# Advanced Java Programming

**Text Book : “The Complete Reference Java”**

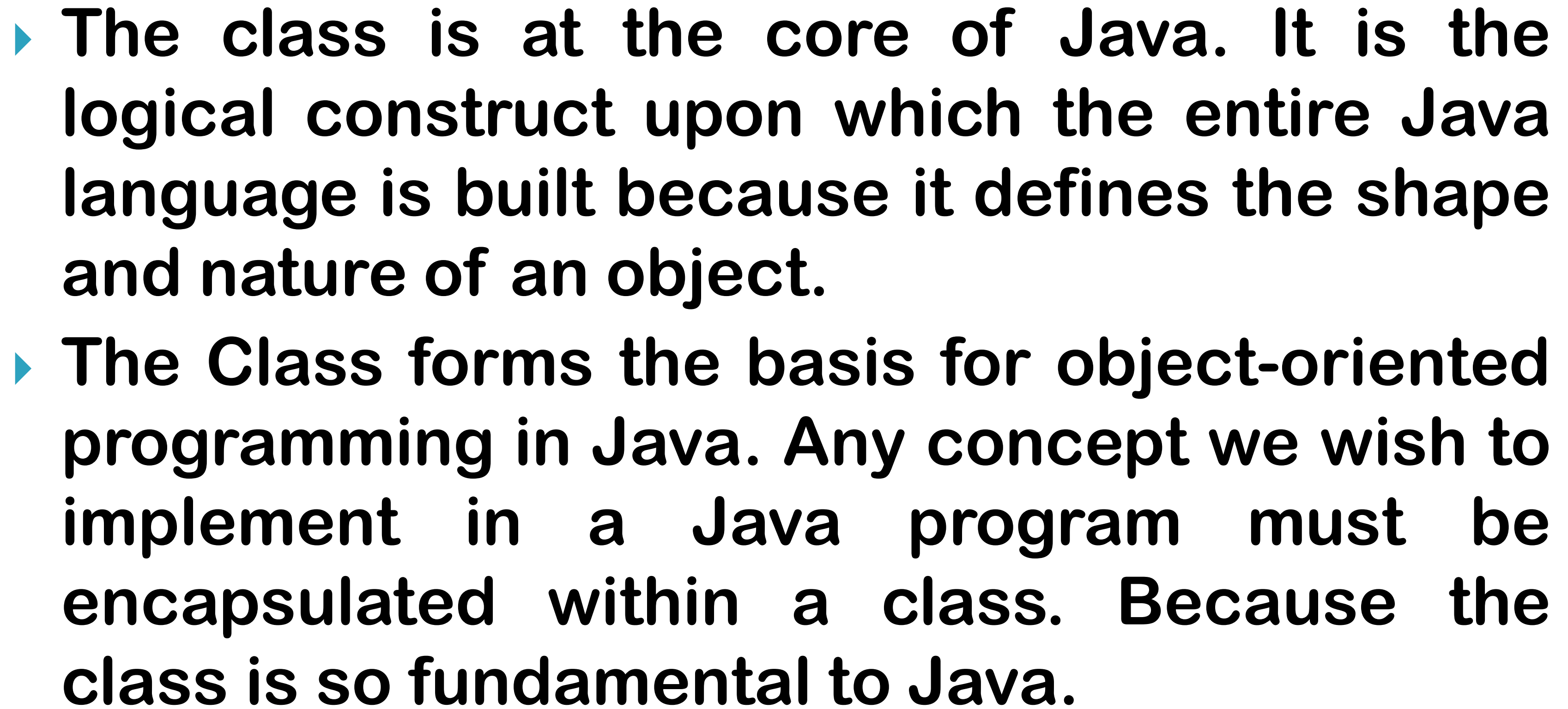
**Herbert Schildt**

**Prepared by : B.Loganathan**

# ADVANCED JAVA PROGRAMMING

- **UNIT I:** Introducing classes: class fundamentals – declaring objects, methods, constructors - this keyword – Method overloading - garbage collection - finalize () method. Inheritance: Inheritance basics – using super – method overriding – dynamic method dispatch – abstract class – final keyword. Packages and interfaces: packages – importing packages – defining interface – implementing interfaces - extending interfaces.
- Text Book : Herbert Schildt, “The Complete Reference Java”, 7th Edition Tata McGraw-Hill Pub., Company Ltd.

# Introducing Classes:



- The class is at the core of Java. It is the logical construct upon which the entire Java language is built because it defines the shape and nature of an object.
- The Class forms the basis for object-oriented programming in Java. Any concept we wish to implement in a Java program must be encapsulated within a class. Because the class is so fundamental to Java.

# Class Fundamentals

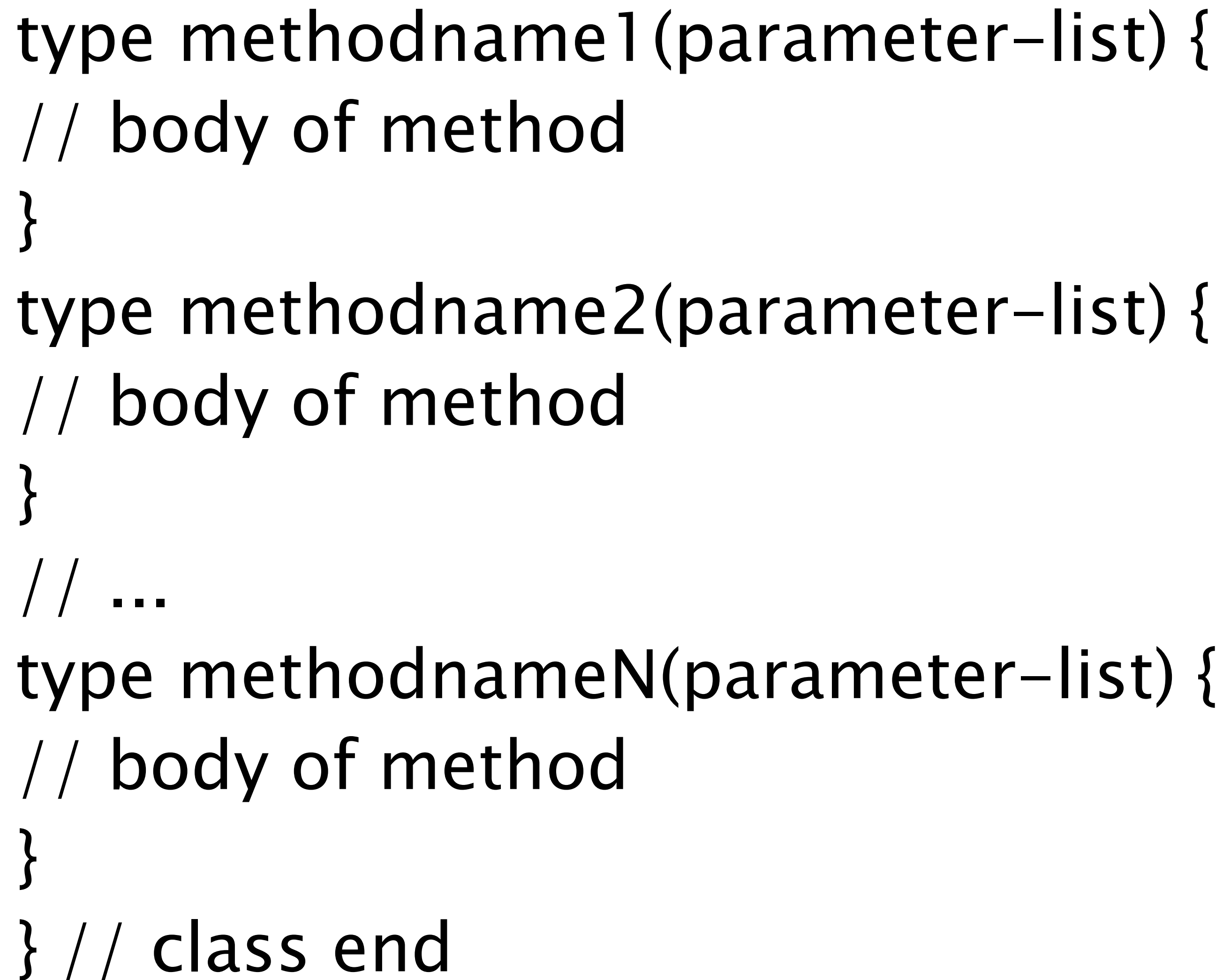
- **The most important thing to understand about a class is that it defines a new data type. Once defined, this new type can be used to create objects of that type.**
- **A class is a template for an object, and an object is an *instance* of a class.**

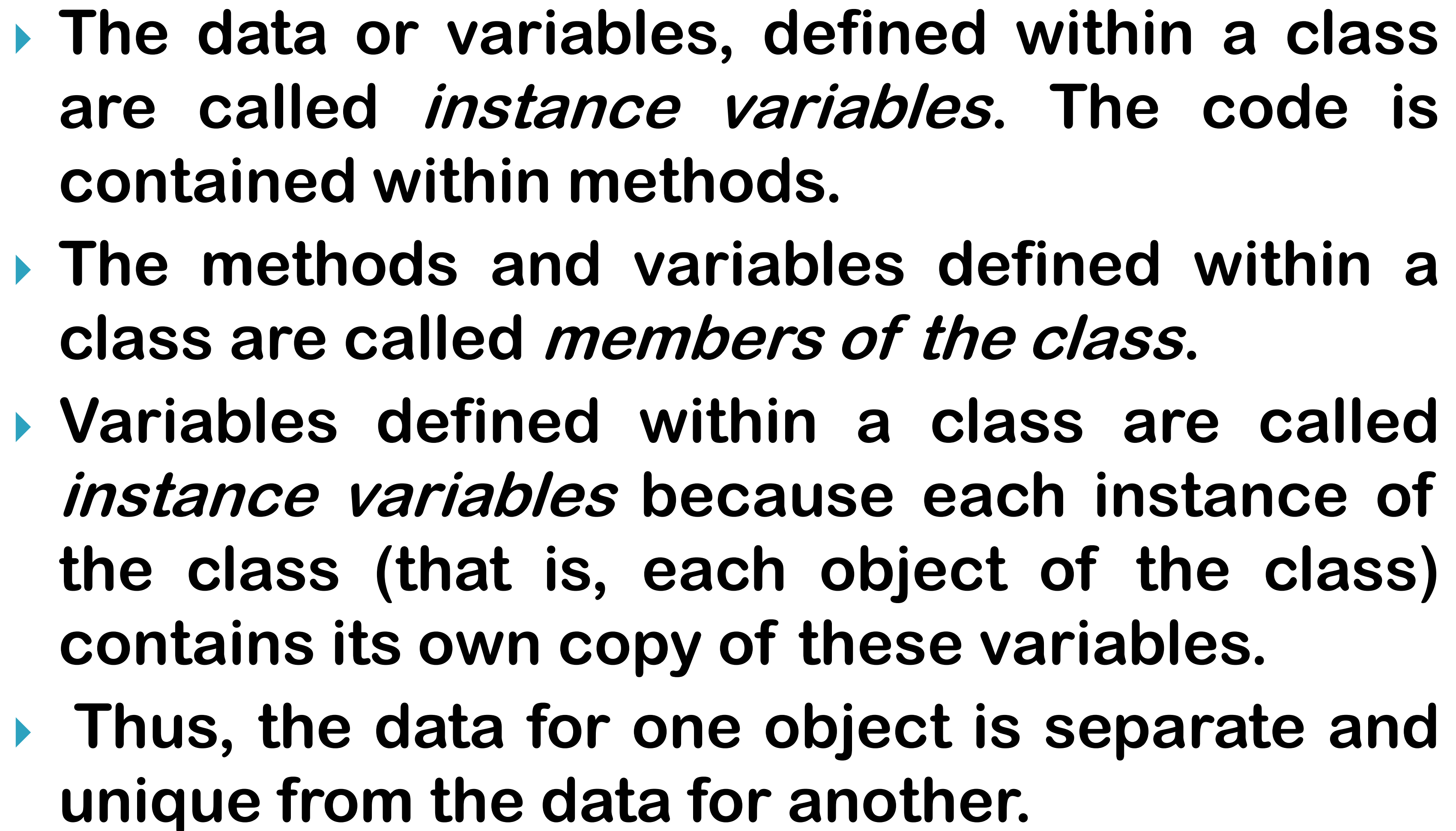
## General Form of a Class :

- **When we define a class, we declare its exact form and nature. we do this by specifying the data that it contains and the code that operates on that data.**

- **A class is declared by use of the class keyword. The classes that have been used up to this point are actually very limited examples of its complete form. Classes can (and usually do) get much more complex. A simplified general form of a class definition is shown here:**

```
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 end



- The data or variables, defined within a class are called *instance variables*. The code is contained within methods.
- The methods and variables defined within a class are called *members of the class*.
- Variables defined within a class are called *instance variables* because each instance of the class (that is, each object of the class) contains its own copy of these variables.
- Thus, the data for one object is separate and unique from the data for another.

## A Simple Class :

- **Let's begin our study of the class with a simple example. Here is a class called Box that defines three instance variables: width, height, and depth. Currently, Box does not contain any methods.**

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

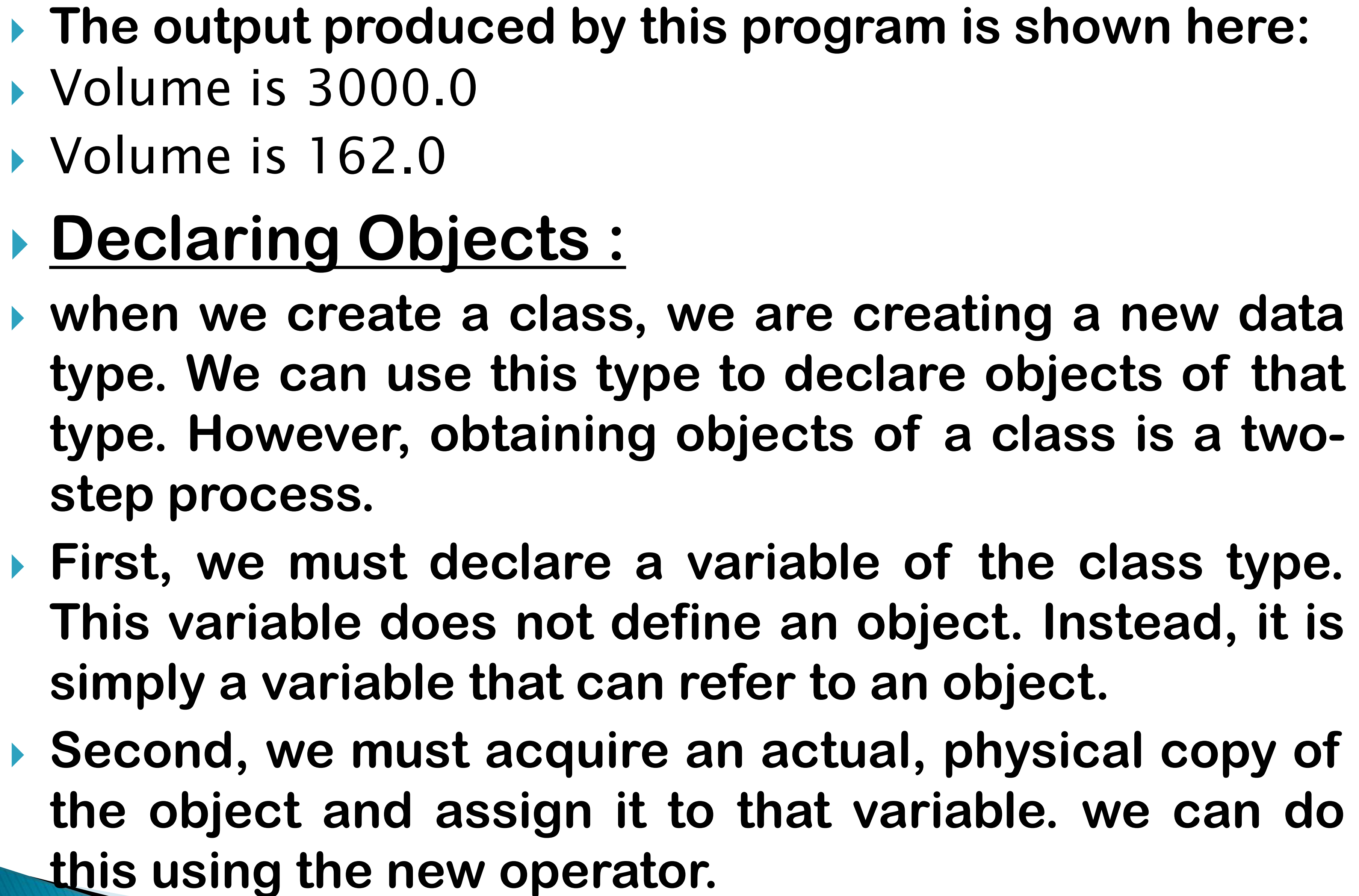
- **A class defines a new type of data. In this case, the new data type is called Box.**
- **To actually create a Box object, we will use a statement like the following:**
- Box mybox = new Box();
- /* create a Box object called mybox */
- **After this statement executes, mybox will be an instance of Box. Thus, it will have physical Reality.**
- **Each time we create an instance of a class, we are creating an object that contains its own copy of each instance variable defined by the class.**

- **Thus, every Box object will contain its own copies of the instance variables width, height, and depth. To access these variables, we will use the dot(.) operator.**
- **The dot operator links the name of the object with the name of an instance variable. For example, to assign the width variable of mybox the value 100, we would use the following statement:**
- mybox.width = 100;
- **This statement tells the compiler to assign the copy of width that is contained within the mybox object the value of 100.**

- Here is a complete program that uses the Boxclass:
- /* A program that uses the Box class.
- Call this file BoxDemo.java
- */
- class Box {
- double width;
- double height;
- double depth;
- }
- // This class declares an object of type Box.
- class BoxDemo {
- public static void main(String args[]) {
- Box mybox = new Box();
- double vol;
- // assign values to mybox's instance variables
- mybox.width = 10;
- mybox.height = 20;
- mybox.depth = 15;
- // compute volume of box
- vol = mybox.width * mybox.height * mybox.depth;
- System.out.println("Volume is = " + vol);
- }
- }

- **we should call the file that contains this program BoxDemo.java, because the main()method is in the class called BoxDemo, not the class called Box. When we compile this program, we will find that two .class files have been created, one for Box and one for BoxDemo.**
- **To run this program, we must execute BoxDemo.class and get the following output:**
- Volume is =3000.0
- **Each object has its own copies of the instance variables. This means that if we have two Boxobjects, each has its own copy of depth, width, and height. It is important to understand that changes to the instance variables of one object have no effect on the instance variables of another.**

- // 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
- **<u>Declaring Objects :</u>**
- **when we create a class, we are creating a new data type. We can use this type to declare objects of that type. However, 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. Instead, 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. we can do this using the new operator.**

- **The *new* operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by new. This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated.**
- **In the preceding sample programs, a line similar to the following is used to declare an object of type Box:**
- Box mybox = new Box();
- **This statement combines the two steps just described.**

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

- **The first line declares mybox as a reference to an object of type Box. After this line executes, mybox contains the value null, which indicates that it does not yet point to an actual object.**
- **The next line allocates an actual object and assigns a reference to it to mybox. After the second line executes, we can use mybox as if it were a Box object.**
- **The mybox simply holds the memory address of the actual Box object.**

- **Assigning Object Reference Variables :**

- Box b1 = new Box();
- Box b2 = b1;
- we might think that b2 is being assigned a reference to a copy of the object referred to by b1. That is, we might think that b1 and b2 refer to separate and distinct objects.
- Instead, after this fragment executes,b1and b2 will both refer to the same object. The assignment of b1 to b2 did not allocate any memory or copy any part of the original object.
- 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 b1is referring, since they are the same object.

- **Introducing Methods :**
- **Classes usually consist of two things: instance variables and methods. This is the general form of a method:**
- type name(parameter-list) {
- // body of method
- }
- **Here, type specifies the type of data returned by the method. This can be any valid type, including class types that we 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.**

- /* 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();
}
}
```

- **This program generates the following output :**
- Volume is 3000.0
- Volume is 162.0
- **Look closely at the following two lines of code:**
- mybox1.volume();
- mybox2.volume();
- **The first line here invokes the volume( )method on mybox1. That is, it calls volume( ) relative to the mybox1object, using the object's name followed by the dot operator. Thus, the call to mybox1.volume( ) displays the volume of the box defined by mybox1, and the call to mybox2.**
- **volume( ) displays the volume of the box defined by mybox2. Each time volume( ) is invoked, it displays the volume for the specified box.**

- **Returning a Value :**
- **While the implementation of volume( ) does move the computation of a box's volume inside the Boxclass where it belongs, it is not the best way to do it.**
- **For example, A better way to implement volume( ) is to have it compute the volume of the box and return the result to the caller.**
- // Now, volume() returns the volume of a box.
- class Box {
- double width;
- double height;
- double depth;
- // compute and return volume
- double volume() {
- return width * height * depth;
- }
- }

# Adding a Method that Takes Parameters :

- **While some methods don't need parameters. Parameters allow a method to be generalized. That is, a parameterized method can operate on a variety of data and be used in a number of slightly different situations. To illustrate this point, let's use a very simple example.**
- **Here is a method that returns the square of the number 10:**

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

- **While this method does, return the value of 10 squared, its use is very limited. However, if we modify the method so that it takes a parameter, then we can make square( ) much more useful.**

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

- **Now, square( ) will return the square of whatever value it is called with. That is, square( )is now a general-purpose method that can compute the square of any integer value, rather than just 10.**
- **Here is an example:**
- int x, y;
- x = square(5); // x equals 25
- x = square(9); // x equals 81
- y = 2;
- x = square(y); // x equals 4
- **In the first call to square( ), the value 5 will be passed into parameter i. In the second call, we will receive the value 9. The third invocation passes the value of y, which is 2 in this example.**
- **As these examples show, square( ) is able to return the square of whatever data it is passed.**

- **Constructors :**
- It can be tedious to initialize all of the variables in a class each time an instance is created. Even when we add convenience functions like SetDim(), it would be simpler and more concise to have all of the setup done at the time the object is first created.
- Because the requirement for initialization is so common, Java allows objects to initialize themselves when they are created. This *automatic initialization* is performed through the use of a constructor.
- Constructor initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method.

```
/* 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);
- }
- }
- **The output from this program is shown here:**
- Volume is 3000.0
- Volume is 162.0

- **The this Keyword :**
- **Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the ‘this” keyword. “this” can be used inside any method to refer to the *current* object. That is, this is always a reference to the object on which the method was invoked. We can use “this” anywhere a reference to an object of the current class type is permitted.**
- **To better understand what “this” refers to, consider the following version of Box( ):**

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

- **Garbage Collection:**
- In some languages, such as C++, dynamically allocated objects must be manually released by use of a delete operator. Java takes a different approach and handles deallocation automatically.
- The technique that accomplishes this is called garbage collection.
- It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be *reclaimed*. It will not occur simply because one or more objects exist that are no longer used.

# The finalize( ) Method

- Sometimes an object will need to *perform some action* when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or character font, then we might want to make sure these resources are freed before an object is destroyed.
- To handle such situations, Java provides a mechanism called finalization. By using finalization, we can define *specific actions* that will occur when an object is just about to be reclaimed by the garbage collector.

- **To add a finalizer to a class, we simply define the finalize( )method.**
- **Inside the finalize( ) method, we will specify those actions that must be performed before an object is destroyed. The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects.**
- **The finalize( )method has this general form:**
- protected void finalize( )
- {
- // finalization code here
- }

- Here, the key word protected is a specifier that prevents access to finalize( )by code defined outside its class. It is important to understand that finalize( )is only called just prior to garbage collection.

- **Overloading Methods :**
- In Java it is possible to define two or more methods within the *same class* that share the *same name*, as long as their parameter declarations are different.
- When this is the case, the methods are said to be overloaded, and the process is referred to as "*method overloading*".

- **Here is a simple example that illustrates 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
- **The test( ) is overloaded four times. The first version takes no parameters, the second takes one integer parameter, the third takes two integer parameters, and the fourth takes one double parameter.**

# Final Keyword

- **A variable can be declared as final then prevents its contents from being modified.**
- **For example:**
- final int FILE_NEW = 1;
- final int FILE_OPEN = 2;
- final int FILE_SAVE = 3;
- final int FILE_SAVEAS = 4;
- final int FILE_QUIT = 5;
- **Subsequent parts of our program can now use FILE_OPEN, etc., as if they were constants, without fear that a value has been changed. A final variableis essentially a constant.**

# Inheritance

- **To inherit a class, we simply incorporate the definition of one class into another by using the "*extends*" keyword.**
- **The following program creates a super class called A and a subclass called B. Notice how the keyword extends is used to create a subclass of A.**

```
// A simple example of inheritance.
// Create a superclass.
class A {
int i, j;
void showij() {
System.out.println("i and j: " + i + " " + j);
}
} // end A
```

- // Create a subclass by extending class A.
- class B extends A {
- int k;
- void showk() {
- System.out.println("k: " + k);
- }
- void sum() {
- System.out.println("i+j+k: " + (i+j+k));
- }
- }
- **The general form of a class declaration that inherits a super class is shown here:**

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

# Using super

- **Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the key word "*super*".**
- **The Super has two general forms. The first calls the superclass' constructor. The second is used to access a member of the superclass that has been hidden by a member of a subclass.**
- **The 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.**

- **The super( ) is used in the improved version of the BoxWeight( )class:**

- /* BoxWeight now uses super to initialize its Box attributes. */
- class BoxWeight extends Box {
- double weight; // weight of box
- /* initialize width, height, and depth using super() */
- BoxWeight(double w, double h, double d, double m) {
- super(w, h, d); // call superclass constructor
- weight = m;
- }
- }

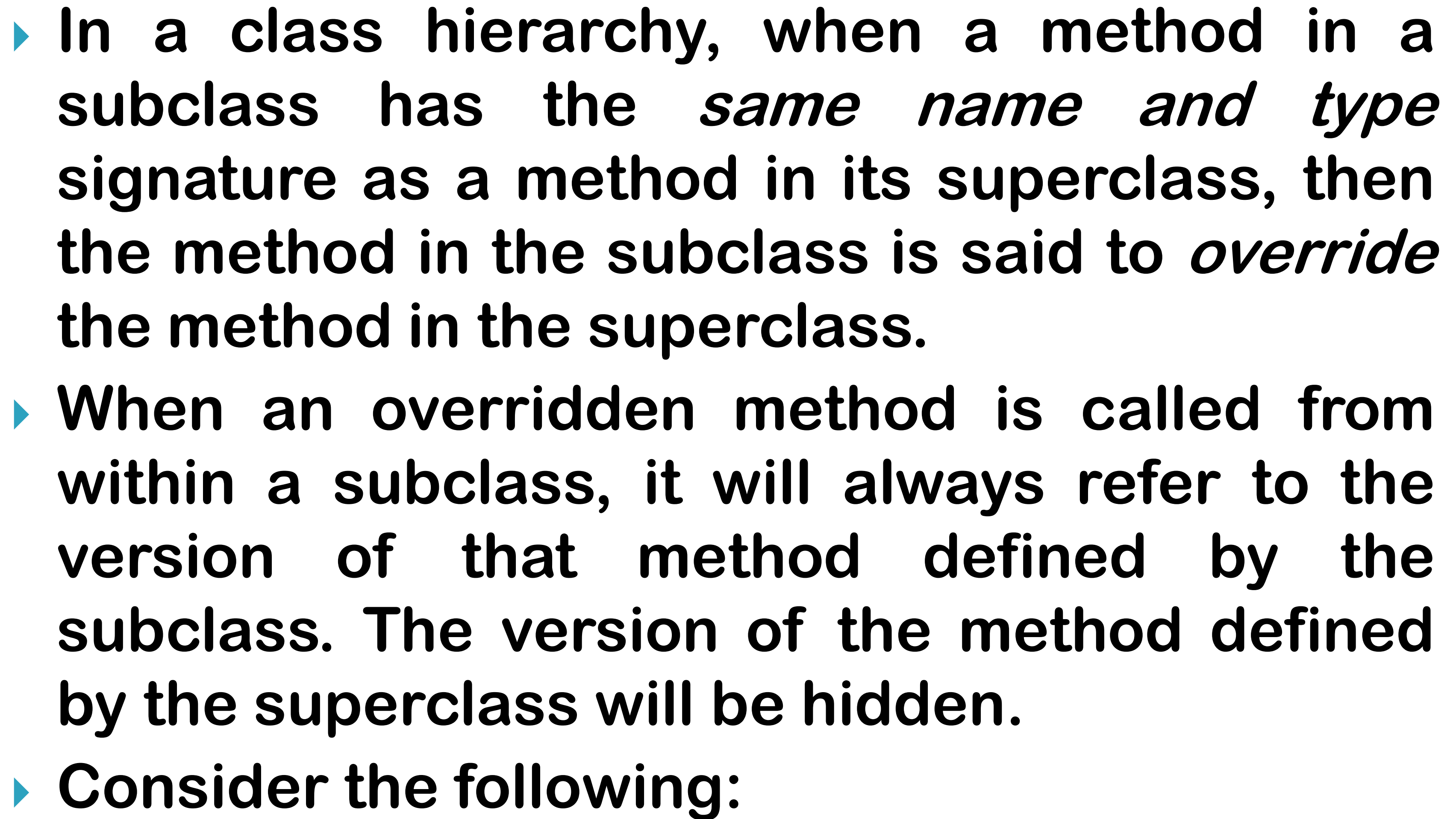
- **Here, BoxWeight( ) calls super( ) with the arguments w, h, and d. This causes the Box( ) constructor to be called, which initializes width, height, and depth using these values.**
- **Super( ) was called with three arguments. Since constructors can be overloaded, super() can be called using any form defined by the superclass.**
- **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. This usage has the following general form:**
- super.member
- **Here, member can be either a method or an instance variable.**

- **The 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 overcome name hiding.
- class A {
- int i;
- }
- // Create a subclass by extending class A.
- class B extends A {
- int i; // this i hides the i in A
- B(int a, int b) {
- super.i = a; // i in A
- i = b; // i in B
- }

- void show() {
- System.out.println("i in superclass: " + super.i);
- System.out.println("i in subclass: " + i);
- }
- }
- class UseSuper {
- public static void main(String args[]) {
- B subOb = new B(1, 2);
- subOb.show();
- }
- }
- **This program displays the following:**
- i in superclass: 1
- i in subclass: 2

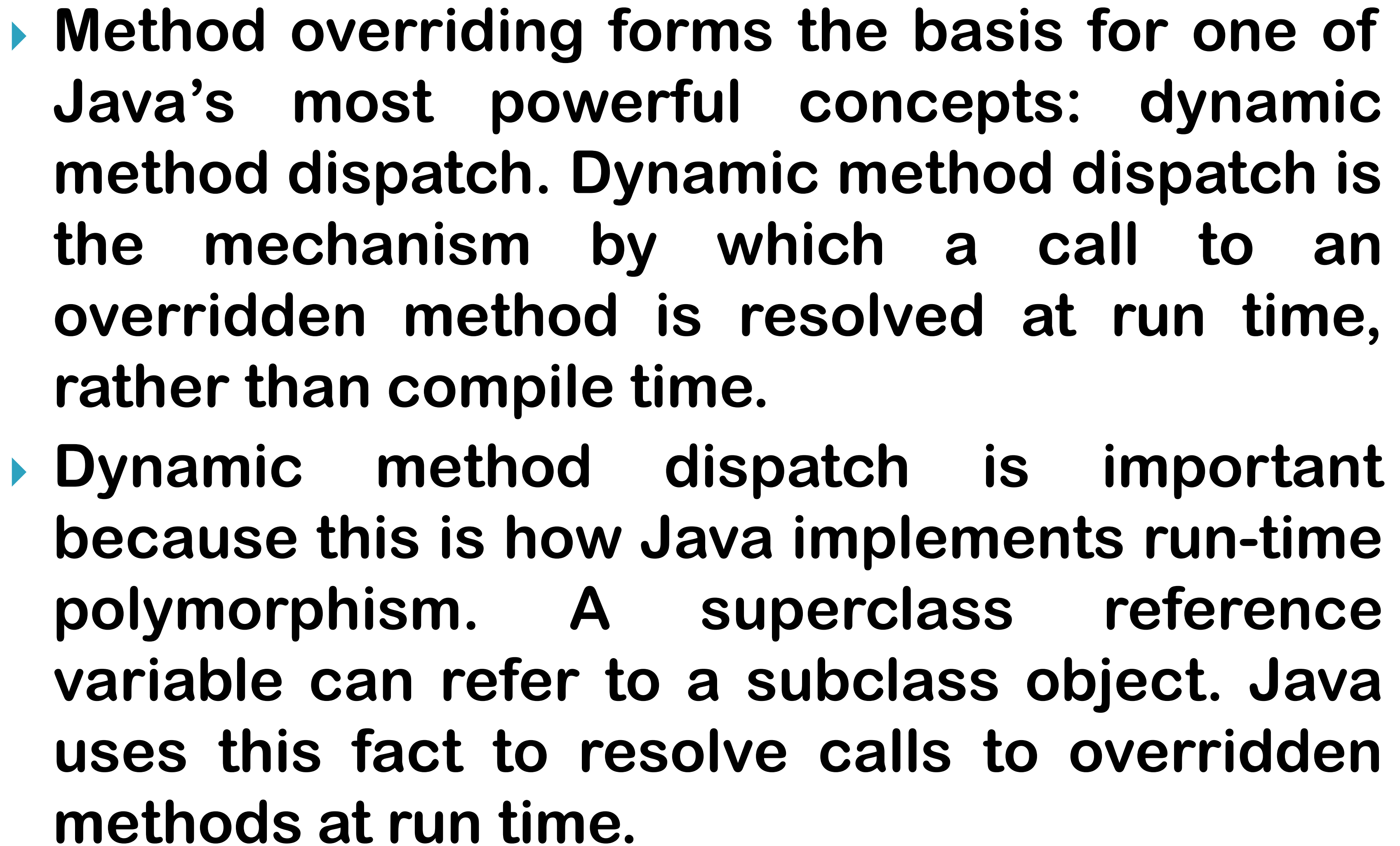
# Method Overriding



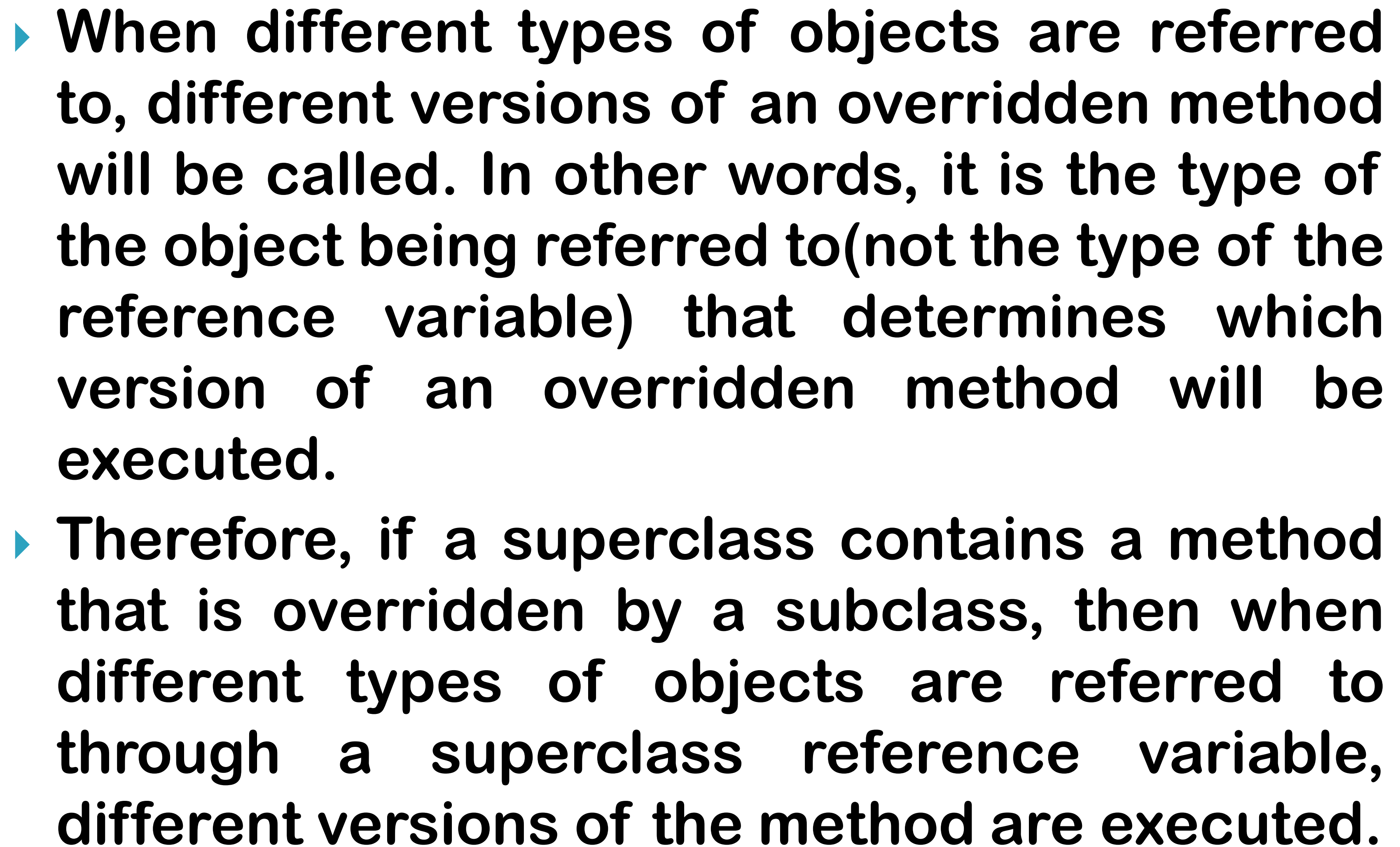
- In a class hierarchy, when a method in a subclass has the *same name and type* signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass.
- When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.
- Consider the following:

```
// 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
- }
- }
- **The output produced by this program is shown here:**
- k: 3



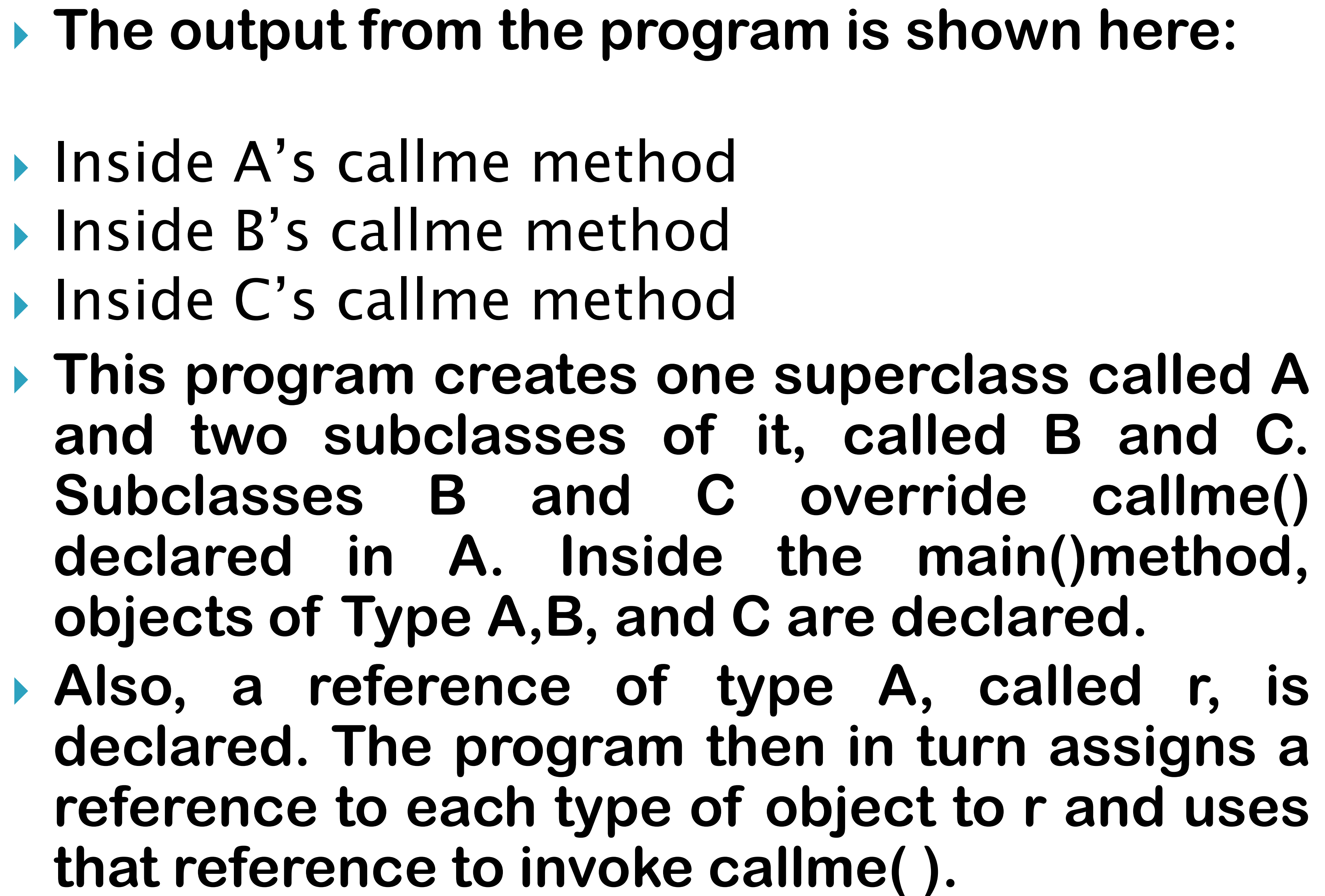
- ***Dynamic Method Dispatch :***
- **Method overriding forms the basis for one of Java's most powerful concepts: dynamic method dispatch. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.**
- **Dynamic method dispatch is important because this is how Java implements run-time polymorphism. A superclass reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time.**



- When different types of objects are referred to, different versions of an overridden method will be called. In other words, it is the type of the object being referred to(not the type of the reference variable) that determines which version of an overridden method will be executed.
- Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

- **Here is an example that illustrates dynamic method dispatch:**
- // 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
- }
- }



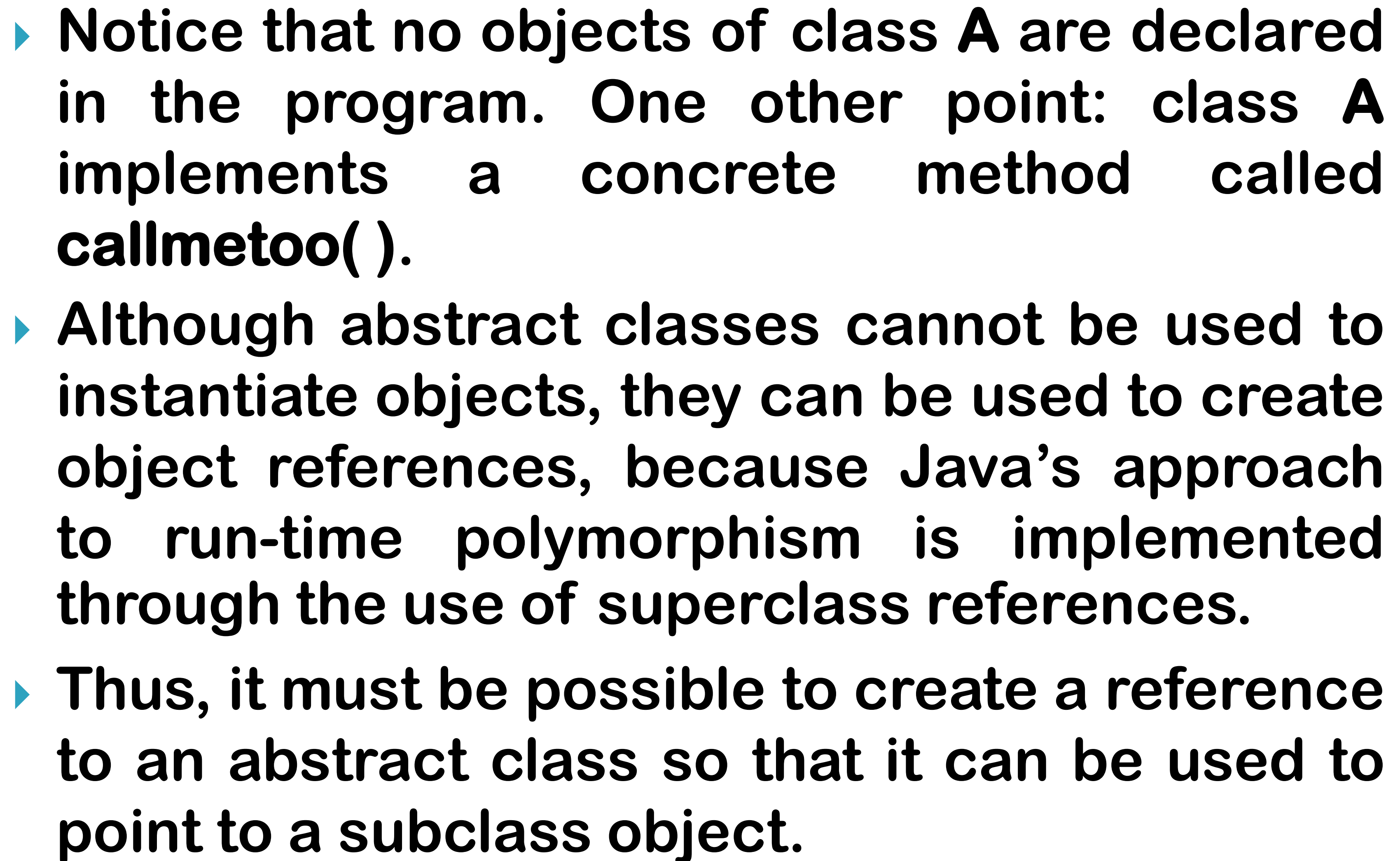
- **The output from the program is shown here:**
- Inside A’s callme method
- Inside B’s callme method
- Inside C’s callme method
- **This program creates one superclass called A and two subclasses of it, called B and C. Subclasses B and C override callme() declared in A. Inside the main()method, objects of Type A,B, and C are declared.**
- **Also, a reference of type A, called r, is declared. The program then in turn assigns a reference to each type of object to r and uses that reference to invoke callme( ).**

# Abstract Classes

- Sometimes, we want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.
- Such a class determines the nature of the methods that the subclasses must implement.
- We want some way to ensure that a subclass does, indeed, override all necessary methods. Java's solution to this problem is the *abstract method.*
- We can require that certain methods be overridden by subclasses by specifying the **abstract** type modifier.

- **To declare an abstract method, use this general form:**
- **abstract** *type name(parameter-list)*;
- **As we can see, no method body is present. Any class that contains one or more abstract methods must also be declared abstract. To declare a class abstract, we simply use the abstract keyword in front of the class keyword at the beginning of the class declaration.**
- **Here is a simple example of a class with an abstract method, followed by a class which implements that method:**

```
// A Simple demonstration of abstract.
abstract class A {
abstract void callme();
// concrete methods are still allowed in abstract classes
void callmetoo() {
System.out.println("This is a concrete method.");
}
}
class B extends A {
void callme() {
System.out.println("B's implementation of callme.");
}
}
class AbstractDemo {
public static void main(String args[]) {
B b = new B();
b.callme();
b.callmetoo();
}
}
```



- Notice that no objects of class **A** are declared in the program. One other point: class **A** implements a concrete method called **callmetoo( )**.
- Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java's approach to run-time polymorphism is implemented through the use of superclass references.
- Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object.

# Packages

- *Packages* are containers for classes that are used to keep the class name space compartmentalized.
- For example, a package allows us to create a class named **List**, which we can store in our own package without concern that it will collide with some other class named **List** stored elsewhere.
- To create a package is quite easy: simply include a **package** command as the first statement in a Java source file. The **package** statement defines a name space in which classes are stored.

- **This is the general form of the package statement:**
- package *pkg*;
- **Here, *pkg* is the name of the package. For example, the following statement creates a package called MyPackage.**
- package MyPackage;
- **Java uses file system directories to store packages. For example, the .class files for any classes we declare to be part of MyPackage must be stored in a directory called MyPackage.**

- **More than one file can include the same package statement. The package statement simply specifies to which package the classes defined in a file belong.**
- **The general form of a multileveled package statement is shown here:**
- package *pkg1*[.*pkg2*[.*pkg3*]];
- **A package hierarchy must be reflected in the file system of our Java development system. For example, a package declared as**
- package java.awt.image;
- **Needs to be stored in java\awt\image in a Windows environment.**

```
// A simple package
package MyPack;
class Balance {
String name;
double bal;
Balance(String n, double b) {
name = n;
bal = b;
}
void show() {
if(bal<0)
System.out.print("--> ");
System.out.println(name + ": $" + bal);
}
}
class AccountBalance {
public static void main(String args[]) {
Balance current[] = new Balance[3];
current[0] = new Balance("K. J. Fielding", 123.23);
current[1] = new Balance("Will Tell", 157.02);
current[2] = new Balance("Tom Jackson", -12.33);
for(int i=0; i<3; i++) current[i].show();
}
}
```

- Call this file **AccountBalance.java** and put it in a directory called **MyPack**.
- Next, compile the file. Make sure that the resulting **.class** file is also in the **MyPack** directory. Then, try executing the **AccountBalance** class, using the following command line:
- java MyPack.AccountBalance
- Remember, we will need to be in the directory above **MyPack** when we execute this command.

# Importing Packages

- Since classes within packages must be fully qualified with their package name or names, it could become *tedious* to type in the long dot-separated package path name for every class we want to use.
- For this reason, Java includes the **import** statement to bring certain classes, or entire packages, into visibility. Once imported, a class can be referred to directly, using only its name.

- **In a Java source file, import statements occur immediately following the package statement (if it exists) and before any class definitions.**
- **This is the general form of the import statement:**
- import *pkg1*[.*pkg2*].(*classname*|*);
- **Here, *pkg1* is the name of a top-level package, and *pkg2* is the name of a subordinate package inside the outer package separated by a dot (.).**
- **We specify either an explicit *classname* or a star (*), which indicates that the Java compiler should import the entire package.**

- import java.util.Date;
- import java.io.*;
- **All of the standard Java classes included with Java are stored in a package called java. The basic language functions are stored in a package inside of the java package called java.lang.**
- **This is equivalent to the following line being at the top of all of our programs:**
- import java.lang.*;
- **When a package is imported, only those items within the package declared as public will be available to non-subclasses in the importing code.**

# Defining Interface

- Through the use of the **interface** keyword, Java allows us to interface one or more classes can implement. Using **interface**, we can specify a set of methods that can be implemented by one or more classes.
- Using the keyword **interface**, we can fully abstract a class interface from its implementation.
- Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body.

- **To implement an interface, a class must create the complete set of methods defined by the interface.**
- **By providing the interface keyword, Java allows us to fully utilize the "one interface, multiple methods" aspect of polymorphism.**
- **An interface is defined much like a class. This is the general form of an interface:**
- *access* interface *name* {
- *return-type method-name1*(*parameter-list*);
- *return-type method-name2*(*parameter-list*);
- *type final-varname1 = value;*
- *type final-varname2 = value;*
- *// ...*
- *return-type method-nameN*(*parameter-list*);
- *type final-varnameN = value;*
- }

- **When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared.**
- ***name*** **is the name of the interface, and can be any valid identifier.**
- **Notice that the methods that are declared have no bodies. They end with a semicolon after the parameter list. They are, essentially, abstract methods.**
- **Here is an example of an interface definition. It declares a simple interface that contains one method called callback( ) that takes a single integer parameter.**
- interface Callback {
- void callback(int param);
- }

# Implementing Interfaces

- Once an **interface** has been defined, one or more classes can implement that interface. To implement an interface, include the **implements** clause in a class definition, and then create the methods defined by the interface.
- The general form of a class that includes the **implements** clause looks like this:
- class *classname* [extends *superclass*] [implements *interface* [,*interface...*]] {
- // class-body
- }

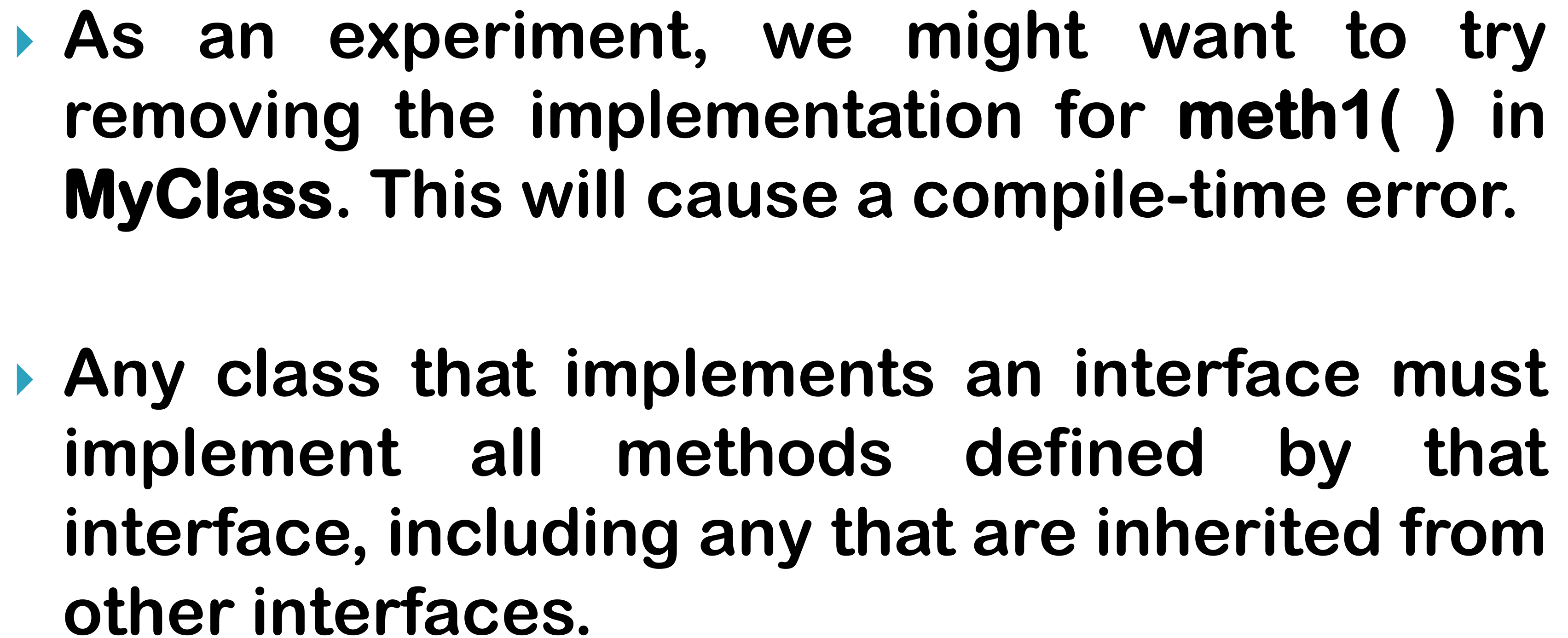
- **If a class implements more than one interface, the interfaces are separated with a comma. If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface.**
- **Here is a small example class that implements the Callback interface.**
- class Client implements Callback {
- // Implement Callback's interface
- public void callback(int p) {
- System.out.println("callback called with " + p);
- }
- }

# Extending Interfaces

- **One interface can inherit another by use of the keyword extends. The syntax is the same as for inheriting classes.**
- **When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.**
- **Following is an example:**
- // One interface can extend another.
- interface A {
- void meth1();
- void meth2();
- }

- /* B now includes meth1() and meth2() -- it adds meth3() */
- interface B extends A {
- void meth3();
- }
- // This class must implement all of A and B
- class MyClass implements B {
- public void meth1() {
- System.out.println("Implement meth1().");
- }
- public void meth2() {
- System.out.println("Implement meth2().");
- }

- public void meth3() {
- System.out.println("Implement meth3().");
- }
- }
- class IFExtend {
- public static void main(String arg[]) {
- MyClass ob = new MyClass();
- ob.meth1();
- ob.meth2();
- ob.meth3();
- }
- }



- As an experiment, we might want to try removing the implementation for **meth1( )** in **MyClass**. This will cause a compile-time error.

- Any class that implements an interface must implement all methods defined by that interface, including any that are inherited from other interfaces.