## JAVA CLASSES AND OBJECTS

* 1. **Introduction to OOPS:**

Object-oriented programming (OOP) is a programming paradigm that represents concepts as "objects" that have data fields (attributes that describe the object) and associated procedures known as methods. Objects, which are usually instances of classes, are used to interact with one another to design applications and computer programs.

An object oriented program may be viewed as a collection of interacting objects, as opposed to the conventional model, in which a program is seen as a list of tasks (subroutines) to perform. In OOP, each object is capable of receiving messages, processing data, and sending messages to other objects. Each object can be viewed as an independent "machine" with a distinct role or responsibility. Actions (or "methods") on these objects are closely associated with the object.

In contrast, the object-oriented approach encourages the programmer to place data where it is not directly accessible by the rest of the program. Instead, the data is accessed by calling specially written functions, commonly called methods, which are bundled in with the data. These act as the intermediaries for retrieving or modifying the data they control. The programming construct that combines data with a set of methods for accessing and managing those data is called an object. An object-oriented program usually contains different types of objects, each corresponding to a particular kind of complex data to manage, or perhaps to a real-world object or concept such as a bank account, a hockey player, or a bulldozer. A program might contain multiple copies of each type of object, one for each of the real-world objects the program deals with. For instance, there could be one bank account object for each real-world account at a particular bank. Each copy of the bank account object would be alike in the methods it offers for manipulating or reading its data, but the data inside each object would differ reflecting the different history of each account.

Java supports the following fundamental concepts:

1. Polymorphism
2. Inheritance
3. Encapsulation
4. Abstraction
5. Classes
6. Objects
7. Instance
8. Method
9. Message Parsing
   1. **Classes:**

In the real world, you'll often find many individual objects all of the same kind. There may be thousands of other bicycles in existence, all of the same make and model. Each bicycle was built from the same set of blueprints and therefore contains the same components. In object-oriented terms, we say that your bicycle is an instance of the class of objects known as bicycles. A class is the blueprint from which individual objects are created.

The following Bicycle class is one possible implementation of a bicycle:

class Bicycle {

private int cadence = 0;

private int speed = 0;

private int gear = 1;

void changeCadence(int newValue) {

cadence = newValue;

}

void changeGear(int newValue) {

gear = newValue;

}

void speedUp(int increment) {

speed = speed + increment;

}

void applyBrakes(int decrement) {

speed = speed - decrement;

}

void printStates() {

System.out.println("cadence:" +

cadence + " speed:" +

speed + " gear:" + gear);

}

}

The syntax of the Java programming language will look new to you, but the design of this class is based on the previous discussion of bicycle objects. The fields cadence, speed, and gear represent the object's state, and the methods (changeCadence, changeGear, speedUp etc.) define its interaction with the outside world.

You may have noticed that the Bicycle class does not contain a main method. That's because it's not a complete application; it's just the blueprint for bicycles that might be used in an application. The responsibility of creating and using new Bicycle objects belongs to some other class in your application.

Here's a BicycleDemo class that creates two separate Bicycle objects and invokes their methods:

class BicycleDemo {

public static void main(String[] args) {

// Create two different

// Bicycle objects

Bicycle bike1 = new Bicycle();

Bicycle bike2 = new Bicycle();

// Invoke methods on

// those objects

bike1.changeCadence(50);

bike1.speedUp(10);

bike1.changeGear(2);

bike1.printStates();

bike2.changeCadence(50);

bike2.speedUp(10);

bike2.changeGear(2);

bike2.changeCadence(40);

bike2.speedUp(10);

bike2.changeGear(3);

bike2.printStates();

}

}

The output of this test prints the ending pedal cadence, speed, and gear for the two bicycles:

cadence:50 speed:10 gear:2

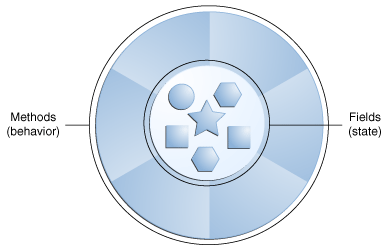
cadence:40 speed:20 gear:3

* 1. **Objects:**

Objects are key to understanding object-oriented technology. Look around right now and you'll find many examples of real-world objects: your dog, your desk, your television set, your bicycle.

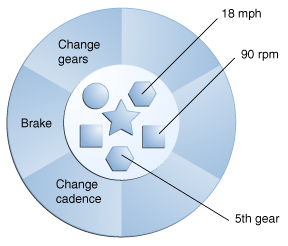
Real-world objects share two characteristics: They all have state and behavior. Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, changing pedal cadence, applying brakes). Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming.

Take a minute right now to observe the real-world objects that are in your immediate area. For each object that you see, ask yourself two questions: "What possible states can this object be in?" and "What possible behavior can this object perform?". Make sure to write down your observations. As you do, you'll notice that real-world objects vary in complexity; your desktop lamp may have only two possible states (on and off) and two possible behaviors (turn on, turn off), but your desktop radio might have additional states (on, off, current volume, current station) and behavior (turn on, turn off, increase volume, decrease volume, seek, scan, and tune). You may also notice that some objects, in turn, will also contain other objects. These real-world observations all translate into the world of object-oriented programming.



Software objects are conceptually similar to real-world objects: they too consist of state and related behavior. An object stores its state in fields (variables in some programming languages) and exposes its behavior through methods (functions in some programming languages). Methods operate on an object's internal state and serve as the primary mechanism for object-to-object communication. Hiding internal state and requiring all interaction to be performed through an object's methods is known as data encapsulation — a fundamental principle of object-oriented programming.

Consider a bicycle, for example:



By attributing state (current speed, current pedal cadence, and current gear) and providing methods for changing that state, the object remains in control of how the outside world is allowed to use it. For example, if the bicycle only has 6 gears, a method to change gears could reject any value that is less than 1 or greater than 6.

* 1. **Defining a Class:**

You've seen classes defined in the following way:

class *MyClass* {

// field, constructor, and

// method declarations

}

This is a class declaration. The class body (the area between the braces) contains all the code that provides for the life cycle of the objects created from the class: constructors for initializing new objects, declarations for the fields that provide the state of the class and its objects, and methods to implement the behavior of the class and its objects.

The preceding class declaration is a minimal one. It contains only those components of a class declaration that are required. You can provide more information about the class, such as the name of its superclass, whether it implements any interfaces, and so on, at the start of the class declaration. For example,

class *MyClass extends MySuperClass implements YourInterface* {

// field, constructor, and

// method declarations

}

means that ***MyClass*** is a subclass of ***MySuperClass*** and that it implements the ***YourInterface*** interface.

You can also add modifiers like public or private at the very beginning—so you can see that the opening line of a class declaration can become quite complicated. The modifiers public and private, which determine what other classes can access ***MyClass.*** In general, class declarations can include these components, in order:

1. Modifiers such as public, private, and a number of others that you will encounter later.
2. The class name, with the initial letter capitalized by convention.
3. The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
4. A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
5. The class body, surrounded by braces, {}.
   1. **Constructors:**

A class contains constructors that are invoked to create objects from the class blueprint. Constructor declarations look like method declarations—except that they use the name of the class and have no return type. For example, Bicycle may include one constructor:

public Bicycle(int startCadence, int startSpeed, int startGear) {

gear = startGear;

cadence = startCadence;

speed = startSpeed;

}

To create a new Bicycle object called myBike, a constructor is called by the new operator:

Bicycle myBike = new Bicycle(30, 0, 8);

new Bicycle(30, 0, 8) creates space in memory for the object and initializes its fields.

Although Bicycle only has one constructor, it could have others, including a no-argument constructor:

public Bicycle() {

gear = 1;

cadence = 10;

speed = 0;

}

Bicycle yourBike = new Bicycle(); invokes the no-argument constructor to create a new Bicycle object called yourBike.

Both constructors could have been declared in Bicycle because they have different argument lists. As with methods, the Java platform differentiates constructors on the basis of the number of arguments in the list and their types. You cannot write two constructors that have the same number and type of arguments for the same class, because the platform would not be able to tell them apart. Doing so causes a compile-time error.

You don't have to provide any constructors for your class, but you must be careful when doing this. The compiler automatically provides a no-argument, default constructor for any class without constructors. This default constructor will call the no-argument constructor of the superclass. In this situation, the compiler will complain if the superclass doesn't have a no-argument constructor so you must verify that it does. If your class has no explicit superclass, then it has an implicit superclass of Object, which *does* have a no-argument constructor.

* 1. **Methods:**

A Java method is a collection of statements that are grouped together to perform an operation. When you call the System.out.println method, for example, the system actually executes several statements in order to display a message on the console. In general, a method has the following syntax:

modifier returnValueType methodName(list of parameters) {

// Method body;

}

Here is an example of a typical method declaration:

/\*\* Return the max between two numbers \*/

public int max(int num1, int num2) {

int result;

if (num1 > num2)

result = num1;

else

result = num2;

return result;

}

In creating a method, you give a definition of what the method is to do. To use a method, you have to call or invoke it. There are two ways to call a method; the choice is based on whether the method returns a value or not. When a program calls a method, program control is transferred to the called method. A called method returns control to the caller when its return statement is executed or when its method-ending closing brace is reached. If the method returns a value, a call to the method is usually treated as a value. For example:

int larger = max(30, 40);

If the method returns void, a call to the method must be a statement. For example, the method println returns void. The following call is a statement:

System.out.println("Welcome to Java!");

**Example:**

public class TestMax {

/\*\* Main method \*/

public static void main(String[] args) {

TestMax tm = new TestMax();

int i = 5;

int j = 2;

int k = tm.max(i, j);

System.out.println("The maximum between " + i +

" and " + j + " is " + k);

}

/\*\* Return the max between two numbers \*/

public int max(int num1, int num2) {

int result;

if (num1 > num2)

result = num1;

else

result = num2;

return result;

}

}

* 1. **Accessing Class Members:**

We can access a class member in two ways, but it depends upon the keyword that you have used before variable declaration.

1. **Accessing a Variables:**

We can access any variables of a class using period operator (.) along with the object name. If any variable of class is defined or declared using the keyword static, then it can be accessed directly with the class reference.

1. **Accessing Methods:**

Accessing a method is same as accessing variables in Java. If any methods of a class is defined or declared using the keyword static, then it can be accessed directly with the class reference.

* 1. **Garbage Collection:**

In the Java programming language, dynamic allocation of objects is achieved using the new operator. An object once created uses some memory and the memory remains allocated till there are references for the use of the object. When there are no references for an object, it is assumed to be no longer needed and the memory occupied by the object can be reclaimed. There is no explicit need to destroy an object as java handles the de-allocation automatically. The technique that accomplishes this is known as Garbage Collection. Programs that do not de-allocate memory can eventually crash when there is no memory left in the system to allocate. These programs are said to have memory leaks.

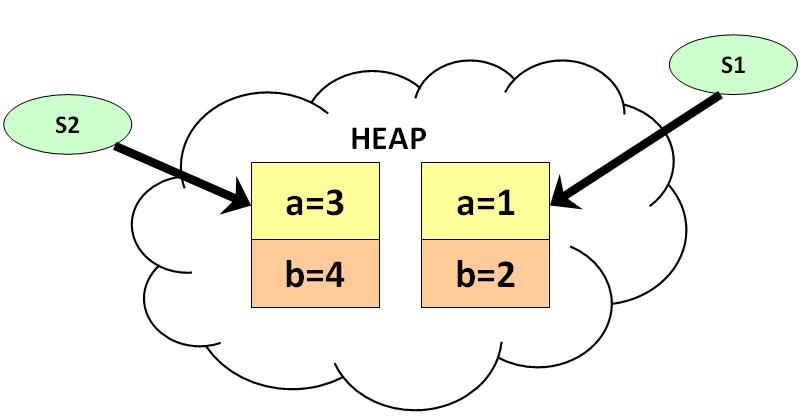
In Java, Garbage collection happens automatically during the lifetime of a java program, eliminating the need to de-allocate memory and avoiding memory leaks.

In C language, it is the programmer’s responsibility to de-allocate memory allocated dynamically using free() function.

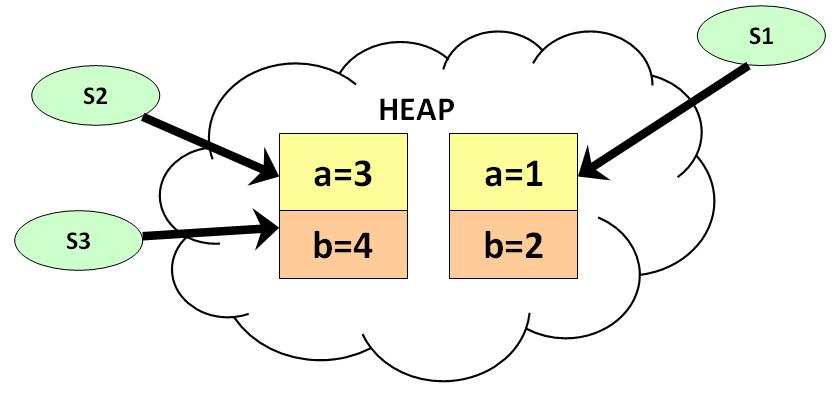
**Note: All objects are created in Heap Section of memory.**

1. class Student{
2. int a;
3. int b;
4. public void setData(int c ,int d){
5. a=c;
6. b=d;
7. }
8. public void showData(){
9. System.out.println("Value of a = "+a);
10. System.out.println("Value of a = "+b);
11. }
12. public static void main(String args[]){
13. Student s1 = new Student();
14. Student s2 = new Student();
15. s1.setData(1,2);
16. s2.setData(3,4);
17. s1.showData();
18. s2.showData();
19. //Student s3;
20. //s3=s2;
21. //s3.showData();
22. //s2=null;
23. //s3.showData();
24. //s3=null;
25. //s3.showData();
26. }
27. }

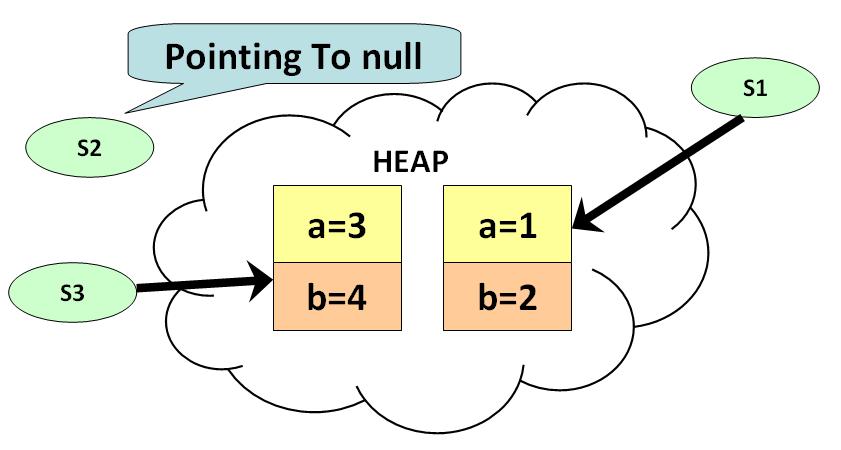
**Save, Compile and Run the code. As shown in the diagram, two objects and two reference variables are created.**



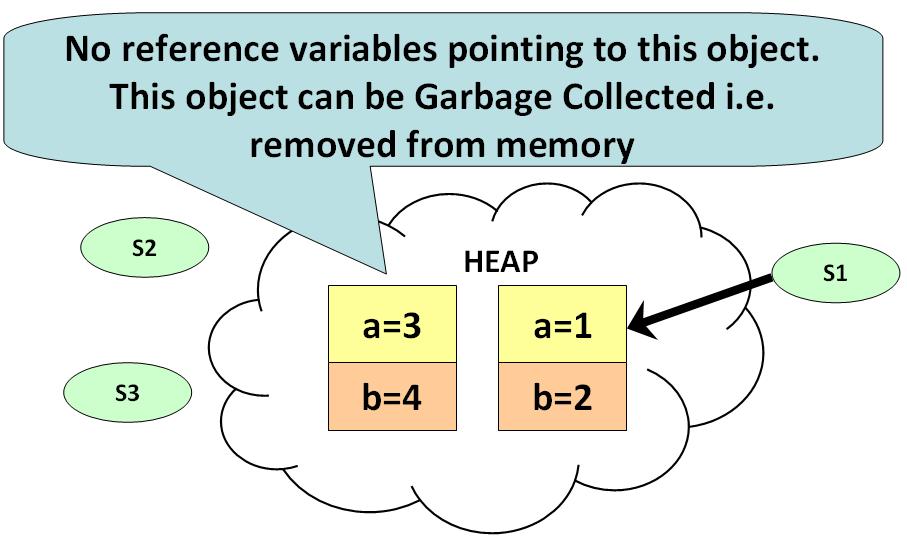
**Uncomment line #19, #20, #21. Save, Compile and Run the code. Two reference variables are pointing to the same object as shown below:**



**Uncomment the line #22, #23. Save, Compile and Run the code. As show in diagram below, s2 becomes null, but s3 is still pointing to the object and is not eligible for garbage collection.**



**Uncomment line # 24 & 25. Save, Compile & Run the Code. At this point there are no references pointing to the object and becomes eligible for garbage collection. It will be removed from memory and there is no way of retrieving it back.**



* 1. **The finalize() Method:**

The java.lang.Object.finalize() is called by the garbage collector on an object when garbage collection determines that there are no more references to the object. A subclass overrides the finalize method to dispose of system resources or to perform other cleanup.

**Syntax:**

protected void finalize()

* 1. **Overloading Methods:**

The Java programming language supports overloading methods, and Java can distinguish between methods with different method signatures. This means that methods within a class can have the same name if they have different parameter lists.

Suppose that you have a class that can use calligraphy to draw various types of data (strings, integers, and so on) and that contains a method for drawing each data type. It is cumbersome to use a new name for each method—for example, drawString, drawInteger, drawFloat, and so on. In the Java programming language, you can use the same name for all the drawing methods but pass a different argument list to each method. Thus, the data drawing class might declare four methods named draw, each of which has a different parameter list.

public class DataArtist {

...

public void draw(String s) {

...

}

public void draw(int i) {

...

}

public void draw(double f) {

...

}

public void draw(int i, double f) {

...

}

}

Overloaded methods are differentiated by the number and the type of the arguments passed into the method. In the code sample, draw(String s) and draw(int i) are distinct and unique methods because they require different argument types. You cannot declare more than one method with the same name and the same number and type of arguments, because the compiler cannot tell them apart. The compiler does not consider return type when differentiating methods, so you cannot declare two methods with the same signature even if they have a different return type.

**Example:**

public class DataArtist {

public void draw(String s) {

System.out.println("Drawn: " + s);

}

public void draw(int i) {

System.out.println("Drawn: " + i);

}

public void draw(double f) {

System.out.println("Drawn: " + f);

}

public void draw(int i, double f) {

System.out.println("Drawn: " + i + " and " + f);

}

public static void main(String[] args) {

DataArtist da = new DataArtist();

da.draw("NCCS");

da.draw(5);

da.draw(10.1);

da.draw(10, 11.11);

}

}

* 1. **Overloading Constructors:**

Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists. The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.

**Example:**

class Demo{

      int  value1;

      int  value2;

     Demo(){

       value1 = 10;

       value2 = 20;

       System.out.println("Inside 1st Constructor");

     }

     Demo(int a){

      value1 = a;

      System.out.println("Inside 2nd Constructor");

     }

     Demo(int a, int b){

     value1 = a;

     value2 = b;

     System.out.println("Inside 3rd Constructor");

    }

    public void display(){

       System.out.println("Value1 === "+value1);

       System.out.println("Value2 === "+value2);

   }

public static  void  main(String args[]){

     Demo d1 = new Demo();

    Demo d2 = new Demo(30);

     Demo d3 = new Demo(30,40);

     d1.display();

     d2.display();

     d3.display();

  }

}

* 1. **Passing Objects as Parameter:**

It is both correct and common to pass objects to methods. For example, consider the following short program:

// Objects may be passed to methods.

class Test {

int a, b;

Test(int i, int j) {

a = i;

b = j;

}

// return true if o is equal to the invoking object

boolean equals(Test o) {

if(o.a == a && o.b == b)

return true;

else

return false;

}

}

class PassObject {

public static void main(String args[]) {

Test ob1 = new Test(100, 22);

Test ob2 = new Test(100, 22);

Test ob3 = new Test(-1, -1);

System.out.println("ob1 == ob2: " + ob1.equals(ob2));

System.out.println("ob1 == ob3: " + ob1.equals(ob3));

}

}

* 1. **Returning Objects:**

A method returns to the code that invoked it when it:

* completes all the statements in the method,
* reaches a return statement, or
* throws an exception,

which ever occurs first.

You declare a method's return type in its method declaration. Within the body of the method, you use the return statement to return the value.

Any method declared void doesn't return a value. It does not need to contain a return statement, but it may do so. In such a case, a return statement can be used to branch out of a control flow block and exit the method and is simply used like this:

return;

If you try to return a value from a method that is declared void, you will get a compiler error.

Any method that is not declared void must contain a return statement with a corresponding return value, like this:

return returnValue;

The data type of the return value must match the method's declared return type; you can't return an integer value from a method declared to return a boolean.

// Returning an object.

class Test {

int a;

Test(int i) {

a = i;

}

Test incrByTen() {

Test temp = new Test(a+10);

return temp;

}

}

class ReturingObject {

public static void main(String args[]) {

Test ob1 = new Test(2);

Test ob2;

ob2 = ob1.incrByTen();

System.out.println("ob1.a: " + ob1.a);

System.out.println("ob2.a: " + ob2.a);

ob2 = ob2.incrByTen();

System.out.println("ob2.a after second increase: "

+ ob2.a);

}

}

* 1. **Static Members (Static Methods and Blocks):**

There will be times when you will want to define a class member that will be used independently of any object of that class. Normally, a class member must be accessed only in conjunction with an object of its class. However, it is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member, precede its declaration with the keyword **static**. When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object. You can declare both methods and variables to be **static**. The most common example of a **static** member is main(). main() is declared as **static** because it must be called before any objects exist. Instance variables declared as **static** are, essentially, global variables. When objects of its class are declared, no copy of a **static** variable is made. Instead, all instances of the class share the same **static** variable. Methods declared as static have several restrictions:

• They can only call other **static** methods.

• They must only access **static** data.

• They cannot refer to **this** or **super** in any way.

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. The following example shows a class that has a static method, some static variables, and a static initialization 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[]) {

UseStatic.meth(42);

}

}

* 1. **Using the final Keyword:**

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

final int FILE\_NEW = 1;

final int FILE\_OPEN = 2;

final int FILE\_SAVE = 3;

final int FILE\_SAVEAS = 4;

final int FILE\_QUIT = 5;

Subsequent parts of your program can now use FILE\_OPEN, etc., as if they were constants, without fear that a value has been changed. It is a common coding convention to choose all uppercase identifiers for final variables. Variables declared as final do not occupy memory on a per-instance basis. Thus, a final variable is essentially a constant. The keyword final can also be applied to methods, but its meaning is substantially different than when it is applied to variables.

* 1. **Introduction to Nested and Inner Classes:**

The Java programming language allows you to define a class within another class. Such a class is called a nested class and is illustrated here:

class OuterClass {

...

class NestedClass {

...

}

}

**Terminology:** Nested classes are divided into two categories: *static* and *non-static*. Nested classes that are declared static are called *static nested classes*. Non-static nested classes are called *inner classes*.

class OuterClass {

...

static class StaticNestedClass {

...

}

class InnerClass {

...

}

}

A nested class is a member of its enclosing class. Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared private. Static nested classes do not have access to other members of the enclosing class. As a member of the OuterClass, a nested class can be declared private, public, protected, or package private.

// Demonstrate an inner class.

class Outer {

int outer\_x = 100;

void test() {

Inner inner = new Inner();

inner.display();

}

// this is an inner class

class Inner {

void display() {

System.out.println("display: outer\_x = " + outer\_x);

}

}

}

class InnerClassDemo {

public static void main(String args[]) {

Outer outer = new Outer();

outer.test();

}

}

* 1. **Introduction to String Class and String Manipulation:**

In Java, a string is a sequence of characters. But, unlike many other languages that implement strings as character arrays, Java implements strings as objects of type String. Implementing strings as built-in objects allows Java to provide a full complement of features that make string handling convenient. For example, Java has methods to compare two strings, search for a substring, concatenate two strings, and change the case of letters within a string. Also, String objects can be constructed a number of ways, making it easy to obtain a string when needed.

* **Creating Strings:**

The most direct way to create a string is to write:

String greeting ="Hello world!";

Whenever it encounters a string literal in your code, the compiler creates a String object with its value, in this case,

"Hello world!”

As with any other object, you can create String objects by using the new keyword and a constructor. The String class has eleven constructors that allow you to provide the initial value of the string using different sources, such as an array of characters:

public class StringDemo{

public static void main(String args[]){

char[] helloArray ={'h','e','l','l','o','.'};

String helloString =new String(helloArray);

System.out.println(helloString);

}

}

* **String Length:**

Methods used to obtain information about an object are known as accessor methods. One accessor method that you can use with strings is the length() method, which returns the number of characters contained in the string object.

public class StringDemo{

public static void main(String args[]){

String palindrome ="Dot saw I was Tod";

int len = palindrome.length();

System.out.println("String Length is: "+ len);

}

}

This will produce the following result:

String Length is: 17

* **Concatenating Strings:**

The String class includes a method for concatenating two strings:

string1.concat(string2);

This returns a new string that is string1 with string2 added to it at the end. You can also use the concat() method with string literals, as in:

"My name is ".concat("Zara");

Strings are more commonly concatenated with the + operator, as in:

"Hello,"+" world"+"!"

which results in:

"Hello, world!"

Let us look at the following example:

public class StringDemo{

public static void main(String args[]){

String string1 ="saw I was ";

System.out.println("Dot "+ string1 +"Tod");

}

}

This would produce the following result:

Dot saw I was Tod

* **String Conversion and toString( ):**

When Java converts data into its string representation during concatenation, it does so by calling one of the overloaded versions of the string conversion method **valueOf()** defined by **String**. **valueOf()** is overloaded for all the simple types and for type Object. For the simple types, **valueOf()** returns a string that contains the human-readable equivalent of the value with which it is called. For objects, **valueOf()** calls the **toString()** method on the object.

**Example:**

public class Test{

public static void main(String args[]){

Integer x = 5;

System.out.println(x.toString());

System.out.println(Integer.toString(12));

}

}

**Output:**

5

12

* **Character Extraction:**

The String class provides a number of ways in which characters can be extracted from a String object. Although the characters that comprise a string within a String object cannot be indexed as if they were a character array, many of the String methods employ an index (or offset) into the string for their operation. Like arrays, the string indexes begin at zero.

* **charAt():**

To extract a single character from a String, you can refer directly to an individual character via the charAt( ) method. It has this general form:

char charAt(int where)

Here, where is the index of the character that you want to obtain. The value of where must be nonnegative and specify a location within the string. **charAt()** returns the character at the specified location. For example,

char ch;

ch = "abc".charAt(1);

assigns the value “b” to ch.

* **String Comparison:**

The String class includes several methods that compare strings or substrings within strings.

* **equals() and equalsIgnoreCase():**

To compare two strings for equality, use equals( ). It has this general form:

boolean equals(Object str)

Here, str is the String object being compared with the invoking String object. It returns true if the strings contain the same characters in the same order, and false otherwise. The comparison is case-sensitive. To perform a comparison that ignores case differences, call equalsIgnoreCase(). When it compares two strings, it considers A-Z to be the same as a-z. It has this general form:

boolean equalsIgnoreCase(String str)

Here, str is the String object being compared with the invoking String object. It, too, returns true if the strings contain the same characters in the same order, and false otherwise. Here is an example that demonstrates equals() and equalsIgnoreCase( ):

// Demonstrate equals() and equalsIgnoreCase().

class EqualsDemo {

public static void main(String args[]) {

String s1 = "Hello";

String s2 = "Hello";

String s3 = "Good-bye";

String s4 = "HELLO";

System.out.println(s1 + " equals " + s2 + " -> " +

s1.equals(s2));

System.out.println(s1 + " equals " + s3 + " -> " +

s1.equals(s3));

System.out.println(s1 + " equals " + s4 + " -> " +

s1.equals(s4));

System.out.println(s1 + " equalsIgnoreCase " + s4 + " -> " +

s1.equalsIgnoreCase(s4));

}

}

The output from the program is shown here:

Hello equals Hello -> true

Hello equals Good-bye -> false

Hello equals HELLO -> false

Hello equalsIgnoreCase HELLO -> true

* **equals() Versus ==**

It is important to understand that the equals() method and the == operator perform two different operations. As just explained, the equals() method compares the characters inside a String object. The == operator compares two object references to see whether they refer to the same instance. The following program shows how two different String objects can contain the same characters, but references to these objects will not compare as equal:

// equals() vs ==

class EqualsNotEqualTo {

public static void main(String args[]) {

String s1 = "Hello";

String s2 = new String(s1);

System.out.println(s1 + " equals " + s2 + " -> " +

s1.equals(s2));

System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));

}

}

The variable s1 refers to the String instance created by “Hello”. The object referred to by s2 is created with s1 as an initializer. Thus, the contents of the two String objects are identical, but they are distinct objects. This means that s1 and s2 do not refer to the same objects and are, therefore, not ==, as is shown here by the output of the preceding example:

Hello equals Hello -> true

Hello == Hello -> false

* **concat():**

You can concatenate two strings using concat( ), shown here:

String concat(String str)

This method creates a new object that contains the invoking string with the contents of str appended to the end. concat( ) performs the same function as +. For example,

String s1 = "one";

String s2 = s1.concat("two");

puts the string “onetwo” into s2. It generates the same result as the following sequence:

String s1 = "one";

String s2 = s1 + "two";

* **Changing the Case of Characters Within a String:**

The method toLowerCase() converts all the characters in a string from uppercase to lowercase. The toUpperCase() method converts all the characters in a string from lowercase to uppercase. Nonalphabetical characters, such as digits, are unaffected. Here are the general forms of these methods:

String toLowerCase( )

String toUpperCase( )

Both methods return a String object that contains the uppercase or lowercase equivalent of the invoking String. Here is an example that uses toLowerCase() and toUpperCase():

// Demonstrate toUpperCase() and toLowerCase().

class ChangeCase {

public static void main(String args[])

{

String s = "This is a test.";

System.out.println("Original: " + s);

String upper = s.toUpperCase();

String lower = s.toLowerCase();

System.out.println("Uppercase: " + upper);

System.out.println("Lowercase: " + lower);

}

}

The output produced by the program is shown here:

Original: This is a test.

Uppercase: THIS IS A TEST.

Lowercase: this is a test.

**UNIT 4: INHERITANCE**

Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it. In the terminology of Java, a class that is inherited is called a superclass. The class that does the inheriting is called a subclass. Therefore, a subclass is a specialized version of a superclass. It inherits all of the instance variables and methods defined by the superclass and add its own, unique elements.

* 1. **Inheritance Basics & Reusability:**

To inherit a class, you simply incorporate the definition of one class into another by using the ***extends*** keyword. To see how, let’s begin with a short example. The following program creates a superclass 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);

}

}

// 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));

}

}

public class SimpleInheritance {

public static void main(String args[]) {

A superOb = new A();

B subOb = new B();

// The superclass may be used by itself.

superOb.i = 10;

superOb.j = 20;

System.out.println("Contents of superOb: ");

superOb.showij();

System.out.println();

/\* The subclass has access to all public members of

its superclass. \*/

subOb.i = 7;

subOb.j = 8;

subOb.k = 9;

System.out.println("Contents of subOb: ");

subOb.showij();

subOb.showk();

System.out.println();

System.out.println("Sum of i, j and k in subOb:");

subOb.sum();

}

}

The output from this program is shown here:

Contents of superOb:

i and j: 10 20

Contents of subOb:

i and j: 7 8

k: 9

Sum of i, j and k in subOb:

i+j+k: 24

* **IS-A Relationship:**

IS-A is a way of saying : This object is a type of that object. Let us see how the extends keyword is used to achieve inheritance.

public class Animal{

}

public class Mammal extends Animal{

}

public class Reptile extends Animal{

}

public class Dog extends Mammal{

}

Now, based on the above example, In Object Oriented terms the following are true:

* Animal is the superclass of Mammal class.
* Animal is the superclass of Reptile class.
* Mammal and Reptile are subclasses of Animal class.
* Dog is the subclass of both Mammal and Animal classes.

Now, if we consider the IS-A relationship, we can say:

* Mammal IS-A Animal
* Reptile IS-A Animal
* Dog IS-A Mammal
* Hence : Dog IS-A Animal as well

With use of the extends keyword the subclasses will be able to inherit all the properties of the superclass except for the private properties of the superclass. We can assure that Mammal is actually an Animal with the use of the “instanceof” operator:

class Animal{}

class Mammal extends Animal{}

public class Dog extends Mammal{

public static void main(String args[]){

Animal a =new Animal();

Mammal m =new Mammal();

Dog d = new Dog();

System.out.println(m instanceof Animal);

System.out.println(d instanceof Mammal);

System.out.println(d instanceof Animal);

}

}

This would produce the following result:

true

true

true

* 1. **Inheritance Types & Multilevel Inheritance:**

There exist basically four types of inheritance.

1. Single Inheritance: In single inheritance, one class extends one class only.
2. Multilevel Inheritance: In multilevel inheritance, the ladder of single inheritance increases.
3. Multiple Inheritance: In multiple inheritance, one class directly extends more than one class.
4. Hierarchical Inheritance: In hierarchical inheritance one class is extended by more than one class.

Lets examine each types briefly:

1. **Single Inheritance:**

Single inheritance enables a derived class to inherit properties and behavior from a single parent class. It allows a derived class to inherit the properties and behavior of a base class, thus enabling code reusability as well as adding new features to the existing code. This makes the code much more elegant and less repetitive.

**Example:**

class Aves {

public void nature() {

System.out.println("Generally, Aves fly");

}

}

public class Bird extends Aves {

public void eat() {

System.out.println("Eats to live");

}

public static void main(String[] args){

Bird b = new Bird();

b.nature();

b.eat();

}

}

Output:

Generally, Aves fly

Eats to live

1. **Multilevel Inheritance:**

In multilevel, one-to-one ladder increases. Multiple classes are involved in inheritance, but one class extends only one. The lowermost subclass can make use of all its super classes' members. Multilevel inheritance is an indirect way of implementing multiple inheritance. Following program explains.

class Aves {

public void nature() {

System.out.println("Generally, Aves fly");

}

}

class Bird extends Aves {

public void eat() {

System.out.println("Eats to live");

}

}

public class Parrot extends Bird {

public void food() {

System.out.println("Parrot eats seeds and fruits");

}

public static void main(String args[]) {

Parrot p1 = new Parrot();

p1.food(); // calling its own

p1.eat(); // calling super class Bird method

p1.nature(); // calling super class Aves method

}

}

Output:

Parrot eats seeds and fruits

Eats to live

Generally, Aves fly

Now, Parrot has two super classes Bird and Aves; but extended one-to-one. Parrot can make use of the methods of Bird and Aves. Bird can make use of the methods of Aves, but Parrot as a super class cannot access subclass members.

1. **Multiple Inheritance:**

In multiple inheritance, one class extends multiple classes. **Java does not support multiple inheritance** but C++ supports. The above program can be modified to illustrate multiple inheritance. The following program does not work.

class Aves { }

class Bird { }

public class Parrot extends Aves, Bird { }

In the above code, Parrot extends both Aves and Bird. This is not supported by Java and the above code raises compilation error. We will discuss this topic in Interface.

1. **Hierarchical Inheritance:**

In hierarchical type of inheritance, one class is extended by many subclasses. It is one-to-many relationship.

Example:

class Aves {

public void fly() {

System.out.println("Generally, aves fly");

}

}

class Parrot extends Aves {

public void eat() {

System.out.println("Parrot eats fruits and seeds");

}

}

class Vulture extends Aves {

public void vision() {

System.out.println("Vulture can see from high altitudes");

}

}

public class FlyingCreatures {

public static void main(String args[]) {

// all the following code is composition for //FlyingCreatures

Parrot p1 = new Parrot();

p1.eat(); // calling its own member

p1.fly(); // calling super class member by inheritance

Vulture v1 = new Vulture();

v1.vision(); // calling its own member

v1.fly(); // calling super class member by inheritance

}

}

Output:

Parrot eats fruits and seeds

Generally, aves fly

Vulture can see from high altitudes

Generally, aves fly

In the above code, Aves class is extended by two classes – Parrot and Vulture. Both classes can make use of the methods of Aves. Even though the Parrot and Vulture are subclasses of the same class Aves, but still they cannot make use of each other members. Parrot and Vulture are known as "siblings". Siblings are disjoint and they cannot make use of other members as between them no inheritance is involved (like two sons of a father; one son's property cannot be shared by other but both can share the property of father).

* 1. **Constructor Inheritance and use of Super:**

Class can not only be determined by state and behavior of class but also by each of its superclasses. Due to that reason it is not sufficient to execute a constructor of that class but also need to initialize each constructor of superclasses.

A superclass constructor must execute before a subclass constructor. So that the state and behavior defined by the superclass may be correctly and completely initialized before a subclass constructor executes.

***Java compiler assumes that the first line of every constructor is an implicit call to the default superclass constructor unless you explicitly use super() or this().***

Note that Super keyword is used to explicitly invoke a superclass constructor. If you use this form, it must appear as the first statement of the constructor to ensure that the superclass constructor executes before the subclass constructor in java.

**Example 1: Program that shows the default execution of constructor when classes are inherited**

class Person{

Person(){

System.out.println("Constructor of person class");

}

}

class Employee extends Person{

Employee(){

System.out.println("Constructor of employee class");

}

}

class PermanentEmployee extends Employee{

PermanentEmployee(){

System.out.println("Constructor of permanent employee class");

}

}

class ConstructorInheritanceDemo{

public static void main(String[] args){

PermanentEmployee pObj = new PermanentEmployee();

}

}

Output

Constructor of person class

Constructor of employee class

Constructor of permanent employee class

**Example 2 : Program that illustrates the explicit use of super() in an inheritance hierarchy.**

class Person{

String FirstName;

String LastName;

Person(String fName, String lName){

FirstName = fName;

LastName = lName;

}

}

class Student extends Person{

int id;

String standard;

String instructor;

Student(String fName, String lName, int nId, String stnd, String instr){

super(fName,lName);

id = nId;

standard = stnd;

instructor = instr;

}

}

class SuperKeywordForConstructorDemo{

public static void main(String[] args){

Student sObj = new Student("Jacob","Smith",1,"1 - B","Roma");

System.out.println("Student :");

System.out.println("First Name : " + sObj.FirstName);

System.out.println("Last Name : " + sObj.LastName);

System.out.println("ID : " + sObj.id);

System.out.println("Standard : " + sObj.standard);

System.out.println("Instructor : " + sObj.instructor);

}

}

Output

Student :

First Name : Jacob

Last Name : Smith

ID : 1

Standard : 1 - B

Instructor : Roma

* 1. **Overriding Methods:**

An instance method in a subclass with the same signature (name, plus the number and the type of its parameters) and return type as an instance method in the superclass overrides the superclass's method.

The ability of a subclass to override a method allows a class to inherit from a superclass whose behavior is "close enough" and then to modify behavior as needed. The overriding method has the same name, number and type of parameters, and return type as the method it overrides. An overriding method can also return a subtype of the type returned by the overridden method. This is called a covariant return type.

class Animal{

public void move(){

System.out.println("Animals can move");

}

}

class Dog extends Animal{

public void move(){

System.out.println("Dogs can walk and run");

}

}

public class TestDog{

public static void main(String args[]){

Animal a =newAnimal();// Animal reference and object

Animal b =newDog();// Animal reference but Dog object

a.move();// runs the method in Animal class

b.move();//Runs the method in Dog class

}

}

This would produce the following result:

Animals can move

Dogs can walk and run

In the above example, you can see that the even though b is a type of Animal it runs the move method in the Dog class. The reason for this is: In compile time, the check is made on the reference type. However , in the runtime, JVM figures out the object type and would run the method that belongs to that particular object. Therefore, in the above example, the program will compile properly since Animal class has the method move. Then, at the runtime, it runs the method specific for that object. Consider the following example:

class Animal{

public void move(){

System.out.println("Animals can move");

}

}

class Dog extends Animal{

public void move(){

System.out.println("Dogs can walk and run");

}

public void bark(){

System.out.println("Dogs can bark");

}

}

public class TestDog{

public static void main(String args[]){

Animal a =new Animal();// Animal reference and object

Animal b =new Dog();// Animal reference but Dog object

a.move();// runs the method in Animal class

b.move();//Runs the method in Dog class

b.bark();

}

}

This would produce the following result:

TestDog.java:30: cannot find symbol

symbol : method bark()

location:classAnimal

b.bark();

^

This program will throw a compile time error since b's reference type Animal doesn't have a method by the name of bark.

**Rules for method overriding:**

* The argument list should be exactly the same as that of the overridden method.
* The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.
* The access level cannot be more restrictive than the overridden method's access level. For example, if the superclass method is declared public, then the overriding method in the subclass cannot be either private or protected.
* Instance methods can be overridden only if they are inherited by the subclass.
* A method declared final cannot be overridden.
* A method declared static cannot be overridden but can be re-declared.
* If a method cannot be inherited, then it cannot be overridden.
* A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.
* A subclass in a different package can only override the non-final methods declared public or protected.
* An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws exceptions or not. However the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.
* Constructors cannot be overridden.
  1. **Dynamic Method Dispatch:**

While the examples in the preceding section demonstrate the mechanics of method overriding, they do not show its power. Indeed, if there were nothing more to method overriding than a name space convention, then it would be, at best, an interesting curiosity, but of little real value. However, this is not the case. 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.

Let’s begin by restating an important principle: a superclass reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time. Here is how. 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. 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( )**. As the output shows, the version of **callme( )** executed is determined by the type of object being referred to at the time of the call. Had it been determined by the type of the reference variable, **r**, you would see three calls to **A**’s **callme( )** method.

* 1. **Abstract Class & Methods:**

There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes you will 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. One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method.

Consider the class **Triangle**. It has no meaning if **area( )** is not defined. In this case, you want some way to ensure that a subclass does, indeed, override all necessary methods. Java’s solution to this problem is the abstract method. You can require that certain methods be overridden by subclasses by specifying the abstract type modifier. These methods are sometimes referred to as subclasser responsibility because they have no implementation specified in the superclass. Thus, a subclass must override them—it cannot simply use the version defined in the superclass.

To declare an abstract method, use this general form:

***abstract type name(parameter-list);***

As you 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, you simply use the **abstract** keyword in front of the **class** keyword at the beginning of the class declaration. There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the **new** operator. Such objects would be useless, because an abstract class is not fully defined. Also, you cannot declare abstract constructors, or abstract static methods. Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared **abstract**.

// Using abstract methods and classes.

abstract class Figure {

double dim1;

double dim2;

Figure(double a, double b) {

dim1 = a;

dim2 = b;

}

// area is now an abstract method

abstract double area();

}

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 AbstractAreas {

public static void main(String args[]) {

// Figure f = new Figure(10, 10); // illegal now

Rectangle r = new Rectangle(9, 5);

Triangle t = new Triangle(10, 8);

Figure figref; // this is OK, no object is created

figref = r;

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

figref = t;

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

}

}

As the comment inside **main( )** indicates, it is no longer possible to declare objects of type **Figure**, since it is now abstract. And, all subclasses of **Figure** must override **area( )**. To prove this to yourself, try creating a subclass that does not override **area( )**. You will receive a compile-time error.

Although it is not possible to create an object of type **Figure**, you can create a reference variable of type **Figure**. The variable **figref** is declared as a reference to **Figure**, which means that it can be used to refer to an object of any class derived from **Figure**. As explained, it is through superclass reference variables that overridden methods are resolved at run time.

* 1. **Using final Keyword with Inheritance (Final Classes & Variables):**

The keyword final has three uses. First, it can be used to create the equivalent of a named constant. This use was described in the preceding chapter. The other two uses of final apply to inheritance. Both are examined here.

**Using final 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. The following fragment illustrates final:

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!");

}

}

Because **meth( )** is declared as **final**, it cannot be overridden in **B**. If you attempt to do so, a compile-time error will result.

**Using final 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. As you might expect, 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.

Here is an example of a final class:

final class A {

// ...

}

// The following class is illegal.

class B extends A { // ERROR! Can't subclass A

// ...

}

As the comments imply, it is illegal for **B** to inherit **A** since **A** is declared as **final**.