**Class Declaration and Modifiers**

A class may be a top-level class, nested class, derived class, super class, or an anonymous class—a class without a name. All these classes are described here.

1. Top-level class: It is a class that is not a nested class.

2. Nested class: It is a class that is declared within the body of another class or interface. There are four types of nested classes:

1. Nested static class: It is like any other static member of the enveloping class.

2. Nested non-static classes are also called inner classes and it can be classified as follows:

(a) Member inner class: It has properties like any other member of the outer class.

(b) Local class: A local class is defined within a method or code block.

(c) Anonymous class: As the name indicates, it is the inner class without a name.

3. Derived class or subclass: It is a class that is derived from (or extends) another class.

4. Super class: It is a class from which another class or other classes are derived.

5. Generic classes: These are classes that declare one or more type parameters.

6. Final class: It is a class that cannot be extended.

Access Modifiers - controls the access level

Non-Access Modifiers - do not control access level, but provides other functionality

**Non-Access Modifiers**

For **classes**, you can use either final or abstract:

Modifier

final: The class cannot be inherited by other classes (You will learn more about inheritance in the Inheritance chapter)

abstract: The class cannot be used to create objects (To access an abstract class, it must be inherited from another class. You will learn more about inheritance and abstraction in the Inheritance and Abstraction chapters)

For **attributes and methods**

|  |  |
| --- | --- |
| **Modifier** | **Description** |
| final | Attributes and methods cannot be overridden/modified |
| static | Attributes and methods belongs to the class, rather than an object |
| abstract | Can only be used in an abstract class, and can only be used on methods. The method does not have a body, for example **abstract void run();**. The body is provided by the subclass (inherited from). You will learn more about inheritance and abstraction in the [Inheritance](https://www.w3schools.com/java/java_inheritance.asp) and [Abstraction](https://www.w3schools.com/java/java_abstract.asp) chapters |
| transient | Attributes and methods are skipped when serializing the object containing them |
| synchronized | Methods can only be accessed by one thread at a time |
| volatile | The value of an attribute is not cached thread-locally, and is always read from the "main memory" |

**These are:**

**#1) Default:**Whenever a specific access level is not specified, then it is assumed to be ‘default’. The scope of the default level is within the package.

**#2) Public:**This is the most common access level and whenever the public access specifier is used with an entity, that particular entity is accessible throughout from within or outside the class, within or outside the package, etc.

**#3) Protected:**The protected access level has a scope that is within the package. A protected entity is also accessible outside the package through inherited class or child class.

**#4) Private:**When an entity is private, then this entity cannot be accessed outside the class. A private entity can only be accessible from within the class.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Access Specifier** | **Inside Class** | **Inside Package** | **Outside package subclass** | **Outside package** |
| Private | Yes | No | No | No |
| Default | Yes | Yes | No | No |
| Protected | Yes | Yes | Yes | No |
| Public | Yes | Yes | Yes | Yes |

**Class Members**

The class members are declared in the body of a class.

These may comprise fields (variables in a class), methods, nested classes, and interfaces. The members of a class comprise the members declared in the class as well as the members inherited from a super class.

The scope of all the members extends to the entire class body. The fields comprise two types of variables:

1. Non-static variables: These include local and instance variables which vary in scope and value.

(a) Instance variables: These variables are individual to an object and an object keeps a copy of these variables in its memory.

(b) Local variables: These are local in scope and not accessible outside their scope.

2. Class variables: These variables are also qualified as static variables. The values of these variables are common to all the objects of the class. The class keeps only one copy of these variables and all the objects share the same copy. As class variables belong to the whole class, these are also called class variables.

Key Points :

1. Instance variables across different objects have different values, whereas class variables across different objects have only one value.
2. Class variables are initialized when a class is first loaded into JVM memory,
3. whereas instance variables are initialized when an instance is created.

**Example**

class CustomerId {

// initializing count to zero.

static int count = 0;// static variable

int id;// instance variable

public CustomerId() { // Every time the constructor runs, it increments count.

count = count + 1;

id = count;

}

public int getId()

{ return id; }

}

public class Application

{

public static void main (String args[])

{

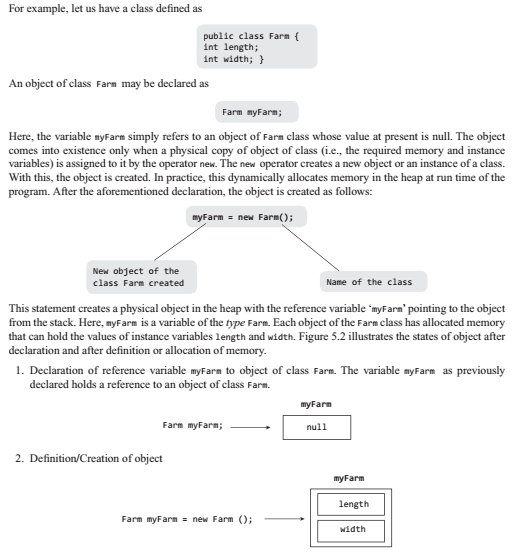
CustomerId obj = new CustomerId();

System.out.println("Customer ID number:"+obj.getId());

}

}

**Declaration of Class Objects**



**Assigning One Object to Another**

class Farm

{

double length;

double width;

double area(){return length\*width;}

}

public class FarmExe1

{

// class with main method().

public static void main (String args [])

{

Farm farm1 = new Farm();//defining an object farm1

Farm farm2 = new Farm(); //defining an object farm2

farm1.width = 20.0;// accessing the variables

farm1.length = 40.0;

System.out.println ("Area of farm1 = " + farm1.area());

farm2 = farm1;

System.out.println ("Area of farm2 = " + farm2.area());

farm2.width = 25;

System.out.println ("Width of farm2 = " + farm2.width);

System.out.println ("Width of farm1 = " + farm1.width);

System.out.println ("Area of farm1 = " + farm1.area());

}

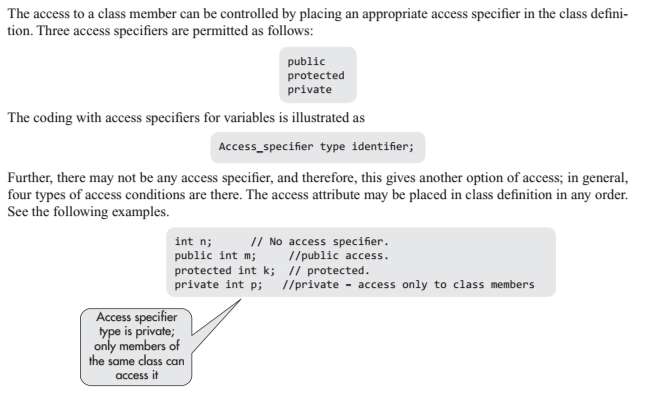
}

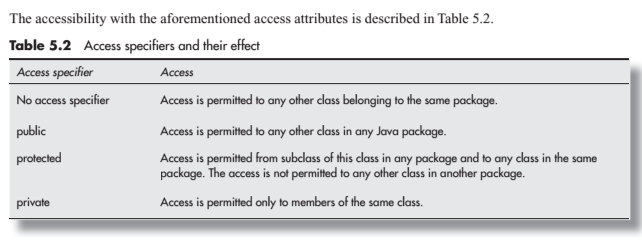
it becomes clear that object names farm1 and farm2 are only references to the objects. When we assign farm1 to farm2, we are in fact only assigning the reference, that is, farm2 becomes another reference to the same object represented by farm1. The farm2 is not an independent new object

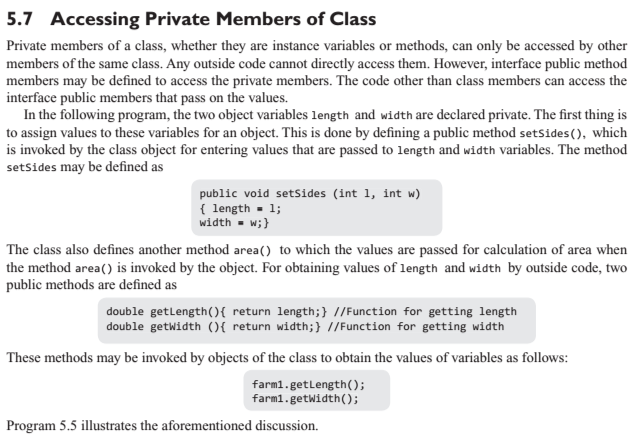
Diagram

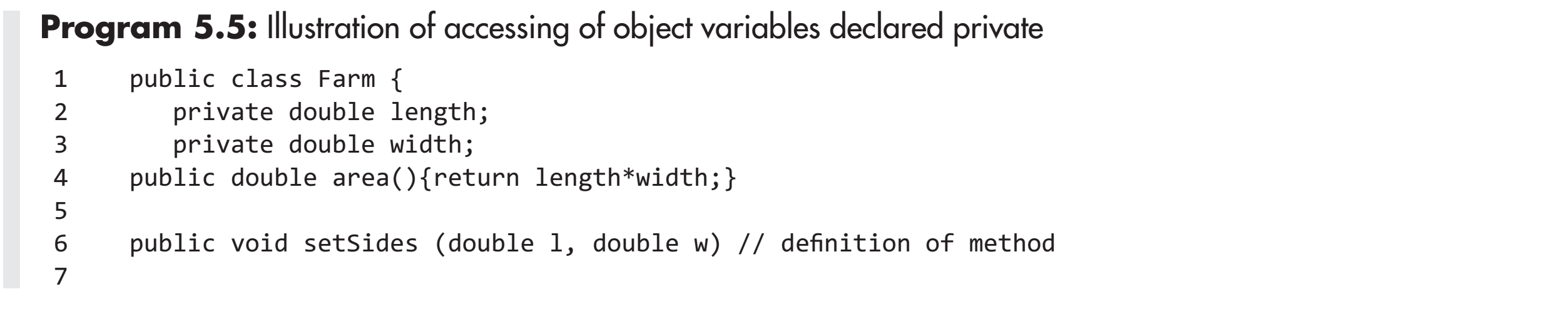
Description automatically generated

**Access Control for Class Members**









Text, letter

Description automatically generated

Text, letter

Description automatically generated

Text

Description automatically generated

**Constructor Methods for classes**

In the aforementioned programs, we have assigned the values to object variables through a public method like setSides(), which is invoked by the object after its definition. It is not called instantly during the object creation. The constructor method can be used for this purpose. Constructor is also a public method with the same name as the class. It has no type; further, it does not return any value, and it only constructs and initializes an object of a class. Every time an object of the class is created, the constructor method is automatically called.

Program 5.7 illustrates a parametric constructor method. The object is constructed by the values of the parameters declared along with the definition of object.

class FarmT {

double length;// declaration of variables

double width;

FarmT (double l, double w) //Constructor method

{

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

length = l; width = w;

}

double area() {return length\*width;}

}// end of class \

class FarmExe// class with main method

{

public static void main (String args [])

{

FarmT farm1 = new FarmT(50.0, 20.0); //creating new object

double farmArea=0;

farmArea = farm1.area();

System.out.println ("Area of farm1 = " + farmArea);

}

}

**Overloaded Constructor Methods**

Like other methods, the constructors may also be overloaded. The name of all the overloaded constructor methods is same as the name of the class, but parameters have to be different either in number or type or order of parameters in each definition. In the case of objects, the parameters are the object variables; however, some of these can have default values in the definition of constructor method itself.

class Farm5

{

private double length;// declaration of variables

private double width;

Farm5 (double l, double w) //Parametric constructor

{

System.out.println("Constructor1 called");

length = l; width = w;

}

Farm5 (double w)// Constructor with one parameter

{

System.out.println("Constructor2 called");

length = 60.0;// Length has default value 60

width = w;

}

Farm5()// Constructors with default values

{

System.out.println("Constructor3 called");

length = 60.0; //Both the dimensions have default values

width = 20.;

}

double area(){return length\*width;}

}// End of class Farm

class FarmExe5

{// class with main method

public static void main (String args [])

{

Farm5 farm1 = new Farm5(50.0, 20.0);// calls constructor1

Farm5 farm2 = new Farm5 (30.0);// calls constructor2

Farm5 farm3 = new Farm5();// creating an object, calls constructor3

double farmArea1,farmArea2,farmArea3;

farmArea1 = farm1.area();

farmArea2 = farm2.area();

farmArea3 = farm3.area();

System.out.println ("Area of farm1 = " + farmArea1);

System.out.println ("Area of farm2 = " + farmArea2);

System.out.println ("Area of farm3 = "+ farmArea3);

}

}

**Nested Classes**

A nested class is one that is declared entirely in the body of another class or interface. Interface is a type of class that defines only abstract methods

The class, which is nested, exists only as long as the enveloping class exists. Therefore, the scope of inner class is limited to the scope of enveloping class.

There are four types of nested class.

Nested static class is like any other static member of the enveloping class

Nested non-static classes are also called inner classes, which may be

(a) member inner class,

(b) local class, and

(c) anonymous class.

This inner class is a member of the enveloping class just like any other member of the enveloping class, and therefore, it has access to all the members of the enveloping class including the private members. However, the enveloping class does not have direct access to members of the nested (inner) class.

The following points regarding non-static nested classes should be noted.

1. The inner class has access to all the members of the enveloping class including the members declared public, protected, or private.

2. The enveloping class does not have direct access to nested class members. It is only through the reference of inner class.

3. An instance of inner class outside the enveloping class can only be created with an instance of outer class.

4. The object of inner (nested) class may be declared only in the scope of outer (enveloping) class.

5. A method of inner class cannot be directly accessed by the object of outer class.

The access is through the object of inner class by a fully qualified name for accessing, which is given in the following example. Here, objectOuter and objectInner are objects of outer and inner classes, respectively.

objectOuter.objectInner.methodName();

**1.**Example for **Nested static**

class Outer1

{ // Enveloping class

static private double length;// declaration of variables

static private double width;

Outer1( double x, double y)// constructor

{length = x; width = y; }

double area (){ return length\* width; }

static class Inner2// nested class

{

private double rate;

Inner2(double r ){ rate = r;}

void displayInner()

{

System.out.println ("The cost of fencing = " + rate\*2\* (length + width));

}

}// End of class Inner2

}// End of class Outer1

// Below is class with main method public

class NestedStatic

{

// class with main method

public static void main (String args [])

{

Outer1.Inner2 inn1 = new Outer1.Inner2(10);// object

Outer1 Obj = new Outer1 (40,30);

System.out.println ("Obj\_Area = " + Obj.area());

inn1.displayInner();

}

}

**2.**Example for **Member inner class**

class Outer

{

private double length;// declaration of variables

private double width;

Outer (double x, double y)// Constructor method

{length = x; width = y;}

double area(){ return length\* width;}// definition of method

void displayOuter()

{

Inner iner = new Inner();// iner is object of inner class

iner.displayInner();

}

class Inner

{

double rate = 10.0;// declaration of variables

double perimeter = 2\*( length + width);

double fencingCost = rate \* perimeter;

void displayInner()

{// definition of another method

System.out.println ("The cost of fencing = " + fencingCost);

}

}

}

class NEST

{

public static void main (String args [])

{

Outer obj = new Outer (40.0, 20.0);

obj.displayOuter();// accessing the method

System.out.println ("Area of Obj = " + obj.area());

}

}

**3.**Example for **local class**

public class LocalDemo

{

public static void main(String[] args)

{

class Local

{// local class defined

private int x;// declaring instance variable

Local(int a) {x = a;}// constructor of local class

public void display()

{

System.out.println("Cube of "+x+" = " + Math.pow(x,3));

}

}// End of local class

Local local = new Local(20);// creating an object

local.display();

}

}

**4.**Example for **Anonymous classes**

As the name indicates, anonymous classes are inner classes without a name. It is defined inside another class. Because class has no name it cannot have a constructor method and its objects cannot be declared outside the class. Therefore, an anonymous class must be defined and initialized in a single expression. An anonymous class may be used where the class has to be used only once.

new class-name ([argument list]){class body}

Illustration of anonymous class implementing an interface

interface InterFace// interface definition

{

public void display();// abstract method

}

public class AnonymousDemo1

{

public static void main (String args[])

{//inner anonymous class definition

InterFace f = new InterFace ()

{// creating reference of interface

public void display()

{

System.out.println( "Anonymous class implements interface.");

}

};// The anonymous implementor of Interface closes

f.display(); //The reference f calls the method display()

}

}

Illustration of an anonymous class extending an abstract class

abstract class Person

{// abstract class declared

abstract void display();// abstract method declared

}

class AnonymousDemo2

{

public static void main (String args[])

{

Person person = new Person() {// creating an object

void display()

{

System.out.println("My name is John.");

}

};// anonymous class closes

person.display();

} //The main class closes

}// class AnonymousDemo2 closes

**Passing Arguments by Value and by Reference**

In normal cases, when a method is defined with local variables, the values are passed by value, that is, the copies of values of variables are passed to the method that may manipulate the copies without affecting the values of variables. If the method is called again, the same copies of values are again passed. Therefore, the result of the method does not change; the variable values also do not change. This is called passing arguments by values.

class PassValue1

{

int func( int m, int n)// method definition

{

m += 4;

n -= 2;

return m+n;

}

}

class FuncExe// class containing the main method

{

public static void main (String args [])

{

int i = 2, j = 8;

PassValue1 Object1 = new PassValue1();

PassValue1 Object2 = new PassValue1();

System.out.println("Func output = " + Object1.func(i,j));

System.out.println("i = " + i + ", j = " + j);

System.out.println("Func output = " + Object2.func(i,j));

System.out.println ("i = " + i + ", j = " + j);

}

}

**By Reference**

if the method is defined with values of variables of objects of class or classes and if the method changes these values, the changed values are retained by objects and if the method is called again, these changed values are passed to the method, which will again change these values. Thus, every time the method is called, the object values as well as the return value of function get changed.

class Ref

{

int n;

Ref(int x)// constructor method of class Ref

{n = x;}

int func(Ref Obt)

{

Obt.n += 4; //Manipulation through object Obt

return Obt.n;

}

}

class PassByRef {// class containing main method

public static void main (String args [])

{

Ref refObj1 = new Ref(10);// new object of class Ref

System.out.println("Value returned by function = " + refObj1.func(refObj1));

System.out.println("Value returned by function = " + refObj1.func(refObj1));

Ref refObj2 = new Ref(20);

System.out.println("Value returned by function = " + refObj2.func(refObj2));

System.out.println("Value returned by function = " + refObj2.func(refObj2));

}

}