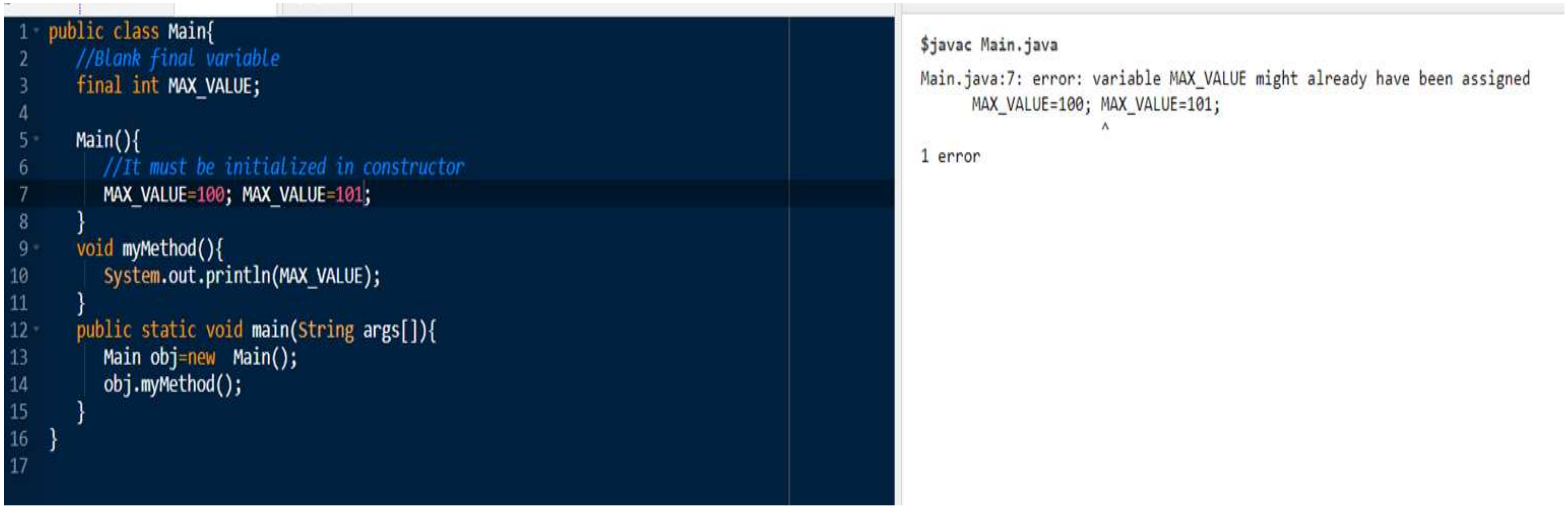
OOPs class
Vararg method, String, Inner & Outer Class

Final keyword



```java
public class Main{
   //Blank final variable
   final int MAX_VALUE;

   Main(){
      //It must be initialized in constructor
      MAX_VALUE=100; MAX_VALUE=101;
   }
   void myMethod(){
      System.out.println(MAX_VALUE);
   }
   public static void main(String args[]){
      Main obj=new  Main();
      obj.myMethod();
   }
}
```

```
$javac Main.java

Main.java:7: error: variable MAX_VALUE might already have been assigned
      MAX_VALUE=100; MAX_VALUE=101;
                     ^
1 error
```

Arrays Revisited

- Arrays are implemented as objects.
- Because of this, there is a special array attribute that you will want to take advantage of.
- Specifically, the size of an array—that is, the number of elements that an array can hold—is found in its **length instance variable.**
- **All arrays have this variable, and it will** always hold the size of the array. Here is a program that demonstrates this property:

```java
// This program demonstrates the length array member.
class Length {
  public static void main(String args[]) {
    int a1[] = new int[10];
    int a2[] = {3, 5, 7, 1, 8, 99, 44, -10};
    int a3[] = {4, 3, 2, 1};

    System.out.println("length of a1 is " + a1.length);
    System.out.println("length of a2 is " + a2.length);
    System.out.println("length of a3 is " + a3.length);
  }
}
```

o/p

```
length of a1 is 10
length of a2 is 8
length of a3 is 4
```

Note: It only reflects the number of elements that the array is designed to hold

Stack program

- The following version lets you create stacks of any size. The value of **stck.length is used to prevent the stack from overflowing.**

```
// Improved Stack class that uses the length array member.
class Stack {
  private int stck[];
  private int tos;

  // allocate and initialize stack
  Stack(int size) {
    stck = new int[size];
    tos = -1;
  }

  // Push an item onto the stack
  void push(int item) {
    if(tos==stck.length-1) // use length member
      System.out.println("Stack is full.");
    else
      stck[++tos] = item;
  }
// Pop an item from the stack
  int pop() {
    if(tos < 0) {
      System.out.println("Stack underflow.");
      return 0;
    }
    else
      return stck[tos--];
  }
}
class TestStack2 {
  public static void main(String args[]) {
    Stack mystack1 = new Stack(5);
    Stack mystack2 = new Stack(8);
    // push some numbers onto the stack
    for(int i=0; i<5; i++) mystack1.push(i);
    for(int i=0; i<8; i++) mystack2.push(i);
    // pop those numbers off the stack
    System.out.println("Stack in mystack1:");
    for(int i=0; i<5; i++)
      System.out.println(mystack1.pop());
    System.out.println("Stack in mystack2:");
    for(int i=0; i<8; i++)
      System.out.println(mystack2.pop());
  }
}
```

Notice that the program creates two stacks: one five elements deep and the other eight elements deep.
As you can see, the fact that arrays maintain their own length information makes it easy to create stacks of any size.

Nested and Inner Classes

- It is possible to define a class within another class; such classes are known as *nested classes*.
- The scope of a nested class is bounded by the scope of its enclosing class.
- Thus, if class B is defined within class A, then B does not exist independently of A.
- A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class.
- A nested class that is declared directly within its enclosing class scope is a member of its enclosing class.
- It is also possible to declare a nested class that is local to a block.

```
//Outer class
class OuterClass {
//Inner class as a nested class
class InnerClass {
....
}
}
```

- There are two types of nested classes: *static and non-static*.
- *A static nested class is one* that has the **static modifier applied.**
- **Because it is static, it must access the non-static members** of its enclosing class through an object.
- That is, it cannot refer to non-static members of its enclosing class directly. Because of this restriction, static nested classes are rarely used.

- The most important type of nested class is the *inner class*.
- *An inner class is a non-static* nested class.
- It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do.

```
// 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();
  }
}
```

o/p

```
display: outer_x = 100
```

Note: It is important to realize that an instance of **Inner can be created only in the context of** class **Outer.**
The Java compiler generates an error message otherwise.

- As explained, an inner class has access to all of the members of its enclosing class, but the reverse is not true.
- Members of the inner class are known only within the scope of the inner class and may not be used by the outer class.

```
// This program will not compile.
class Outer {
  int outer_x = 100;
  void test() {
    Inner inner = new Inner();
    inner.display();
  }
  // this is an inner class
  class Inner {
    int y = 10; // y is local to Inner

    void display() {
      System.out.println("display: outer_x = " + outer_x);
    }
  }
   void showy() {
    System.out.println(y); // error, y not known here!
  }
}
class InnerClassDemo {
  public static void main(String args[]) {
    Outer outer = new Outer();
    outer.test();
  }
}
```

- we have been focusing on inner classes declared as members within an outer class scope, it is possible to define inner classes within any block scope.
- For example, you can define a nested class within the block defined by a method or even within the body of a **for loop, as this next program shows:**

```
// Define an inner class within a for loop.
class Outer {
  int outer_x = 100;

  void test() {
    for(int i=0; i<10; i++) {
      class Inner {
        void display() {
          System.out.println("display: outer_x = " + outer_x);
        }
      }
      Inner inner = new Inner();
      inner.display();
    }
  }
}
class InnerClassDemo {
  public static void main(String args[]) {
    Outer outer = new Outer();
    outer.test();
  }
}
```

o/p

```
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
```

```
// Java program to demonstrate accessing a static nested class
// outer class
class OuterClass
{
        // static member
        static int outer_x = 10;
        // instance(non-static) member
        int outer_y = 20;
        // private member
        private static int outer_private = 30;
        // static nested class
        static class StaticNestedClass
        {
                void display()
                {
                        // can access static member of outer class
                        System.out.println("outer_x = " + outer_x);
                        // can access display private static member of outer class
                        System.out.println("outer_private = " + outer_private);
                        // The following statement will give compilation error
                        // as static nested class cannot directly access non-static members
                        // System.out.println("outer_y = " + outer_y);
                }
        }
}
```

```
// Driver class
public class StaticNestedClassDemo
{
	public static void main(String[] args)
	{
		// accessing a static nested class
		OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();

		nestedObject.display();

	}
}
```

Output:

```
outer_x = 10 outer_private = 30
```

String Class

- **String is probably the most commonly used class** in Java’s class library. The understandable reason for this is that strings are a very important part of programming.

- The first thing to understand about strings is that every string you create is actually an object of type **String.**

- **Even string constants are actually String objects. For example, in the** statement the string "This is a String, too" is a **String object.**

- `System.out.println("This is a String, too");`

- The second thing to understand about strings is that objects of type **String are immutable;**
- once a **String object is created, its contents cannot be altered. While this may seem like a** serious restriction, it is not, for two reasons:
- If you need to change a string, you can always create a new one that contains the modifications.
- Java defines peer classes of **String, called StringBuffer and StringBuilder, which** allow strings to be altered, so all of the normal string manipulations are still available in Java.

- Strings can be constructed in a variety of ways. The easiest is to use a statement like this:
- `String myString = "this is a test";`

- Once you have created a **String object, you can use it anywhere that a string is allowed.**
- For example, this statement displays **myString:**
- `System.out.println(myString);`

- Java defines one operator for **String objects: +. It is used to concatenate two strings. For** example, this statement
- `String myString = "I" + " like " + "Java.";`
- results in **myString containing "I like Java."**

```
// Demonstrating Strings.
class StringDemo {
  public static void main(String args[]) {
    String strOb1 = "First String";
    String strOb2 = "Second String";
    String strOb3 = strOb1 + " and " + strOb2;

    System.out.println(strOb1);
    System.out.println(strOb2);
    System.out.println(strOb3);
  }
}
```

o/p

```
•    First String
•    Second String
•    First String and Second String
```

The **String class contains several methods that you can use. Here are a few.**

You can test two strings for equality by using **equals( ).**

You can obtain the length of a string by calling the **length( ) method.**

You can obtain the character at a specified index within a string by calling **charAt( ).**

The general forms of these three methods are shown here:
boolean equals(*secondStr)*
int length( )
char charAt(*index)*

String Length

```
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 );
   }
}
```

o/p

```
String Length is : 17
```

Concatenating Strings

```
public class StringDemo
 {
 public static void main(String args[])
 {
    String string1 = "saw I was ";
     System.out.println("Dot " + string1 + "Tod");
}
}
```

o/p

```
Dot saw I was Tod
```

```
// Demonstrating some String methods.
class StringDemo2 {
  public static void main(String args[]) {
    String strOb1 = "First String";
    String strOb2 = "Second String";
    String strOb3 = strOb1;

    System.out.println("Length of strOb1: " +
                       strOb1.length());

    System.out.println("Char at index 3 in strOb1: " +
strOb1.charAt(3));

    if(strOb1.equals(strOb2))
      System.out.println("strOb1 == strOb2");
    else
      System.out.println("strOb1 != strOb2");

    if(strOb1.equals(strOb3))
      System.out.println("strOb1 == strOb3");
    else
      System.out.println("strOb1 != strOb3");
  }
}
```

o/p

```
Length of strOb1: 12
Char at index 3 in strOb1: s
strOb1 != strOb2
strOb1 == strOb3
```

More string methods

- **equalsIgnoreCase(String s)**

```
String x = "Exit";
 System.out.println( x.equalsIgnoreCase("EXIT")); // is "true"
System.out.println( x.equalsIgnoreCase("tixe")); // is "false"
```

- **replace(char old, char new**

```
String x = "oxoxoxox";
System.out.println( x.replace('x', 'X') ); // output is "oXoXoXoX"
```

- **public String substring(int begin)/ public String substring(int begin, int end)**

```
String x = "0123456789"; // the value of each char is the same as its index!
 System.out.println( x.substring(5) ); // output is "56789"
System.out.println( x.substring(5, 8)); // output is "567"
```

- **toLowerCase()**

```
String x = "A New Java Book";
System.out.println( x.toLowerCase() ); // output is "a new java book"
```

- **toUpperCase()**

```
String x = "A New Java Book";
System.out.println( x.toUpperCase() ); // output is"A NEW JAVA BOOK"
```

- **contains(“searchString”)**

```
String x = “Java is programming language”;
System.out.println(x.contains(“Amit”)); // This will print false
 System.out.println(x.contains(“Java”)); // This will print true
```

- **trim()**

```
String x = " hi ";
System.out.println( x + "x" ); // result is" hi x"
System.out.println(x.trim() + "x"); // result is "hix"
```

arrays of strings

```
// Demonstrate String arrays.
class StringDemo3 {
  public static void main(String args[]) {
    String str[] = { "one", "two", "three"
};

    for(int i=0; i<str.length; i++)
      System.out.println("str[" + i + "]: "
+
                                str[i]);
  }
}
```

```
// Demonstrate String arrays.
class StringDemo3 {
  public static void main(String args[]) {
    String str[] = { "one", "two", "three" };

    for(int i=0; i<str.length; i++)
      System.out.println("str[" + i + "]: " +
                          str[i]);
  }
}
```

Here is the output from this program:

```
str[0]: one
str[1]: two
str[2]: three
```

Command-Line Arguments

- To pass information into a program when you run it.
- This is accomplished by passing command-line arguments to main( ).
- A command-line argument is the information that directly follows the program's name on the command line when it is executed.
- They are stored as strings in a String array passed to the args parameter of main( ).
- The first command-line argument is stored at args[0], the second at args[1], and so on.

For example, the following program displays all of the command-line arguments that it is called with:

```
// Display all command-line arguments.
class CommandLine {
  public static void main(String args[]) {
    for(int i=0; i<args.length; i++)
      System.out.println("args[" + i + "]: " +
                          args[i]);
  }
}
```

```
java CommandLine this is a test 100 -1
```

When you do, you will see the following output:

```
args[0]: this
args[1]: is
args[2]: a
args[3]: test
args[4]: 100
args[5]: -1
```

Varargs: Variable-Length Arguments

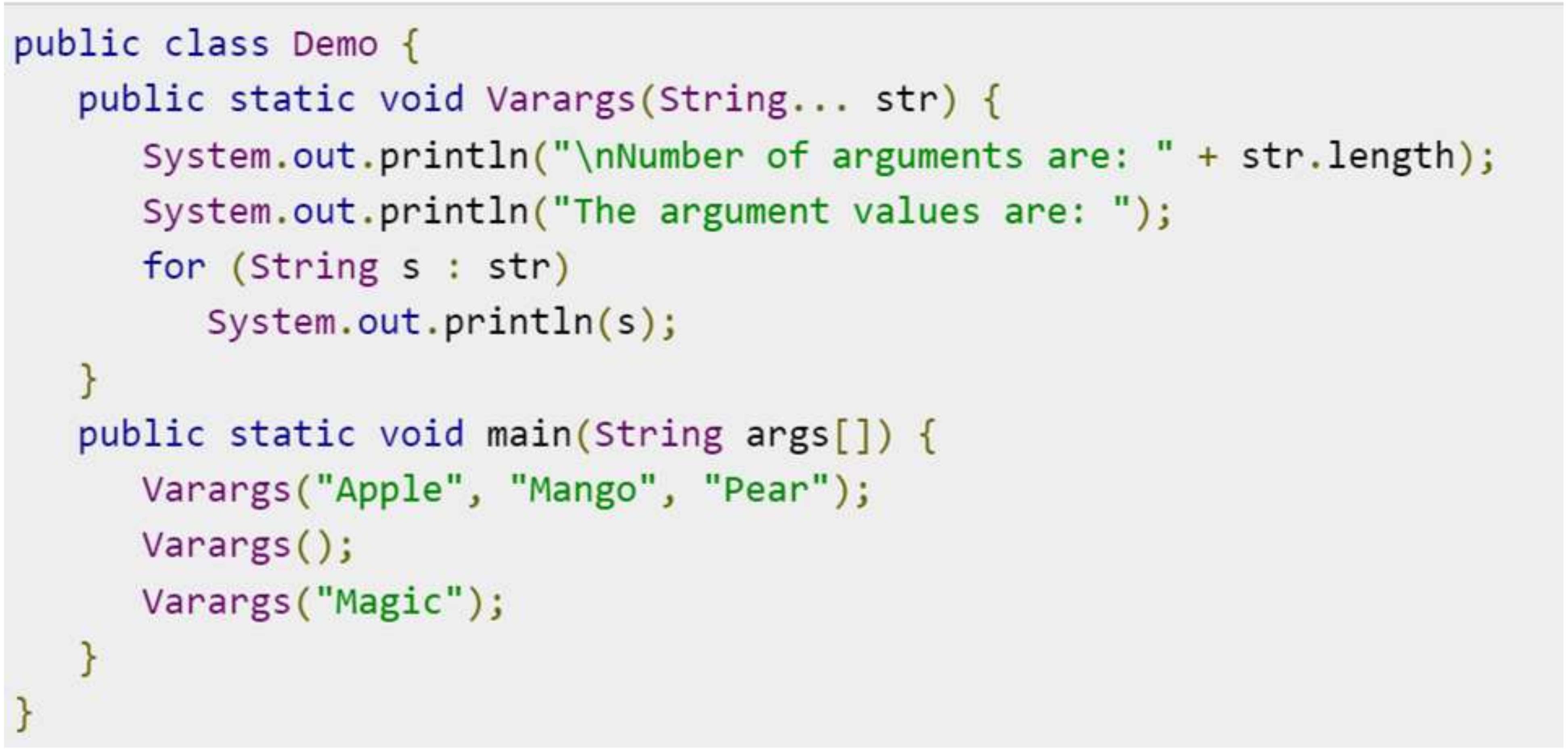
- The varrags allows the method to accept zero or muliple arguments.
- The varargs uses three dots after the data type. Syntax is as follows:

- `void vaTest(int ... v)`

```
class VarargsExample1{

 static void display(String... values){
  System.out.println("display method invoked ");
 }

 public static void main(String args[]){

 display();//zero argument
 display("my","name","is","varargs");//four argum
ents
 }
}
```

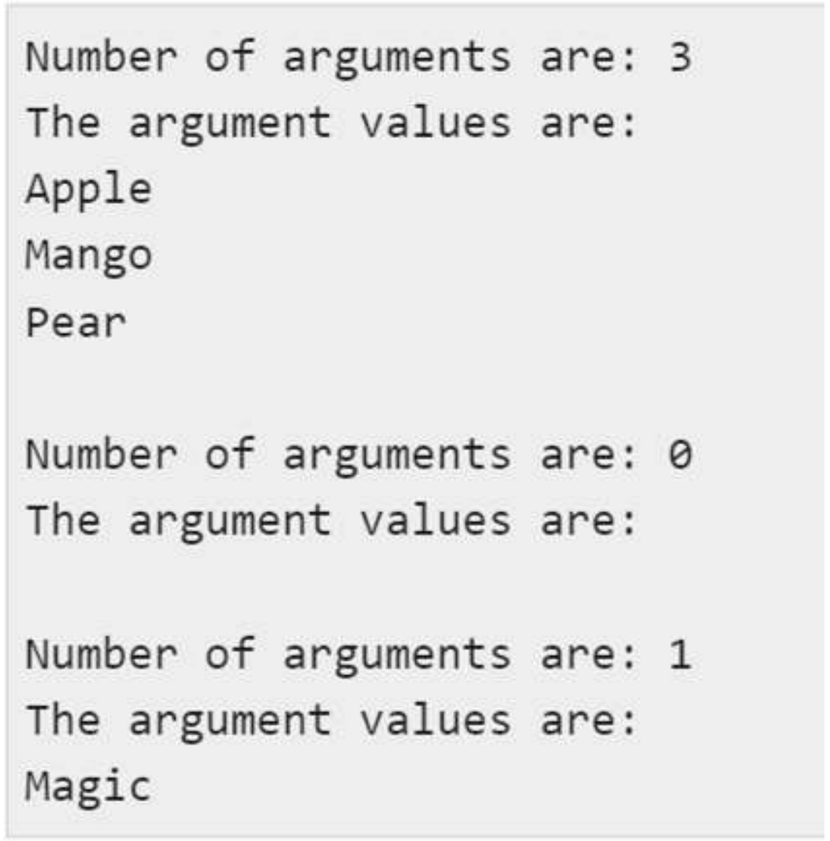
o/p

- display method invoked
- display method invoked



```java
public class Demo {
   public static void Varargs(String... str) {
      System.out.println("\nNumber of arguments are: " + str.length);
      System.out.println("The argument values are: ");
      for (String s : str)
         System.out.println(s);
   }
   public static void main(String args[]) {
      Varargs("Apple", "Mango", "Pear");
      Varargs();
      Varargs("Magic");
   }
}
```

o/p



```
Number of arguments are: 3
The argument values are:
Apple
Mango
Pear

Number of arguments are: 0
The argument values are:

Number of arguments are: 1
The argument values are:
Magic
```

```
class Main {
  public int sumNumber(int ... args){
      System.out.println("argument length: " + args.length);
      int sum = 0;
      for(int x: args){
          sum += x;
      }
      return sum;
   }
   public static void main( String[] args ) {
      Main ex = new Main();
      int sum2 = ex.sumNumber(2, 4);
      System.out.println("sum2 = " + sum2);
      int sum3 = ex.sumNumber(1, 3, 5);
      System.out.println("sum3 = " + sum3);
      int sum4 = ex.sumNumber(1, 3, 5, 7);
      System.out.println("sum4 = " + sum4);
   }}
```

o/p

- argument length: 2
- sum2 = 6
- argument length: 3
- sum3 = 9
- argument length: 4
- sum4 = 16

```
int doIt(int a, int b, double c, int ... vals) {
```

In this case, the first three arguments used in a call to **doIt( ) are matched to the first three** parameters. Then, any remaining arguments are assumed to belong to **vals.**

- Remember, the varargs parameter must be last. For example, the following declaration is incorrect:
- `int doIt(int a, int b, double c, int ... vals, boolean stopFlag) { // Error!`
- Here, there is an attempt to declare a regular parameter after the varargs parameter, which is illegal.

- There is one more restriction to be aware of: there must be only one varargs parameter.
- For example, this declaration is also invalid:
- `int doIt(int a, int b, double c, int ... vals, double ... morevals) { // Error!`
- The attempt to declare the second varargs parameter is illegal.

Overloading Vararg Methods

- You can overload a method that takes a variable-length argument. For example, the following program overloads **vaTest( ) three times:**

```
// Varargs and overloading.
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);
  }
}
```

o/p

The output produced by this program is shown here:

```
vaTest(int ...): Number of args: 3 Contents: 1 2 3
vaTest(String, int ...): Testing: 2 Contents: 10 20
vaTest(boolean ...) Number of args: 3 Contents: true false false
```

Note:

First, the types of its vararg parameter can differ. This is the case for **vaTest(int ...) and vaTest(boolean ...).** Remember, the **... causes the parameter to be treated as an array of the specified type.** Therefore, just as you can overload methods by using different types of array parameters, you can overload vararg methods by using different types of varargs. In this case, Java uses the type difference to determine which overloaded method to call.

The second way to overload a varargs method is to add one or more normal parameters. This is what was done with **vaTest(String, int ...). In this case, Java uses both the number of** arguments and the type of the arguments to determine which method to call.