

Java

More Details

Array

Arrays

- A group of variables containing values that all have the same type
- Arrays are fixed-length entities
- In Java, arrays are objects, so they are considered reference types
- But the elements of an array can be either primitive types or reference types

Arrays

- We access the element of an array using the following syntax
 - name[index]
 - “index” must be a nonnegative integer
 - “index” can be int/byte/short/char but not long
- In Java, every array knows its own length
- The length information is maintained in a public final int member variable called **length**

Declaring and Creating Arrays

- `int c[] = new int [12]`
 - Here, “c” is a reference to an integer array
 - “c” is now pointing to an array object holding 12 integers
 - Like other objects arrays are created using “new” and are created in the heap
 - “int c[]” represents both the data type and the variable name. Placing number here is a syntax error
 - **`int c[12]; // compiler error`**

Declaring and Creating Arrays

- `int[] c = new int [12]`
 - Here, the data type is more evident i.e. “`int[]`”
 - But does the same work as
 - `int c[] = new int [12]`
- Is there any difference between the above two approaches?

Declaring and Creating Arrays

- `int c[], x`
 - Here, 'c' is a reference to an integer array
 - 'x' is just a normal integer variable
- `int[] c, x;`
 - Here, 'c' is a reference to an integer array (same as before)
 - But, now 'x' is also a reference to an integer array

Arrays

```
ArrayDemo.java ×  
1 ▶ public class ArrayDemo {  
2 ▶   public static void main(String[] args) {  
3     int [] a = new int[10];  
4     for (int i = 0; i < a.length; i++) {  
5       a[i] = i;  
6     }  
7     for (int i = 0; i < a.length; i++) {  
8       System.out.println(a[i]);  
9     }  
10  }  
11 }  
12
```


Using an Array_INITIALIZER

- We can also use an array initializer to create an array
 - `int n[] = {10, 20, 30, 40, 50}`
- The length of the above array is 5
- `n[0]` is initialized to 10, `n[1]` is initialized to 20, and so on
- The compiler automatically performs a “new” operation taking the count information from the list and initializes the elements properly

Arrays of Primitive Types

- When created by “new”, all the elements are initialized with default values
 - byte, short, char, int, long, float and double are initialized to zero
 - boolean is initialized to false
- This happens for both member arrays and local arrays

Arrays of Reference Types

- `String [] str = new String[3]`
 - Only 3 String references are created
 - Those references are initialized to **null** by default
 - Need to explicitly create and assign actual String objects in the above three positions.
 - `str[0] = new String("Hello");`
 - `str[1] = "World";`
 - `str[2] = "I" + " Like" + " Java";`

Passing Arrays to Methods

```
void modifyArray(double d[ ]) {...}  
double [] temperature = new double[24];  
modifyArray(temperature);
```

- Changes made to the elements of 'd' inside "modifyArray" is visible and reflected in the "temperature" array
- But inside "modifyArray" if we create a new array and assign it to 'd' then 'd' will point to the newly created array and changing its elements will have no effect on "temperature"

Passing Arrays to Methods

- Changing the elements is visible, but changing the array reference itself is not visible

```
void modifyArray(double d[ ]) {  
    d[0] = 1.1; // visible to the caller  
}
```

```
void modifyArray(double d[ ]) {  
    d = new double [10];  
    d[0] = 1.1; // not visible to the caller  
}
```

Multidimensional Arrays

- Can be termed as array of arrays.
- `int b[][] = new int[3][4];`
 - Length of first dimension = 3
 - `b.length` equals 3
 - Length of second dimension = 4
 - `b[0].length` equals 4
- `int[][] b = new int[3][4];`
 - Here, the data type is more evident i.e. “`int[][]`”

Multidimensional Arrays

- `int b[][] = { { 1, 2, 3 }, { 4, 5, 6 } };`
 - `b.length` equals 2
 - `b[0].length` and `b[1].length` equals 3
- All these examples represent rectangular two dimensional arrays where every row has same number of columns
- Java also supports jagged array where rows can have different number of columns

Multidimensional Arrays

Example – 1

```
int b[ ][ ];  
b = new int[2][ ];  
b[0] = new int[2];  
b[1] = new int[3];  
b[0][2] = 7; //will throw an exception
```

Example – 2

```
int b[ ][ ] = { { 1, 2 }, { 3, 4, 5 } };  
b[0][2] = 8; //will throw an exception
```

In both cases

b.length equals 2
b[0].length equals 2
b[1].length equals 3

Array 'b'

	Col 0	Col 1	Col 2
Row 0			
Row 1			

b[0][2] does not exist

Command Line Arguments

Using Command-Line Arguments

- `java MyClass arg1 arg2 ... argN`
 - words after the class name are treated as command-line arguments by Java
 - Java creates a separate `String` object containing each command-line argument, places them in a `String` array and supplies that array to `main`
 - That's why we have to have a `String` array parameter (`String args[]`) in `main`
 - We do not need a “`argc`” type parameter (for parameter counting) as we can easily use “`args.length`” to determine the number of parameters supplied.

Using Command-Line Arguments

```
CommandLineTest.java x
1  ▶ public class CommandLineTest {
2  ▶     public static void main(String[] args) {
3      System.out.println( args.length );
4
5      for( int i = 0; i < args.length; i++)
6      {
7          System.out.println( args[i] );
8      }
9  }
10 }
11 |
```

java CommandLineTest Hello 2 You

**3
Hello
2
You**

For-Each

For-Each version of the for loop

```
ForEachTest.java x
1 public class ForEachTest {
2     public static void main(String[] args) {
3         int numbers [] = {1,2,3,4,5};
4         for(int x : numbers)
5         {
6             System.out.print(x + " ");
7             x = x * 10; // no effect on numbers
8         }
9         System.out.println();
10
11        int numbers2 [][] = { {1,2,3}, {4,5,6}, {7,8,9} };
12        for(int []x:numbers2)
13        {
14            for(int y:x)
15            {
16                System.out.print(y + " ");
17            }
18            System.out.println("");
19        }
20    }
21 }
```

Scanner

Scanner

- It is one of the utility class located in the java.util package
- Using Scanner class, we can take inputs from the keyboard
- Provides methods for scanning
 - int
 - float
 - double
 - line etc.

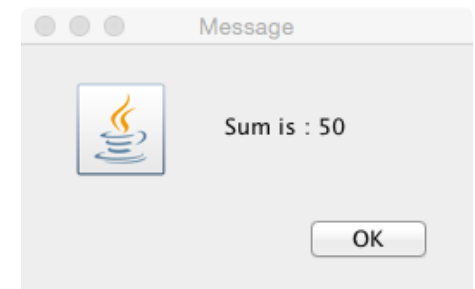
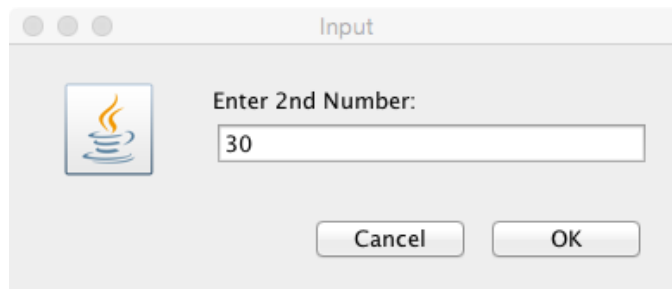
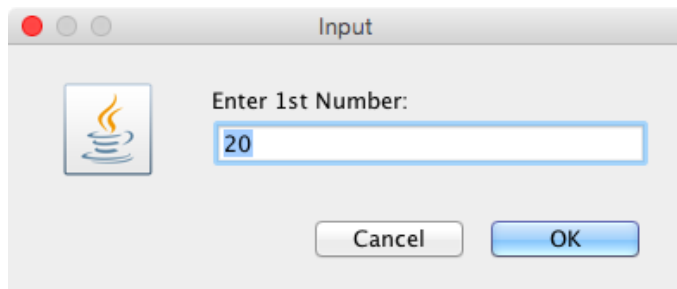
Scanner

```
3 import java.util.Scanner;
4
5 public class ScannerTest {
6     public static void main(String[] args) {
7         Scanner scn=new Scanner(System.in);
8         while(scn.hasNextLine())
9         {
10             System.out.println(scn.nextLine());
11         }
12     }
13 }
```

```
3 import java.util.Scanner;
4
5 public class ScannerTest {
6     public static void main(String[] args) {
7         Scanner scn=new Scanner(System.in);
8         while(scn.hasNextInt())
9         {
10             System.out.println(scn.nextInt());
11         }
12     }
13 }
```


JOptionPane

```
3 import javax.swing.JOptionPane;
4
5 public class JOptionPaneTest {
6     public static void main(String[] args) {
7         String s1 = JOptionPane.showInputDialog(null, "Enter 1st Number:");
8         String s2 = JOptionPane.showInputDialog(null, "Enter 2nd Number:");
9         int num1 = Integer.parseInt(s1);
10        int num2 = Integer.parseInt(s2);
11        JOptionPane.showMessageDialog(null, "Sum is : " + (num1+num2));
12    }
13 }
```



Static

Static Variables

- When a member (both methods and variables) is declared static, it can be accessed before any objects of its class are created, and without reference to any object
- Static variable
 - Instance variables declared as static are like global variables
 - When objects of its class are declared, no copy of a static variable is made

Static Methods & Blocks

- Static method
 - They can only call other static methods
 - They must only access static data
 - They cannot refer to ***this*** or ***super*** in any way
- Static block
 - Initialize static variables.
 - Get executed exactly once, when the class is first loaded

Static

```
3 public class StaticTest {
4     static int a = 3, b;
5     int c;
6
7     static void f1(int x) {
8         System.out.println("x = " + x);
9         System.out.println("a = " + a);
10        System.out.println("b = " + b);
11        // System.out.println("c = " + c); // Error
12    }
13    int f2() {
14        return a*b;
15    }
16    static {
17        b = a*4;
18        // c = b; // Error
19    }
20    public static void main(String[] args) {
21        f1(42); // StaticTest.f1(84);
22        System.out.println("b = " + b);
23        //System.out.println("Area = " + f2()); // Error
24    }
25 }
```

Final

- Declare a final variable, prevents its contents from being modified
- final variable must initialize when it is declared
- It is common coding convention to choose all uppercase identifiers for final variables

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;

Unsigned right shift operator

- The >> operator automatically fills the high-order bit with its previous contents each time a shift occurs
- This preserves the sign of the value
- But if you want to shift something that doesn't represent a numeric value, you may not want the sign extension
- Java's >>> shifts zeros into the high-order bit

int a = -1; a = a >>> 24;

11111111	11111111	11111111	11111111	[-1]
00000000	00000000	00000000	11111111	[255]

Nested and Inner Classes

Nested Classes

- It is possible to define a class within another classes, such classes are known as nested classes
- The scope of nested class is bounded by the scope of its enclosing class. That means if class B is defined within class A, then B doesn't exists without A
- The nested class has access to the members (including private!) of the class in which it is nested
- The enclosing class doesn't have access to the members of the nested class

Static Nested Classes

- Two types of nested classes.
 - Static
 - Non-Static
- A static nested class is one which has the static modifier applied. Because it is static, it must access the members of its enclosing class through an object
- That is, it cannot refer to members of its enclosing class directly. Because of this restriction, static nested classes are seldom used

Static Nested Classes

```
1  class OuterStaticInner {
2      private int outer_x = 100;
3
4      void test() {
5          Inner inner = new Inner();
6          inner.display(outer: this);
7      }
8      // this is a static nested class
9      static class Inner {
10         void display(OuterStaticInner outer) {
11             System.out.println(outer.outer_x);
12         }
13     }
14 }
15
16 public class StaticNestedClassDemo {
17     public static void main(String[] args) {
18         OuterStaticInner outer = new OuterStaticInner();
19         outer.test();
20         OuterStaticInner.Inner x = new OuterStaticInner.Inner();
21         x.display(outer);
22     }
23 }
```

Inner Classes

- 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
- Thus, an inner class is fully within the scope of its enclosing class

Inner Classes

```
1  class Outer1
2  {
3      private int outer_x = 100;
4
5      void test() {
6          Inner inner = new Inner();
7          inner.display();
8      }
9      // this is an inner class
10     class Inner {
11         void display() {
12             System.out.println(outer_x);
13         }
14     }
15 }
16
17 public class InnerClassDemo1 {
18     public static void main(String[] args) {
19         Outer1 outer = new Outer1();
20         outer.test();
21         Outer1.Inner innerObj = outer.new Inner();
22         innerObj.display();
23     }
24 }
```

Inner Classes

```
1  class Outer2
2  {
3      int outer_x = 100;
4
5      void test() {
6          Inner inner = new Inner();
7          inner.display();
8      }
9
10     class Inner {
11         int y = 10; // y is local to Inner
12         void display() { System.out.println(outer_x); }
13     }
14
15     void showy() {
16         //System.out.println(y); // error, y not known here!
17     }
18 }
19
20
21
22 public class InnerClassDemo2 {
23     public static void main(String[] args) {
24         Outer2 outer = new Outer2();
25         outer.test();
26     }
27 }
```

Variable Arguments

```
1  ► public class VarArgsTest {
2      static void vaTest(int ... v){
3          for(int x: v) {
4              System.out.print(x + " ");
5          }
6          System.out.println();
7      }
8      static void vaTest(boolean ... v){
9          for(boolean x: v) {
10             System.out.print(x + " ");
11         }
12         System.out.println();
13     }
14     static void vaTest(String msg, int ... v){
15         System.out.print(msg + " ");
16         for(int x: v) {
17             System.out.print(x + " ");
18         }
19         System.out.println();
20     }
21     ► public static void main(String[] args) {
22         vaTest(msg: "Testing", ...v: 10, 20);
23         vaTest(...v: true, false, false);
24         vaTest(...v: 1, 2, 3);
25     }
26 }
```

Variable Arguments Ambiguity

```
1  ▶ public class VarArgsTest {
2      static void vaTest(int ... v){
3          for(int x: v) {
4              System.out.print(x + " ");
5          }
6          System.out.println();
7      }
8      static void vaTest(boolean ... v){
9          for(boolean x: v) {
10             System.out.print(x + " ");
11         }
12         System.out.println();
13     }
14     static void vaTest(int n, int ... v){
15         for(int x: v) {
16             System.out.println(x + " ");
17         }
18     }
19     ▶ public static void main(String[] args) {
20         vaTest(); // ambiguity type 1 because of int and boolean but works with int and double
21         vaTest(1, 2, 3); // ambiguity type 2 with vaTest(int n, int ... v) and vaTest(int ... v)
22     }
23 }
24
```


Local Variable Type Inference

- Recently added to the Java language (Java 10)
 - all variables must be declared prior to their use
 - a variable can be initialized with a value when it is declared
 - when a variable is initialized, the type of the initializer must be the same as the declared type of the variable
- In principle, it would not be necessary to specify an explicit type for an initialized variable
 - it could be inferred by the type of its initializer

Local Variable Type Inference

- Compiler infer the type of a local variable based on the type of its initializer without explicit specification
- Advantages:
 - Streamline code by eliminating the need to redundantly specify a variable's type when it can be inferred
 - Simplify declarations when the type name is quite lengthy, such as can be the case with some class names
 - Helpful when a type is difficult to determine
 - Its inclusion helps keep Java up-to-date with evolving trends in language design

Local Variable Type Inference

- The context-sensitive identifier `var` was added to Java as a reserved type name
- To use local variable type inference, the variable must be declared with `var` as the type name and it must include an initializer
 - **`double avg = 10.0; // type is explicitly specified`**
 - **`var avg = 10.0; // type is inferred as double because initializer (10.0) is of type double`**
- `var` can still be used as user-defined identifier
 - **`int var = 1; // valid`**

Local Variable Type Inference

- `var` cannot be used as the name of a class
- `var` can be used to declare an array type, but cannot be used with an array initializer
 - **`var myArray = new int[10]; // valid`**
 - **`var myArray = { 1, 2, 3 }; // invalid`**
- `var` is not allowed as an element type of an array
 - **`var[] myArray = new int[10]; // invalid`**
 - **`var myArray[] = new int[10]; // invalid`**

Local Variable Type Inference

- var can be used to declare a variable only when that variable is initialized
 - **var counter; // invalid**
- var cannot be used to declare a variable with null as the initializer
- var can be used only to declare local variables, it cannot be used when declaring instance variables, parameters, or return types
- var can be used in a for/for-each loop when declaring and initializing the loop control/iteration variable

Local Variable Type Inference

- Local variable type inference can also be used with reference types
 - **var str = “This is a string”;**
 - Type inference is primarily used with reference types
- Local variable type inference is especially effective in shortening declarations that involve long class names
- Local variable type inference can also be used with user-defined classes
 - **var mc = new MyClass();**