Topic Covered:
The Access Specifier, Static Class Members, Passing array to method and Variable Arguments (Varargs).

ACCESS SPECIFIER

For each class and for each member of a class, we need to provide an access modifier that specifies where the class or member can be used.

Access Modifier	Class or member can be referenced by ….
Public	Methods of the same class as well as methods of the other class
Private	Methods of the same class only
Protected	Methods in the same class, as well as methods in sub-classes and methods in same package
No modifier	Methods in the same package only

Typically access modifier for a class will be public and if class defined with public modifier than it must be stored with same file name.

Access Modifier	Within Class	Within Package	Same Package by subclasses	Outside Package by subclasses	Global
Public	Yes	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	Yes	No
Default	Yes	Yes	Yes	No	No
Private	Yes	No	No	No	No

PROGRAM 1: This program demonstrates the difference between public and private.

```
class Test {
int a; // default access
public int b; // public access
private int c; // private access
// methods to access c
void setc(int i) { // set c's value
c = i;
}
int getc() { // get c's value
return c;
}
}


class AccessTest {
public static void main(String args[]) {
Test ob = new Test();
// These are OK, a and b may be accessed directly
ob.a = 10;
ob.b = 20;
ob.c = 100; // Error!
// You must access c through its methods
ob.setc(100); // OK
System.out.println("a, b, and c: " + ob.a + " " + ob.b + " " + ob.getc());
}
}
```

PROGRAM 2: Demonstrate an Auto class for creating cars

```
public class Auto
{
 // instance variables
 private String model; //model of auto
 private int milesDriven; //number of miles driven
 private double gallonsOfGas; //number of gallons of gas

 // Default constructor: initializes model to "unknown";
 // milesDriven is auto-initialized to 0 and gallonsOfGas to 0.0

 public Auto( )
 {
 model = "unknown";
 }
```

```
// Overloaded constructor: allows client to set beginning values for
 // model, milesDriven, and gallonsOfGas.

 public Auto( String startModel,
 int startMilesDriven,
 double startGallonsOfGas )
 {
 model = startModel;

 // validate startMiles parameter
 if ( startMilesDriven >= 0 )
 milesDriven = startMilesDriven;
 else
 {
 System.err.println( "Miles driven is negative." );
 System.err.println( "Value set to 0." );
 }

 // validate startGallonsOfGas parameter
 if ( startGallonsOfGas >= 0.0 )
 gallonsOfGas = startGallonsOfGas;
 else
 {
 System.err.println( "Gallons of gas is negative" );
 System.err.println( "Value set to 0.0." );
 }
 }

 // Accessor method: returns current value of model
 public String getModel( )
 {   return model;   }

 // Accessor method: returns current value of milesDriven
 public int getMilesDriven( )
 {   return milesDriven;   }

 // Accessor method: returns current value of gallonsOfGas
 public double getGallonsOfGas( )
 {   return gallonsOfGas;  }
}
```

```
public class AutoClient
{
 public static void main( String [] args )
 {
 Auto sedan = new Auto( );
 String sedanModel = sedan.getModel( );
 int sedanMiles = sedan.getMilesDriven( );
 double sedanGallons = sedan.getGallonsOfGas( );

 System.out.println( "sedan: model is " + sedanModel
 + "\n miles driven is " + sedanMiles
 + "\n gallons of gas is " + sedanGallons );

 Auto suv = new Auto( "Trailblazer", 7000, 437.5 );
 String suvModel = suv.getModel( );
 int suvMiles = suv.getMilesDriven( );
 double suvGallons = suv.getGallonsOfGas( );
 System.out.println( "suv: model is " + suvModel
 + "\n miles driven is " + suvMiles
 + "\n gallons of gas is " + suvGallons );
 }
}
```

STATIC CLASS MEMBERS

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.

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. (The keyword super relates to inheritance and is described in the next chapter.)

If you need to do computation in order to initialize your static variables, you can declare astatic block that gets executed exactly once, when the class is first loaded.

ACCESS RESTRICTION FOR STATIC AND NON-STATIC METHODS

	Static method	Non-static method
Access instance variables	No	Yes
Access static class variables	Yes	Yes
Call static class methods	Yes	Yes
Call non-static instance methods	No	Yes
Use the reference this	No	Yes

PROGRAM 3: static variable

```
class Counter{
static int count=0;//will get memory only once and retain its value

Counter(){
count++;//incrementing the value of static variable
System.out.println(count);
}
}

class Test{
public static void main(String args[]){
//creating objects
Counter c1=new Counter();
Counter c2=new Counter();
Counter c3=new Counter();
}
}
```

PROGRAM 4: static variable and static method

```
class Student{
     int rollno;
     String name;
     static String college = "ITS";

     //static method to change the value of static variable
     static void change(){
     college = "BBDIT";
     }

     //constructor to initialize the variable
     Student(int r, String n){
     rollno = r;
     name = n;
     }

     //method to display values
     void display(){
     System.out.println(rollno+" "+name+" "+college);
     }
  }

//Test class to create and display the values of object
class TestStaticMethod{
    public static void main(String args[]){

    Student s1 = new Student(111,"Karan");
    Student s2 = new Student(222,"Aryan");
    Student s3 = new Student(333,"Sonoo");

    s1.display();
    s2.display();
    s3.display();

    Student.change();//calling change method

    s1.display();
    s2.display();
    s3.display();
    }
}
```

PROGRAM 5: Demonstrate static variables, methods, and blocks.

```
class UseStatic {
static int a = 3;
static int b;
static void meth(int x) {
System.out.println("x = " + x);
System.out.println("a = " + a);
System.out.println("b = " + b);
}
static {
System.out.println("Static block initialized.");
b = a * 4;
}
public static void main(String args[]) {
meth(42);
}
}
```

THIS KEYWORD

When a method begins executing, the JVM sets the object reference, this, to refer to the object for which the method has been called.

PROGRAM 6: Using 'this' keyword to refer current class instance variables and to invoke current class constructor

```
class Test {
 int a;
 int b;

 //Default constructor
 Test()
 {
  this(10, 20);
  System.out.println("Inside  default constructor \n");
 }

 //Parameterized constructor
 Test(int a, int b)
 {
  this.a = a;
  this.b = b;
  System.out.println("Inside parameterized constructor");
 }

 public static void main(String[] args)
 {
  Test object = new Test();
 }
}
```

PROGRAM 7: Using 'this' keyword to return the current class instance

```
class Test
{
 int a;
 int b;

 Test()
 {
  a = 10;
  b = 20;
 }

 //Method that returns current class instance
 Test get() {
 return this;
 }

 //Displaying value of variables a and b
 void display()
 {
  System.out.println("a = " + a + "  b = " + b);
 }

 public static void main(String[] args)
 {
  Test object = new Test();
  object.get().display();
 }
}
```

PROGRAM 8: Using 'this' keyword to invoke current class method

```
class Test {
    void display()
    {
        // calling fuction show()
        this.show();

       System.out.println("Inside display function");
    }

    void show() {
        System.out.println("Inside show funcion");
    }

    public static void main(String args[]) {
        Test t1 = new Test();
        t1.display();
    }
}
```

PASSING ARRAY TO METHOD

You can pass arrays to a method just like normal variables. When we pass an array to a method as an argument, actually the address of the array in the memory is passed (reference). Therefore, any changes to this array in the method will affect the array.

PROGRAM 9: Demonstrate passing of array to method

```
class Test
{
public static void sum(int[] arr) // getting sum of array values
{
int sum = 0;
for (int i = 0; i < arr.length; i++)
sum+=arr[i];
System.out.println("sum of array values : " + sum);
}

public static void main(String args[])
{
int arr[] = {3, 1, 2, 5, 4};
sum(arr); // passing array to method m1
}
}
```

VARIABLE ARGUMENTS (VARARGS) IN JAVA

In JDK 5, Java has included a feature that simplifies the creation of methods that need to take a variable number of arguments. This feature is called varargs and it is short-form for variable-length arguments. A method that takes a variable number of arguments is a varargs method.

Prior to JDK 5, variable-length arguments could be handled two ways. One using overloaded method(one for each) and another put the arguments into an array, and then pass this array to the method. Both of them are potentially error-prone and require more code. The varargs feature offers a simpler, better option.

A variable-length argument is specified by three periods(…). For Example,

```
public static void fun(int ... a)
{
 // method body
}
```

This syntax tells the compiler that fun( ) can be called with zero or more arguments. As a result, here a is implicitly declared as an array of type int[].

A method can have variable length parameters with other parameters too, but one should ensure that there exists only one varargs parameter that should be written last in the parameter list of the method declaration.

```
int nums(int a, float b, double … c)
```

In this case, the first two arguments are matched with the first two parameters and the remaining arguments belong to c.

- Vararg Methods can also be overloaded but overloading may lead to ambiguity.
- Prior to JDK 5, variable length arguments could be handled into two ways : One was using overloading, other was using array argument.
- There can be only one variable argument in a method.
- Variable argument (varargs) must be the last argument.

PROGRAM 10: Demonstrate Variable Arguments

```
class Test1 {
// A method that takes variable number of integer arguments.
static void fun(int ...a) {
System.out.println("Number of arguments: " + a.length);
// using for each loop to display contents of a
for (int i: a)
System.out.print(i + " ");
System.out.println();
}

public static void main(String args[]) {
// Calling the varargs method with different number of parameters
fun(100); // one parameter
fun(1, 2, 3, 4); // four parameters
fun(); // no parameter
}
}
```

PROGRAM 11: Overloading Vararg Methods

```
class VarArgs3 {
static void vaTest(int ... v) {
System.out.print("vaTest(int ...): " + "Number of args: " + v.length +
" Contents: ");
for(int x : v)
System.out.print(x + " ");
System.out.println();
}
static void vaTest(boolean ... v) {
System.out.print("vaTest(boolean ...) " + "Number of args: " +
v.length + " Contents: ");
for(boolean x : v)
System.out.print(x + " ");
System.out.println();
}
static void vaTest(String msg, int ... v) {
System.out.print("vaTest(String, int ...): " + msg + v.length +
" Contents: ");
for(int x : v)
System.out.print(x + " ");
System.out.println();
} public static void main(String args[])
{
vaTest(1, 2, 3);
vaTest("Testing: ", 10, 20);
vaTest(true, false, false);
}
}
```