# Chapter Four

# Object and Classes

One of the fundamental ways in which we handle complexity is abstractions. An abstraction denotes the essential properties and behaviors of an object that differentiate it from other objects. The essence of OOP is modelling abstractions, using classes and objects.

## What is class and object in java?

* A class denotes a category of objects, and acts as a **blueprint** for creating such objects. A class models an abstraction by defining the properties and behaviors for the objects representing the abstraction.
* The properties of an object of a class are also called attributes, and are defined by fields in Java.
* A field in a class definition is a variable which can store a value that represents a particular property.
* The behaviors of an object of a class are also known as operations, and are defined using methods in Java.
* Fields and methods in a class definition are collectively called members.
* Defining Classes:A class declaration introduces a new reference type. It has the following general syntax:

<class modifiers> class <class name> <extends clause> <implements clause> //Classheader

{ // Class body
     <field declarations>
     <method declarations>
     <nested class declarations>
     <nested interface declarations>
     <constructor declarations>
     <initializer blocks>
}

* The keyword class and class name are mandatory in a class declaration.
* Access modifiers: Determine from where the class can be accessed or how the class can be used: abstract, final, protected and public. If you do not specify an access modifier, the default access is assumed.
* Class name and the file name of the class are the same with extension **.java**

**E.g if the class name is Student then the file name of the class is Student.java**

* Members declared **static** belong to the class and are called **static members**, and non-static members belong to the objects of the class and are called **instance members.**
* An object is an instance of a class. The object is constructed using the class as a blueprint and is a concrete instance of the abstraction that the class represents.
* The process of creating objects from a class is called **instantiation.** And an object must be created before it can be used in a program.
* The process of creating objects usually involves the following steps:
1. Declaration of a variable to store the object reference:

<ClassName> <variableName> ;

Student stud1,stud2;

1. Creating an object.

This involves using the **new** operator in conjunction with a call to a constructor, to create an instance of the class.

<refrancevariable1>=new <class constructor >

stud1=new Student();

stud2=new Student();

1. Creating array Object

<ClassName> <ObjectName> [] = new <ClassName[<arraySize>];

For(int i=0;i<arraySize;i++){

 <ObjectName>[i] = new <Constructor>

 <ObjectName>[i].<Variablename> = <Value>

}

Student stud[] = new Student[5];

for(int i=0;i<5;i++){

 stud[i].setName(“Abebe”);

}

Example: Declaring a Class with a Method and Instantiating an Object of a Class

 // GradeBook.java

 // Class declaration with one method.

 public class GradeBook {

 // display a welcome message to the GradeBook user

 public void displayMessage() {

 System.out.println( "Welcome to the Grade Book!" );

 } // end method displayMessage

 } // end class GradeBook

 // GradeBookTest.java

 // Create a GradeBook object and call its displayMessage method.

 public class GradeBookTest {

 // main method begins program execution

 public static void main( String args[] ){

 // create a GradeBook object and assign it to myGradeBook

 GradeBook myGradeBook = new GradeBook();

 // call myGradeBook's displayMessage method

 myGradeBook.displayMessage();

 } // end main

 }

## Defining a Method

* A method is a self-contained block of code that performs specific operations on the data by using some logic.
* Defining a method in a program is called method declaration. A method consists of the following elements: Name, Parameter, Return type, Access modifier and arguments.

The following is the syntax for writing a method in Java:

<modifier> <returnType> <methodName> ( <Parameters>) {

// body of the method. The code statements go here.

}

* The <modifier>specifies the method further, such as its visibility, and <returnType>defines the data type that will be returned by the method. While zero or more modifiers may be used, <returnType> and <methodName>are mandatory. The parameters are used to pass data into the method.

For example, consider the following method:

public int square (int number) {

return number\*number;

}

* Example 2

|  |  |
| --- | --- |
| class ClassRoom {private String roomNumber;private int totalSeats = 60;private static int totalRooms = 0;void setRoomNumber(String rn) {roomNumber = rn; } String getRoomNumber() { return roomNumber; } void setTotalSeats(int seats) { totalSeats = seats; } int getTotalSeats() { return totalSeats; } } | class ClassRoomManager { public static void main(String[] args){ ClassRoom roomOne = new ClassRoom(); roomOne.setRoomNumber("MH227"); roomOne.setTotalSeats(30); System.out.println("Room number: " + roomOne.getRoomNumber()); System.out.println("Total seats: " + roomOne.getTotalSeats());}} |

## Exercise :

1. Show how a method called myMeth( )is declared if it has a return type of double and has two int parameters called a and b.

## Using Method with Array:

* To receive an array reference through a method call, the method's parameter list must specify an array parameter.

Example:

void modifyArray( int b[] ){ …}

* To pass an array argument to a method, specify the name of the array without any brackets.

Example to invoke a method with array parameter defined above

 int b[ ]={1,2,3,4,5}

 modifyArray(b);

* Two ways to pass arguments in method calls: **pass-by-value** and **pass-by-reference.**
* When an argument is passed by value, a copy of the argument's value is passed to the called method. The called method works exclusively with the copy. Changes to the called method's copy do not affect the original variable's value in the caller.
* When an argument is passed by reference, the called method can access the argument's value in the caller directly and modify that data, if necessary.

Example of Passing arrays and individual array elements to methods:

public class PassArray{

 public static void main( String args[] ){

 int array[] = { 1, 2, 3, 4, 5 };

 for ( int i=0;i<5; i++ ){

 System.out.printf( " %d", array[i] );}

 modifyArray( array ); // pass array reference

 System.out.println( "\n\nThe values of the modified array are:" );

 for ( int i=0;i<5; i++ ){

 System.out.printf( " %d", array[i] );}

 System.out.printf("\n\nEffects of passing array element value:\n" +

 "array[3] before modifyElement: %d\n", array[ 3 ] );

 modifyElement( array[ 3 ] ); // attempt to modify array[ 3 ]

 System.out.printf( "array[3] after modifyElement: %d\n", array[ 3 ] );

 } // end main

 public static void modifyArray( int array2[] )

 {

 for ( int counter = 0; counter < array2.length; counter++ )

 array2[ counter ] \*= 2;

 } // end method modifyArray

// multiply argument by 2

public static void modifyElement( int element ){

element \*=2;

System.out.printf("Value of element in modifyElement:%d\n",element);

 // end method modifyElement

 } // end class PassArray

Output of the program above:

Effects of passing reference to entire array:

The values of the original array are:

 1 2 3 4 5

The values of the modified array are:

 2 4 6 8 10

Effects of passing array element value:

array[3] before modifyElement: 8

Value of element in modifyElement: 16

array[3] after modifyElement: 8

## Instance Members and static members

* Member of the class is classified in to two :
* Instance member
* Static member
* Instance Member: it is members of a class and created for each object of the class.
* **Instance variable:** A field that is allocated when the class is instantiated, that is, when an object of the class is created. Also called non-static field. Unique to each instance (object) of that class (that is, each object has its own set of instance fields)
* **Instance method:** A method that belongs to an instance of the class. Objects of the same class share its implementation. Instance methods can access both instance and static data and methods.
* These are instance variables and instance methods of an object. They can only be accessed or invoked through an object reference.
* Static member: members should only belong to the class, and not be part of any object created from the class. They can be accessed or invoked either by using the class name or through an object reference. Distinguished from instance members in a class definition by the keyword static in their declaration.
* **Static variable**: A field that is allocated when the class is loaded. It belongs to the class and not to any object of the class. Also called **static field** and **class variable.**it is unique to the entire class and shared by all instances (objects) of that class
* **Static method**: A method which belongs to the class and not to any object of the class. Also called **class method.** Static methods can access only static data and invoke other static methods.
* Static element of a class is visible to all the instances of the class, if one instance makes a change to it; all the instances see that change.
* A static method can only access the static members of the class. In other words a method declared static in a class cannot access the nonstatic variables and methods of the class.
* A class can also have a static code block outside of any method; i.e. the code block does not belong to any method.
* Non static member can access static members.
* Static code is not executed in the context of an object, therefore the references this and super are not available.

Example:

|  |  |
| --- | --- |
| class StaticExample {static int staticCounter =0;//static int counter = 0;//insranceStaticExample() { staticCounter++; counter++; } } | class RunStaticExample {public static void main(String[] args) {StaticExample se1 = new StaticExample();StaticExample se2 = new StaticExample();System.out.println("Value of staticCounter for se1: " +se1.staticCounter); System.out.println("Value of staticCounter for se2: " +se2.staticCounter);System.out.println("Value of counter for se1: " + se1.counter);System.out.println("Value of counter for se2: " + se2.counter);StaticExample.staticCounter = 100;System.out.println("Value of staticCounter for se1: " +se1.staticCounter); System.out.println("Value of staticCounter for se2: " +se2.staticCounter); } } |

The output of the above code is :

Value of staticCounter for se1: 2

Value of staticCounter for se2: 2

Value of counter for se1: 1

Value of counter for se2: 1

Value of staticCounter for se1: 100

Value of staticCounter for se2: 100

* A static method defined in a class cannot access the nonstatic variables and methods of the class.

Example:

class MyClass {

String salute = "Hello";

public static void main(String[] args){

System.out.println("Salute: " + salute);//4

 }

 }

This code generates a compiler error on line 4, because it tries to access a nonstatic variable from inside a static method.

Exarcise: what is the error of this code?

class MyClass {

 int i = 5;

 static int j = 7;

 public static void printSomething () {

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

 System.out.println("j: " + j);

 } }

## Constructor

* A constructor is a variety of method that is called when an object of the class is created

Using new.

* Constructors are used to initialize objects.
* A constructor must have the same name as the class to which it belongs.
* Arguments for a constructor are given in parentheses after the class name.
* A constructor is defined very much like any ordinary method except that it does not have a type returned and does not even include a void in the constructor heading.
* A constructor has the following general syntax:

<accessibility modifier> <class name> (<formal parameter list> )<throws clause> // Constructor header
  { // Constructor body
     <local variable declarations>
     <nested local class declarations>

<statements>
  }

Example :

**public class Name {**

 **Name() { // (1)**

 **System.out.println("Constructor");**

 **}**

 **void Name() { // (2)**

 **System.out.println("Method");**

 **}**

 **public static void main(String[] args) {**

 **new Name().Name(); // (3) Constructor call followed by method call.**

 **}**

 **}**

Output from the program:

 Constructor

 Method

* **Constructors can be :**
* **Default constructor**
* implicit default constructor
* explicit default constructor
* **explicit constructor**
* A default constructor is a constructor without any parameters. it has the following signature:

<class name>(){…}

* **Implicit default constructor:** If a class does not specify any constructors, then an implicit default constructor is supplied for the class. The implicit default constructor is equivalent to the following implementation:

<class name>() { super(); } // No parameters.Calls superclassconstructor

Example :

**class Student {**

 **// Fields**

 **int age; // age**

 **String name; // name**

 **// No constructors**

 **//...**

**}**

**class StudentDemo {**

 **// ...**

 **Student stud = new Student();//(1)Call of implicit default constructor.**

**}**

* Explicit default constructor: creating constructor with no argument explicitly inside the class.

Example:

**class Student {**

 **// Fields**

 **int age; // age**

 **String name; // name**

 **Public Student(){//explicit default constructor.**

 **age=25;**

 **name=abebe**

 **//...**

**}**

**class StudentDemo {**

 **// ...**

**Student stud = new Student();//(1)Call of explicit default constructor.**

**}**

* **explicit constructor**

If a class defines any explicit constructors that means constructor with argument, it can no longer rely on the implicit default constructor to set the state of the objects. And any attempt to call the implicit default constructor will be flagged as a compile time error.

Example :

**class Student {**

 **// Fields**

 **int age; // age**

 **String name; // name**

 **Public Student(String name,int age){//explicit default constructor.**

 **this.age=age;**

 **this.name=name;**

 **//...**

**}**

**class StudentDemo {**

 **// ...**

 **Student stud = new Student(“abebe”,25);//(1)Call of explicit constructor.**

 **Student stud1=new Student();//(3) Compile time error.**

**}**

* From inside a constructor of a class, you can call another constructor of the same class, or a constructor of the superclass. You use the keyword **this** to call another constructor in the same class, and use the keyword **super** to call a constructor in the superclass. If you use either this or super, it must appear in the **beginning of the code block of the constructor.**
* Unlike other methods, the constructors are not inherited. If the superclass has constructors and the subclass does not, the compiler assumes that the subclass does not have any constructor and creates the default constructor.

## Nested Classes



* The Java programming language allows you to define a class within another class. This kind of class is called a **nested class.**
* In this context, the top-level class is also called an outer class or enclosing class.
* Any nested class can be declared in any block of code such as a class or a method.
* A nested class is a member (like a variable or a method) of another class. It would look like the following:

class <OuterClassName> {

// variables and methods for the outer class

…

class <NestedClassName> {

// variables and methods for the nested class

…

}

}

* Nested and Outer classes have Access to each Other’s private members
* Why Use Nested Classes?
* It is a way of logically grouping classes that are only used in one place.
* It increases encapsulation.
* Nested classes can lead to more readable and maintainable code.
* Nested classes are divided into two categories:
* static nested class
* non-static class, is also called an inner class

### Static nested class:

* A static nested class is associated with its outer class.
* A static nested class can have non static instance variables and methods, but an object of a static inner class has no connection to an object of the outer class.
* A static nested class **cannot refer directly** to instance variables or methods defined in its enclosing class, it can use them only through an object reference.
* Static nested classes are accessed using the enclosing class name:

***OuterClass.StaticNestedClass***

* *For example, to create an object for the static nested class, use this syntax:*

***OuterClass.StaticNestedClass*** *nestedObject* ***= new OuterClass.StaticNestedClass( );***

**Example for demonstrating static nested class:**

**class TestStaticNested** {

 public static void main(String[] args) {

 String ext = "From external class";

new MyTopLevel( ).gateToStatic( );

//to create an object of static nested class

**MyTopLevel.StaticNested sn = new MyTopLevel.StaticNested( );**

sn.accessStaticNested(ext);

 }

 }

**class MyTopLevel**{

private static String top = "From top level class";

String insst=”instance member”;

 public static void gateToStatic(){

 StaticNested s = new StaticNested();

 s.accessStaticNested(top);

 }

 static class StaticNested {

 public void accessStaticNested(String st) {

 System.out.println(insst);**//error ,it must be accessed through object reference**

 System.out.println(st);

}

 }

}

### non-static class, is also called an inner class:

* An inner class (nonstatic nested class) is associated with an instance of its outer class.
* Unlike an external class, an inner class has unlimited access to the members of the outer class, including the private members, because the inner class itself is a member of the outer class.
* Because an inner class, being a nonstatic member, is associated only with an instance of its outer class, you cannot define a static member inside an inner class.
* An instance of InnerClass can exist only within an instance of OuterClass and has direct access to the methods and fields of its enclosing instance.
* To instantiate an inner class, you must first instantiate the outer class. Then, create the inner object within the outer object with this syntax:

**OuterClass outerObject=new OuterClass();**

**OuterClass.InnerClass innerObject = outerObject.new InnerClass();**

**Or**

**OuterClass.InnerClass innerObject = new OuterClass().new InnerClass();**

**Example for demonstrating inner class:**

 class TestNested {

public static void main(String[] args) {

String ext = "From external class";

MyTopLevel mt = new MyTopLevel();

 mt.createNested();

MyTopLevel.MyInner inner = mt.new MyInner();

inner.accessInner(ext);

}

 }

 class MyTopLevel{

 private String top = "From Top level class";

 MyInner minn = new MyInner();

 public void createNested() {

 minn.accessInner(top);

}

class MyInner {

 public void accessInner(String st) {

 System.out.println(st);

 }

 }

 }

The output from executing above code follows:

From Top level class

From external class

Exercise:

Consider the following code:

1. class ConventionCenter{

2. class AudioSystem {

3. public void test(String say){

4. System.out.println(say);

5. }

6. }

7. }

8. public class JobFair{

9. public static void main(String [] args) {

10. String st = "Hello";

11. // Insert code here.

12. as.test(st);

13. }

14. }

Depending on the above java program, which of the following inserted independently at line 11 will generate the output Hello?

A. ConventionCenter.AudioSystem as = new ConventionCenter( ).new AudioSystem( );

B. AudioSystem as = new ConventionCenter( ).new AudioSystem( );

C. ConventionCenter.AudioSystem as = new AudioSystem( );

D. ConventionCenter.AudioSystem as = new ConventionCenter( ).AudioSystem( );

E. ConventionCenter cc = new ConventionCenter( );

ConventionCenter.AudioSystem as1 = cc.new AudioSystem( );

## The Garbage Collector

* Some object-oriented languages require that you keep track of all the objects you create and that you explicitly destroy them when they are no longer needed. Managing memory explicitly is tedious and error-prone.
* The Java platform allows you to create as many objects as you want (limited, of course, by what your system can handle), and you don't have to worry about destroying them.
* The Java runtime environment deletes objects when it determines that they are no longer being used. This process is called *garbage collection*.
* An object is eligible for garbage collection when there are no more references to that object.
* References that are held in a variable are usually dropped when the variable goes out of scope.
* Or, you can explicitly drop an object reference by setting the variable to the special value null.
* Benefits of garbage collection:
* Frees up programmers from having to manage memory.
* Prevents memory leaks — each object is tracked down and disposed off as soon as it is no longer used.
* Prevents deallocation of objects that are still in use or have already been released.

## Packages

* A package is a grouping of related types providing access protection and name space management.
* You can bundle related classes and interfaces into a group called a package.
* You store all the class files related to a package in a directory whose name matches the name of the package.
* Bundling related classes and interfaces into a package offers the following advantages:
* It makes it easier to find and use classes.
* It avoids naming conflicts. Two classes with the same name existing in two different packages do not have a name conflict, as long as they are referenced by their fully qualified name.( package creates a new namespace.)
* It provides access control.
* Can easily determine related types.

### Defining a Package:

* 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.
* The package statement must be the first line in the source file.
* This is the general form of the package statement:

package packegename;

* For example, the following statement creates a package called MyPackage.

Package MyPackage;

* You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period .The general form of a multileveled package statement is shown here:.

package packegename1. Packegename2. Packegename3;

* **Using Package Members:**
* The types that include a package are known as the package members.
* To use a public package member from outside its package, you must do one of the following:
* Refer to the member by its fully qualified name

Packegename1.classname obj=new Packegename1.classname ( );

Example : Java.util.Scanner sc = new Java.util.Scanner( );

* Import the package member before the top level declaration

 import Packegename1.classname ;

Example : import java.util.Scanner;

* Import the member's entire package

 import Packegename1.classname ;

Example : import java.util.\*;

* **Importing a Package Member:**
* To import a specific member into the current file, put an import statement at the beginning of the file before any type definitions but after the package statement, if there is one.
* In a Java file, the 1. package declaration,2. the import statement, and 3.the class definition must appear in this order.
* example, the following code will not compile because the import statement appears before the package statement:

import otherPackage;

package thisPackage;

class A { }

* For example, if the class contained useful nested classes, you could import top level class and its nested classes by using the following two statements.

 import package.toplevelclass;

 import package.toplevelclass.innerclass;

## Using Access Modifiers

* Access modifiers, also called visibility modifiers, determine the accessibility scope of the Java elements they modify.
* If you do not explicitly use an access modifier with a Java element, the element implicitly has the default access modifier.
* The explicit access modifiers may be used with a class and its members.
* They cannot be used with the variables inside a method.
* The Java language offers three explicit access modifiers, public, protected, and private, and a default modifier, which is enforced when you do not specify a modifier.

### The public Modifier:

* The public modifier makes a Java element most accessible.
* It may be applied to classes, interface and to their members.
* A class, variable, or method, declared as public, may be accessed from anywhere in the Java application. From inside the package and also from outside the package. (Form anywhere.)
* It is not a good object-oriented programming practice to declare the instance variables public. If you declare them public, they could be accessed directly, whereas they should always be accessed through the class methods.
* Example of accessing public class and method from outside the package:

|  |  |
| --- | --- |
| package packageA; import java.util.Scanner; public class PublicClassA { public void displayArea(char shapeType){ Scanner input=new Scanner(System.in); char c=shapeType; switch (c){ case 'T':{ System.out.print("ensert the hight"); double h=input.nextDouble(); System.out.print("ensert the base"); double b=input.nextDouble(); double area=1.0/2.0\*(h\*b); System.out.println("the area of triangle ="+area); }break; case 'R':{ System.out.print("ensert the hight"); double h=input.nextDouble(); System.out.print("ensert the base"); double b=input.nextDouble(); double area=h\*b; System.out.println("the area of rectangle ="+area); }  break; default:System.out.print("enter corect letter"); }}} | package packageB;import packageA.PublicClassA; import java.util.Scanner;public class PublicClassD{ public static void main(String args[]){ PublicClassA pcA=new PublicClassA(); Scanner sc=new Scanner(System.in); System.out.println ("enter R for calculating area”+ “ of Rectangele and /n OR T for calculating the Area”+  “of triangle "); String stype; stype =sc.nextLine(); char stp=stype.charAt(0); pcA.displayArea(stp); }} |

### The private Modifier :

* The private modifier makes a Java element (a class or a class member) least accessible.
* The private
* Modifier cannot be applied to a top-level class.
* It can be applied only to the members of a top-level class.
* A private member of a class may only be accessed from the code inside the same class in which this member is declared.
* It can be accessed neither from any other class nor from a subclass of the class in which it is declared.
* Note: A top-level class cannot be declared private; it can only be public or default (that is, no access modifier is specified).

Example:

1. class PrivateTest {

2.

3. // public int myNumber = 10;

4. private int myNumber = 10;

5. public int getMyNumber(){

6. return myNumber;

7. }

8. }

9. class SubPrivateTest extends PrivateTest {

10. public void printSomething(){

11. System.out.println (" The value of myNumber is " + this.myNumber);

12. System.out.println (" The value returned by the method is " +

this.getMyNumber());

13. }

14. }

15. class TestPrivateTest{

16. public static void main(String[] args) {

17. SubPrivateTest spt = new SubPrivateTest();

18. spt.printSomething();

19. }

20. }

This code will not compile because line 11 will generate a compiler error. You cannot access a private data variable of the parent class directly, as the private class members are not inherited.

* Note: Public and private access modifiers are on the two extremes of access: access from **everywhere** and access from **nowhere outside of the class**.

### The protected Modifier:

* The protected modifier makes a class member more accessible than the private modifier would, but still less accessible than public.
* This modifier may be applied only to class members—that is, the variables, methods, and inner classes—but not to the class itself.
* A class member declared protected is accessible to the following elements:
* All the classes in the same package that contains the class that owns the protected member.
* All the subclasses of the class that owns the protected member. These subclasses have access even if they are not in the same package as the parent class.

For example, consider these two code fragments:

1. package networking;

2. class Communicator {

3. void sendData() {}

4. protected void receiveData() {}

5. }

1. package internetworking;

2. import networking.\*;

3. class Client extends Communicator {

4. void communicate(){

5. receiveData();

6. }

7. }

The class Client is a subclass of the class Communicator. But both classes are in different packages. The method receiveData()is declared protected in the class Communicator. It may be called from the class Client (line 5), even though Communicator and Client are in different packages. This is because Client is a subclass of Communicator.

### The Default Modifier:

* When you do not specify any access modifier for an element, it is assumed that the access is default.
* If an element does not explicitly use any modifier, the default access is implied. It may be applied to a class, a variable, or a method.
* A class or a class member declared default (no access modifier specified) is accessible from anywhere (classes or subclasses) in the same package in which the accessed class exists.

As an example, consider the following code fragment:

1. package internetworking;

2. import networking.\*;

3. class Client extends Communicator {

4. void communicate(){

5. receiveData();

6. sendData(); // compiler error.

7. }

8. }

1. package networking;

2. class Communicator {

3. void sendData() {}

4. protected void receiveData() {}

5. }

At the first Line 6 would generate a compiler error, because the method sendData()is declared default in the class Communicator, which is in a different package.

**Note**: the difference between the protected modifier and the default modifier. A member with a default access can be accessed only if the accessing class belongs to the same package, whereas a member with the protected modifier can be accessed not only from the same package but also from a different package if the accessing class is a subclass of the accessed class.

**Note:** You cannot specify any modifier for the variables inside a method.



## Understanding Usage Modifiers

* The access modifiers specify the access behavior (from where we can access?), while usage modifiers that specify how a class or a class member can be used.
* Some of them, such as final, abstract, and static….

### The final Modifier:

* The final modifier may be applied to a class, a method, or a variable.
* The specific meaning slightly depends upon the element it applies to.
* If the element **declared final is a variable**, that means the value of the variable is **constant**, and cannot be changed.
* If a **class is declared final**, it means the class cannot be **extended.**
* If a method is **final method** that means it cannot be **overridden.**

Example :

1. class Calculator {

2. final int dime = 10;

3. int count = 0;

4. Calculator (int i) {

5. count = i;

6. }

7. }

8. class RunCalculator {

9. public static void main(String[] args) {

10. final Calculator calc = new Calculator(1);

11. calc = new Calculator(2); // compiler error.

12. calc.count = 2; //ok

13. calc.dime = 11; // compiler error.

14. System.out.println("dime: " + calc.dime);

15. }

16. }

The code lines 11 and 13 will generate compiler errors because they attempt to modify the values of the final variables calc and dime, respectively. However, note that line 12 will compile fine, which shows that the final object reference may be used to modify the value of a non-final variable.

If you comment out lines 11 and 13, the code will compile, and the result of execution or the output will be as follows:

dime: 10

**Note:** Although a final object reference may not be modified, it can be used to modify the value of a nonfinal variable in the object to which the final object reference refers.

* If you declare a final method inside a non-final class, it will be legal to extend the class, but you cannot override the final method of the parent class in the subclass. Similarly, you can pass the final variable to a method through arguments, but you cannot change their value even inside the method.

### The static Modifier:

* The static modifier can be applied to variables, methods, and a block of code inside a method. The static elements of a class are visible to all the instances of the class. As a result, if one instance of the class makes a change to a static element, all the instances will see that change.
* A static method cannot access the non-static variables and methods of the class in which it is defined. Also, a static method cannot be overridden as non-static.

### The abstract Modifier:

* The abstract modifier may be applied to a class or a method, but not to a variable.
* **A class** that is declared abstract **cannot be instantiated**. Instantiation of an abstract class is not allowed, because it is not fully implemented yet.
* There is a relationship between an abstract class and an abstract method.
* If a class has one or more abstract methods, it must be declared abstract.
* Note: A class with one or more abstract methods must be declared abstract. However, a class with no abstract method may also be declared abstract. An abstract class cannot be instantiated.
* Example :

1. abstract class Shape {

2. abstract void draw(); //Note that there are no curly braces here.

3. void message() {

4. System.out.println("I cannot live without being a parent.");

5. }

6. }

7. class Circle extends Shape {

8. void draw() {

9. System.out.println("Circle drawn.");

10. }

11. }

12. class Cone extends Shape {

13. void draw() {

14. System.out.println("Cone drawn.");

15. }

16. }

17. public class RunShape {

18. public static void main(String[] args) {

 //Shape sh=new Shape( ); //if we create the object it is compile error

19. Circle circ = new Circle( );

20. Cone cone = new Cone( );

21. circ.draw( );

22. cone.draw( );

23. cone.message( );

24. }

25. }

The output follows:

Circle drawn.

Cone drawn.

I cannot live without being a parent.

### The native Modifier:

* The native modifier can only apply to a method.
* Like abstract, the keyword native indicates that the implementation of the method exists elsewhere.
* But the difference between abstract and native is :
* In case of abstract, the implementation may exist in a subclass of the class in which the abstract method is declared.
* In case of native, the implementation of the method exists in a library outside of the JVM.
* The native method is usually implemented in a non-Java language such as C or C++.
* Example :

1. class MyNativeExample {

2. Private int a;

3. native void myNativeMethod();//it is implemented in non-java language 4. }

# *Important Concepts(See Illustrative Example By Your Self)*

* **“*this*”** Key word in java
	+ There can be a lot of usage of **java this keyword**. In java, this is a **reference variable** that refers to the current object.

 

* + ***Usage of java this keyword***

Here is given the 6 usage of java this keyword.

1. this can be used to refer current class instance variable.
2. this can be used to invoke current class method (implicitly)
3. this () can be used to invoke current class constructor.
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this can be used to return the current class instance from the method.

**Suggestion:** If you are beginner to java, lookup only three usage of this keyword.

* **“*super*”** keyword in java
	+ The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.
	+ Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.
	+ ***Usage of Java super Keyword***
1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super () can be used to invoke immediate parent class constructor.