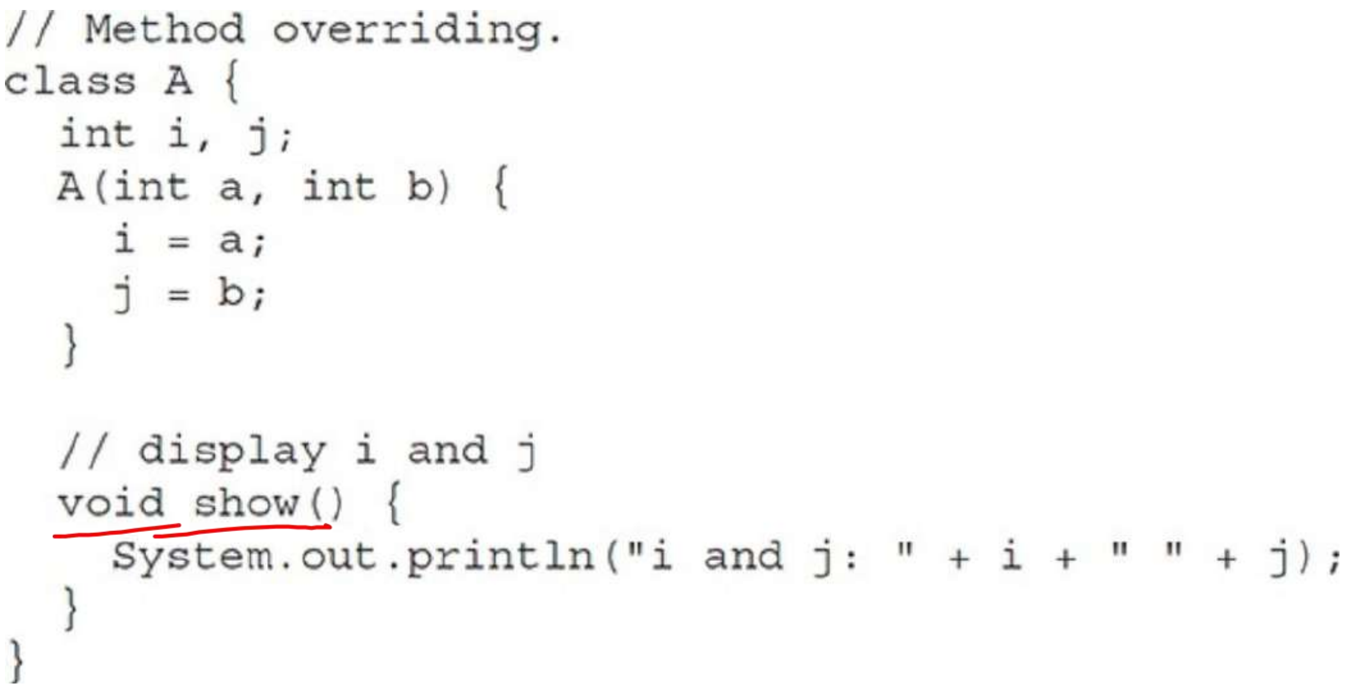
- Arrays of objects store references, not the objects themselves.
- Each element must be instantiated separately.
- Useful for managing collections of related data, like student lists, employee records, etc.

Method Overriding

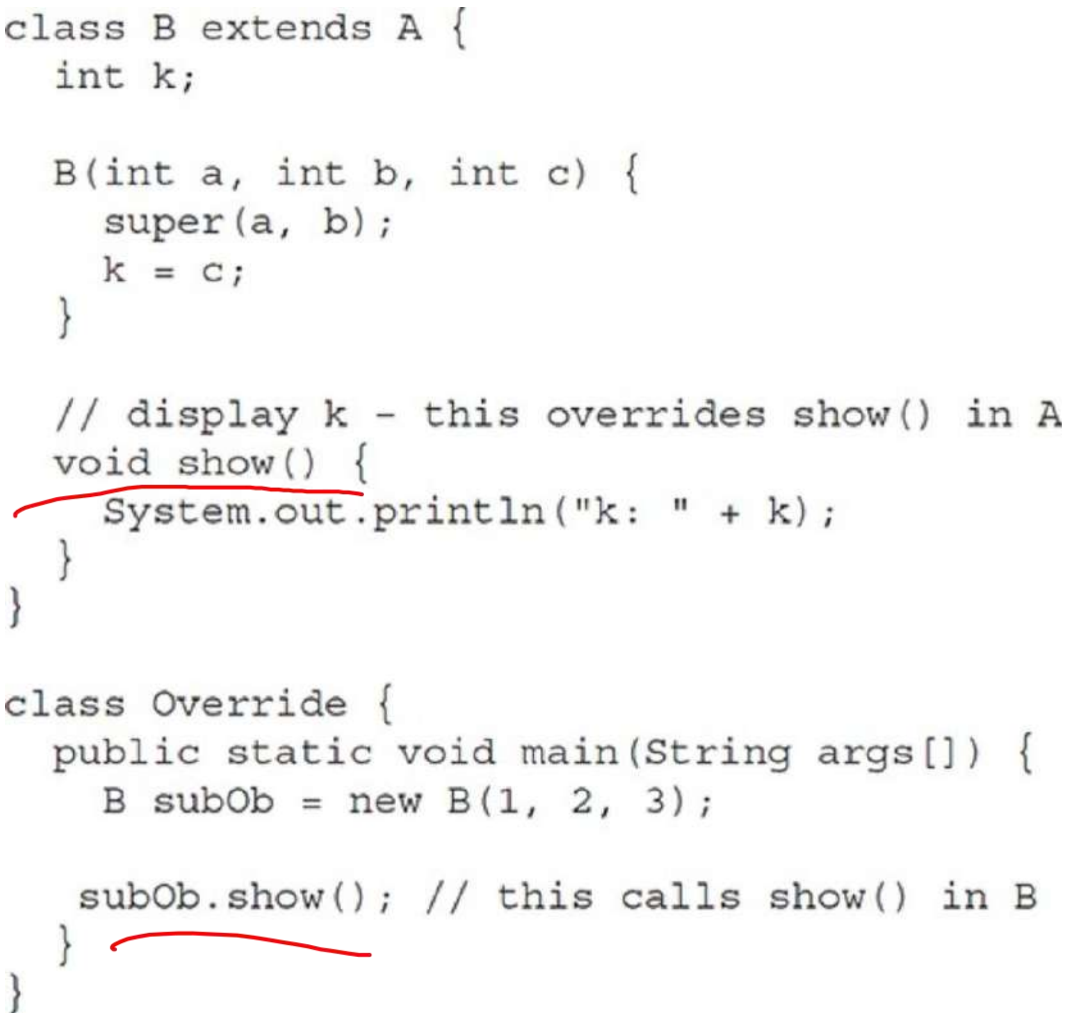
- When a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass.

- When an overridden method is called from within its subclass, it will always refer to the version of that method defined by the subclass.

Example of Method Overriding: Override.java



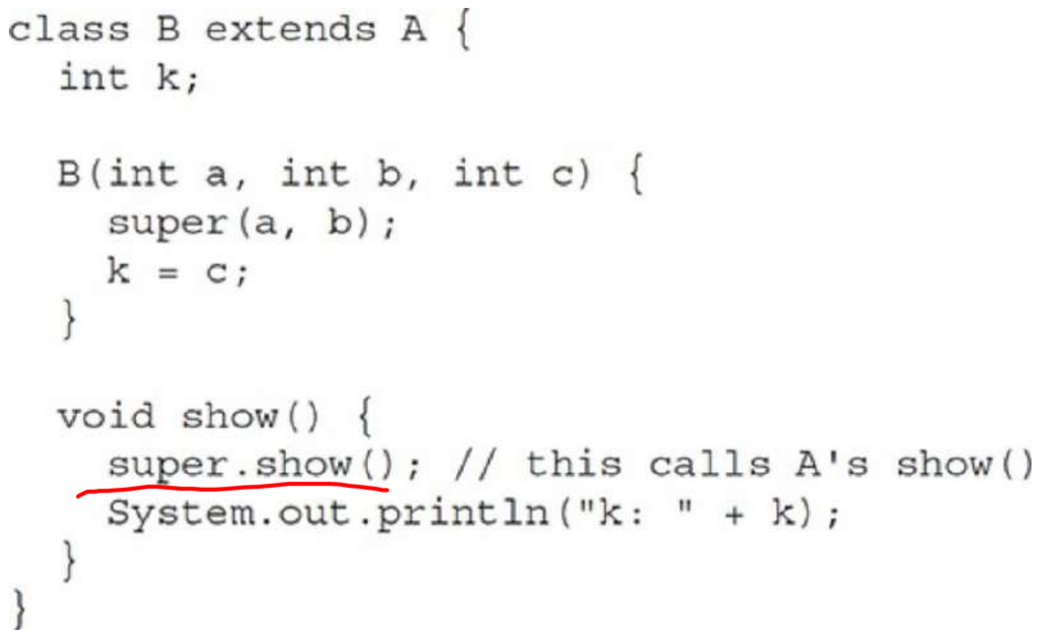
```
// Method overriding.
class A {
  int i, j;
  A(int a, int b) {
    i = a;
    j = b;
  }

  // display i and j
  void show() {
    System.out.println("i and j: " + i + " " + j);
  }
}
```



```
class B extends A {
  int k;

  B(int a, int b, int c) {
    super(a, b);
    k = c;
  }

  // display k - this overrides show() in A
  void show() {
    System.out.println("k: " + k);
  }
}

class Override {
  public static void main(String args[]) {
    B subOb = new B(1, 2, 3);

   subOb.show(); // this calls show() in B
  }
}
```

Method Overriding and use of super

- If you wish to access the superclass version of an overridden method, you can do so by using **super**.



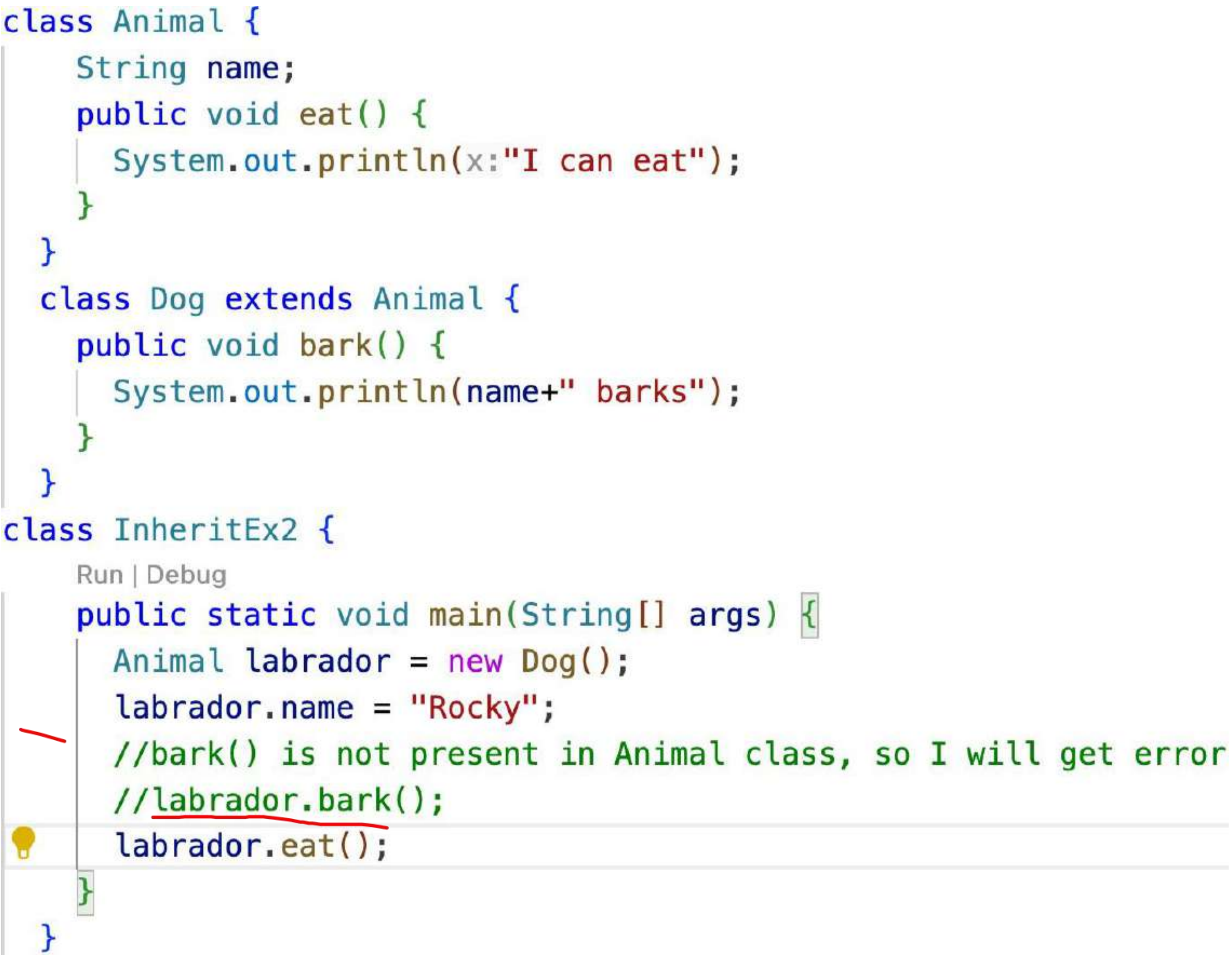
```
class B extends A {
  int k;

  B(int a, int b, int c) {
    super(a, b);
    k = c;
  }

  void show() {
    super.show(); // this calls A's show()
    System.out.println("k: " + k);
  }
}
```

A Superclass Variable Can Reference a Subclass Object

- A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass.
- It is the *type of the reference variable—not the type of the object that it refers to—that determines what members can be accessed*.
- That is, when a reference to a subclass object is assigned to a superclass reference variable, *you will have access only to those members of the object defined by the superclass*. This is why **labrador** can’t access **bark()**

A Superclass Variable Can Reference a Subclass Object

- A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass.



```
class Animal {
    String name;
    public void eat() {
      System.out.println(x:"I can eat");
    }
  }
  class Dog extends Animal {
    public void bark() {
      System.out.println(name+" barks");
    }
  }
class InheritEx2 {
    Run | Debug
    public static void main(String[] args) {
      Animal labrador = new Dog();
      labrador.name = "Rocky";
      //bark() is not present in Animal class, so I will get error
      //labrador.bark();
      labrador.eat();
    }
  }
```

Run time Polymorphism using Method Overriding

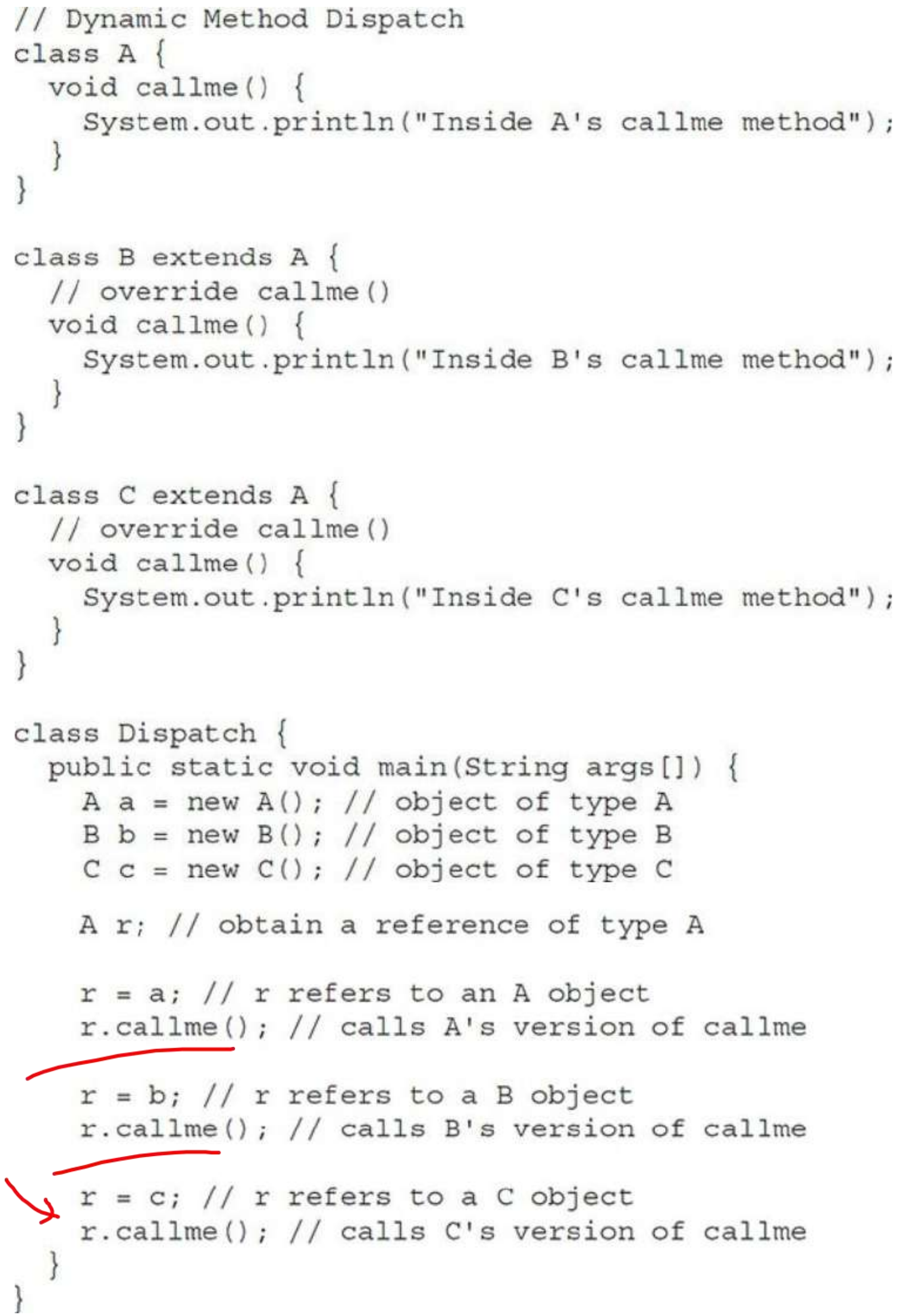
- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- It is important because this is how Java implements run-time polymorphism.
- When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon **the type of the object being referred to at the time the call occurs**. Thus, this determination is made at run time.

Run-time Polymorphism

- **Run-time polymorphism** (also called **dynamic polymorphism**) in Java occurs **when a method call is resolved at runtime** rather than compile time.
- It is achieved through **method overriding** — when a subclass provides a specific implementation of a method that is already defined in its superclass.
-

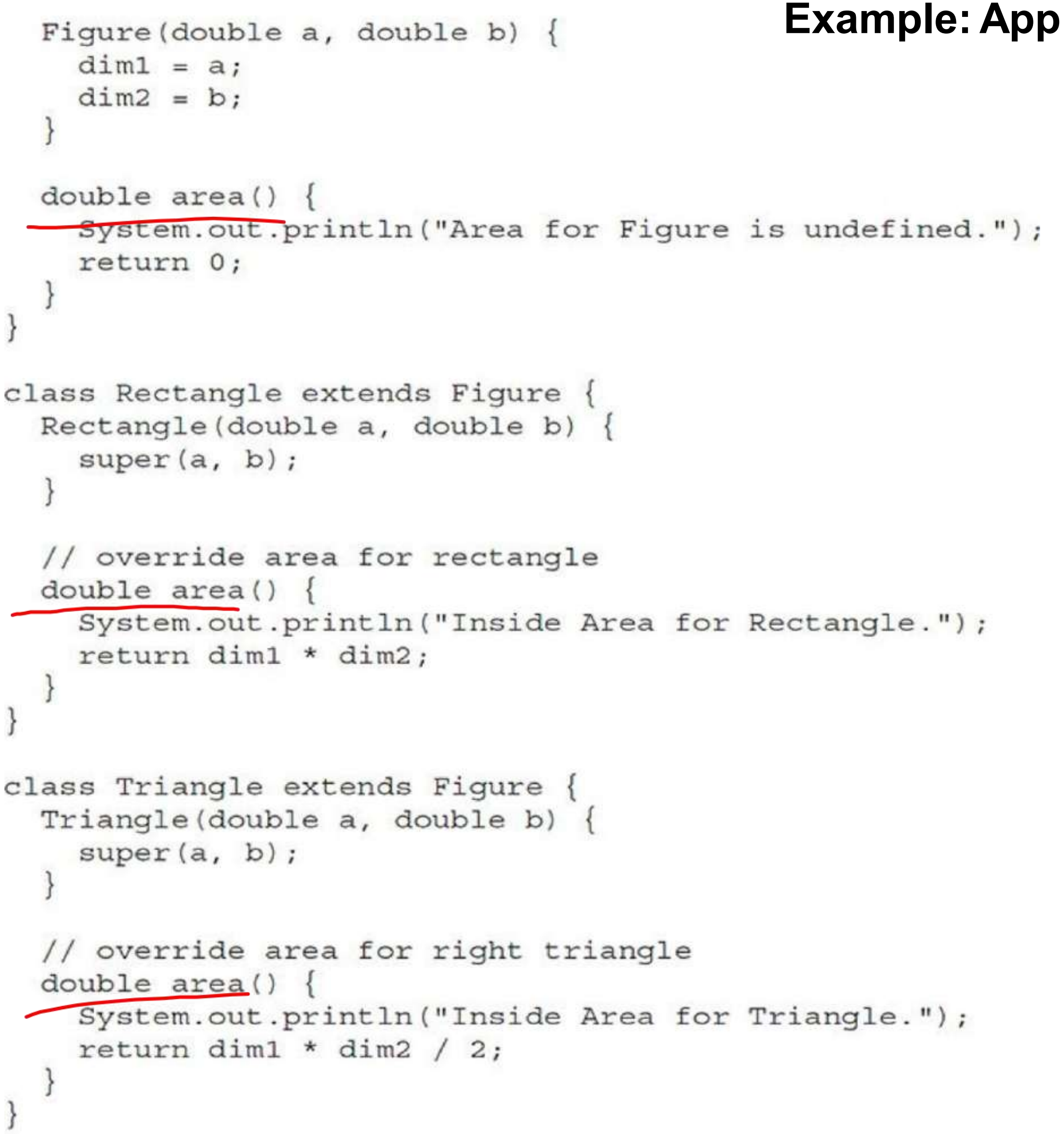
- **Key Concept:**
  - The **reference variable** of the parent class refers to the **object** of the child class.
  - The method to be executed is determined **at runtime** based on the **object** being referred to.

Example of Run time Polymorphism using Method Overriding

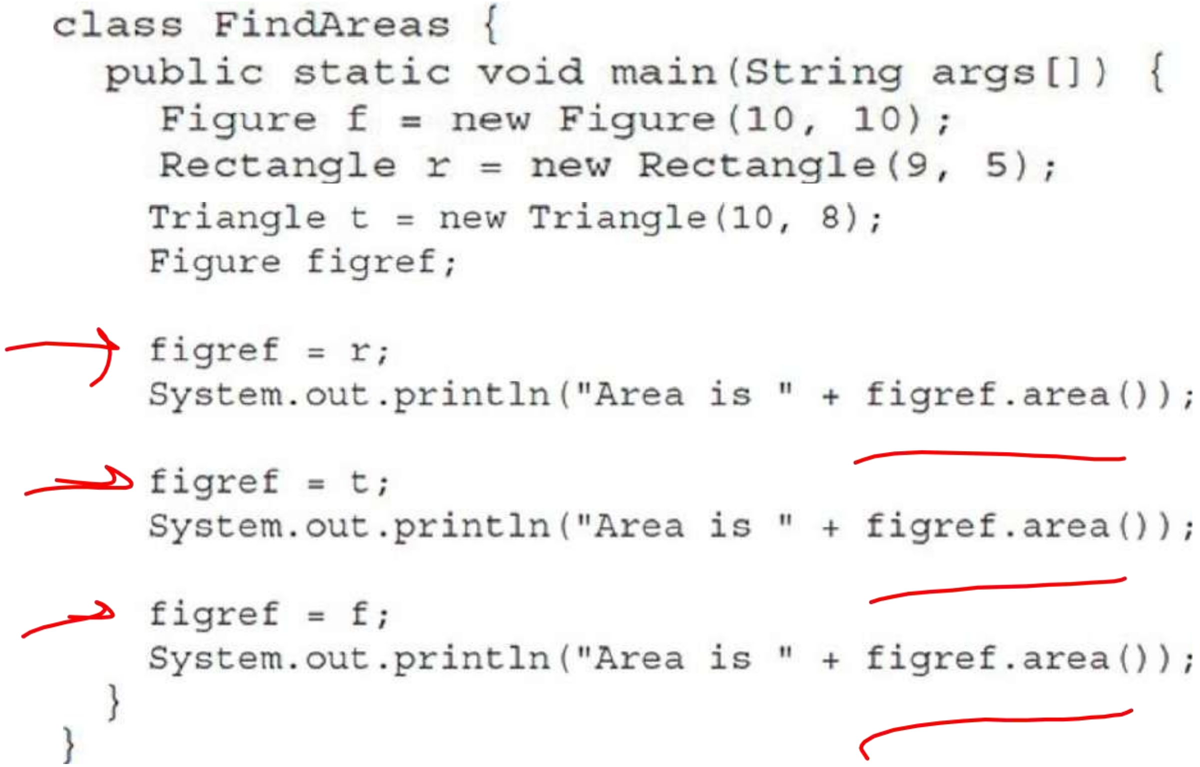


```
// Dynamic Method Dispatch
class A {
  void callme() {
    System.out.println("Inside A's callme method");
  }
}

class B extends A {
  // override callme()
  void callme() {
    System.out.println("Inside B's callme method");
  }
}

class C extends A {
  // override callme()
  void callme() {
    System.out.println("Inside C's callme method");
  }
}

class Dispatch {
  public static void main(String args[]) {
    A a = new A(); // object of type A
    B b = new B(); // object of type B
    C c = new C(); // object of type C

    A r; // obtain a reference of type A

    r = a; // r refers to an A object
    r.callme(); // calls A's version of callme

    r = b; // r refers to a B object
    r.callme(); // calls B's version of callme

    r = c; // r refers to a C object
    r.callme(); // calls C's version of callme
  }
}
```

Example: Applying Method Overriding



```
  Figure(double a, double b) {
    dim1 = a;
    dim2 = b;
  }

  double area() {
    System.out.println("Area for Figure is undefined.");
    return 0;
  }
}

class Rectangle extends Figure {
  Rectangle(double a, double b) {
    super(a, b);
  }

  // override area for rectangle
  double area() {
    System.out.println("Inside Area for Rectangle.");
    return dim1 * dim2;
  }
}

class Triangle extends Figure {
  Triangle(double a, double b) {
    super(a, b);
  }

  // override area for right triangle
  double area() {
    System.out.println("Inside Area for Triangle.");
    return dim1 * dim2 / 2;
  }
}
```



```
class FindAreas {
  public static void main(String args[]) {
    Figure f = new Figure(10, 10);
    Rectangle r = new Rectangle(9, 5);
    Triangle t = new Triangle(10, 8);
    Figure figref;

    figref = r;
    System.out.println("Area is " + figref.area());

    figref = t;
    System.out.println("Area is " + figref.area());

    figref = f;
    System.out.println("Area is " + figref.area());
  }
}
```

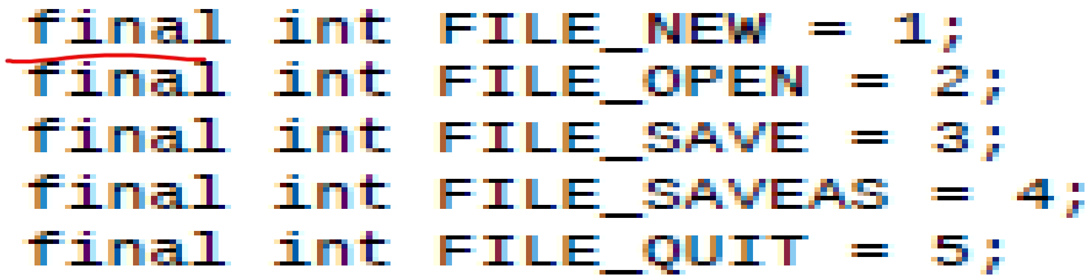
```
Inside Area for Rectangle.
Area is 45
Inside Area for Triangle.
Area is 40
Area for Figure is undefined.
Area is 0
```

final

- The **final** keyword in Java used to restrict users from **modifying** variables, methods, or classes.
- It helps in making code **more secure, stable, and predictable**.
- In Java, we can have:
  - final variable
  - final method
  - final class

Using final for fields

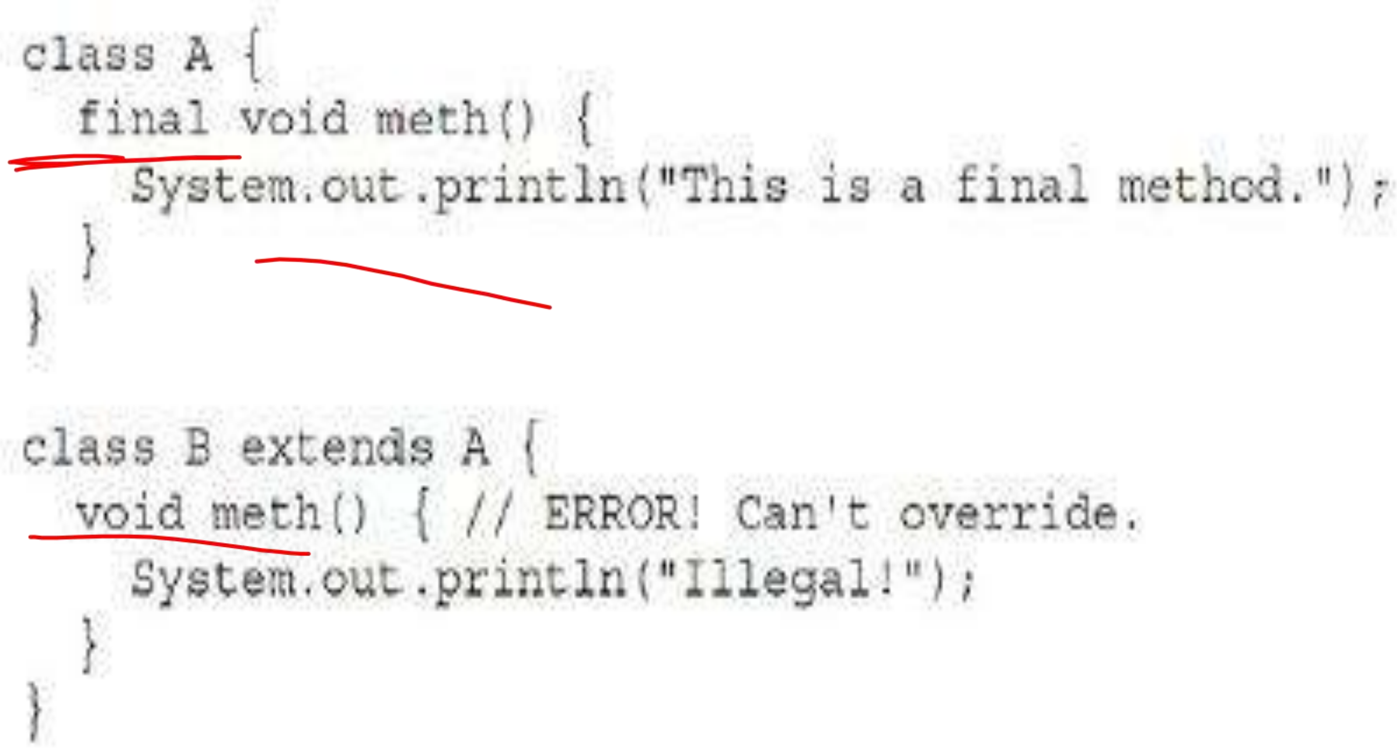
- A field can be declared as **final**. Doing so prevents its contents from being modified, making it, essentially, a constant.
- This means that you must initialize a **final** field when it is declared. You can do this in one of two ways:
  - First, you can give it a value when it is declared.
  - Second, you can assign it a value within a constructor.
- In addition to fields, both method parameters and local variables can be declared **final**.
  - Declaring a parameter **final** prevents it from being changed within the method.
  - Declaring a local variable **final** prevents it from being assigned a value more than once.



```
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;
```

Using final Methods to Prevent Overriding

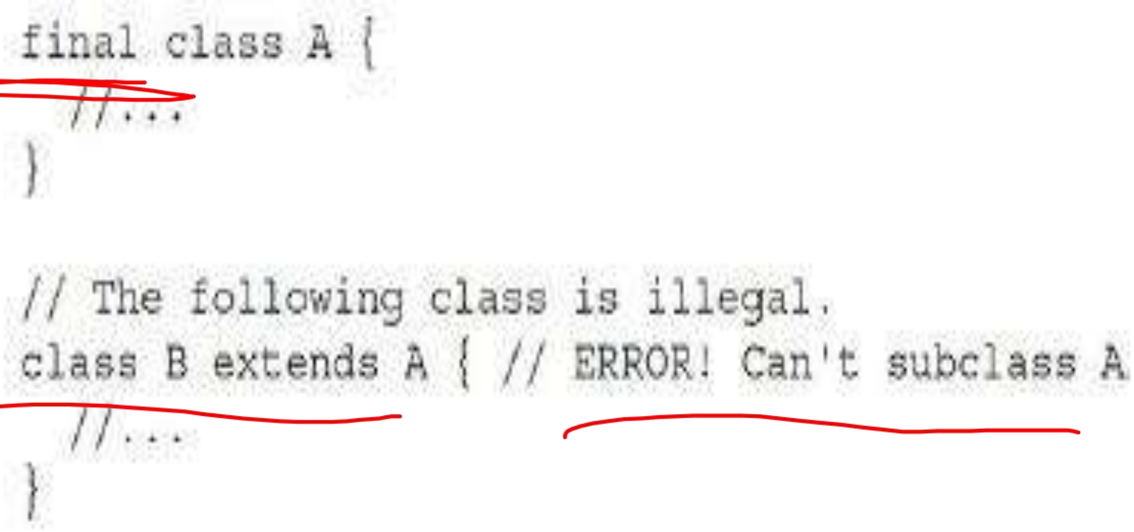
- While method overriding is one of Java’s most powerful features, there will be times when you will want to prevent it from occurring. To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration. Methods declared as **final** cannot be overridden.



```
class A {
  final void meth() {
    System.out.println("This is a final method.");
  }
}

class B extends A {
  void meth() { // ERROR! Can't override.
    System.out.println("Illegal!");
  }
}
```

Using final Classes to Prevent Inheritance

- Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with **final**.
- Declaring a class as **final** implicitly declares all of its methods as **final**, too.
- It is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.



```
final class A {
  //...
}

// The following class is illegal.
class B extends A { // ERROR! Can't subclass A
  //...
}
```

- Thanks!