14. Array Basics

Java

Fall 2009

Instructor: Dr. Masoud Yaghini

Outline

- Introduction
- Declaring an Array
- Creating Arrays
- Accessing an Array
- Simple Processing on Arrays
- Copying Arrays
- References

Introduction

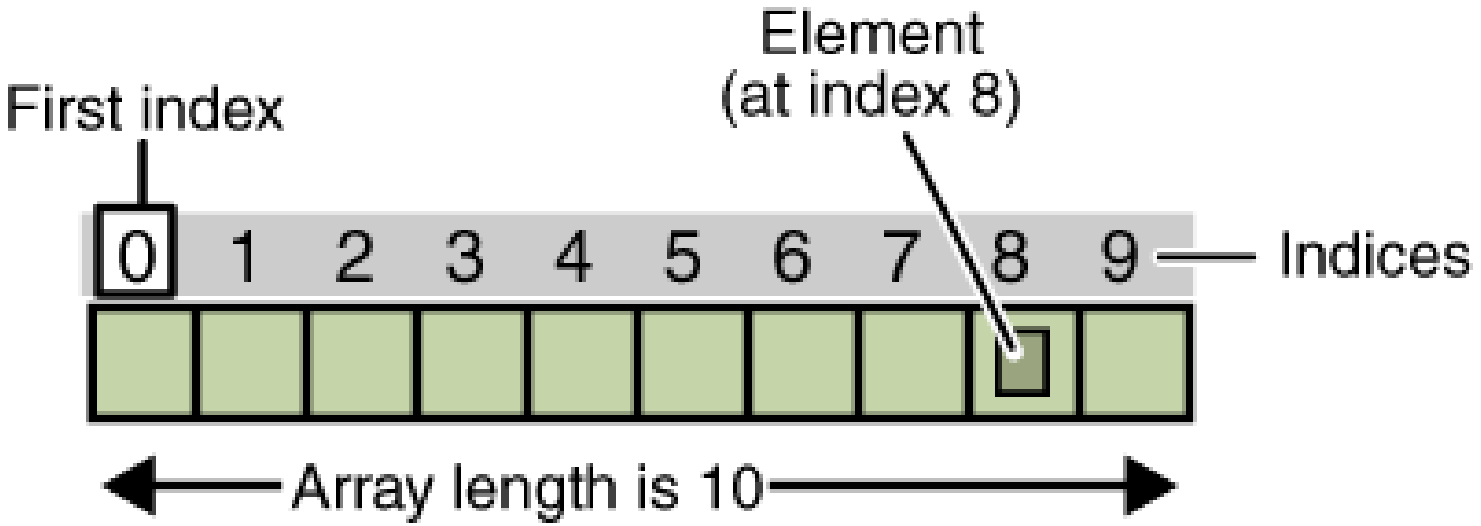
A problem with simple variables

- One variable holds one value
  - The value may change over time, but at any given time, a variable holds a single value
- If you want to keep track of many values, you need many variables
- All of these variables need to have names
- What if you need to keep track of hundreds or thousands of values?

Arrays

- An array lets you associate one name with a fixed number of values
- All values must have the same type
- Each item in an array is called an element
- The values are distinguished by a numerical index between 0 and array size minus 1
- The length of an array is established when the array is created.

An array of ten elements



An Example

- An Example:
  - ArrayDemo.java
- Program output:

```
Element at index 0: 100
Element at index 1: 200
Element at index 2: 300
Element at index 3: 400
Element at index 4: 500
Element at index 5: 600
Element at index 6: 700
Element at index 7: 800
Element at index 8: 900
Element at index 9: 1000
```

Declaring an Array

Declaring an Array

- Declaring an array:

  datatype[ ] arrayRefVar;

- Example:

  double[ ] myList;

- An array declaration has two components:
  - the array's type, and
  - the array's name

Declaring an Array

- An array's type is written as type[ ], where:
  - type is the data type of the contained elements;
  - the square brackets are special symbols indicating that this variable holds an array.
- The **size of the array is not** part of its type (which is why the brackets are empty).
- Unlike declarations for primitive data type variables, the declaration of an array variable **does not allocate** any space in memory for the array.

Declaring an Array

- An array's name can be anything you want, provided that it follows the rules and conventions as variables.
- The declaration does not actually create an array, it simply tells the compiler that this variable will hold an array of the specified type.

Declaring an Array

- Similarly, you can declare arrays of other types:

```
byte[ ] anArrayOfBytes;
short[ ] anArrayOfShorts;
long[ ] anArrayOfLongs;
float[ ] anArrayOfFloats;
double[ ] anArrayOfDoubles;
boolean[ ] anArrayOfBooleans;
char[ ] anArrayOfChars;
String[ ] anArrayOfStrings;
```

Declaring an Array

- You can also place the square brackets after the array's name:

  float anArrayOfFloats[ ];

  - Convention **discourages** this form
  - The brackets identify the array type and should appear with the **type designation**.

Declaring an Array

- You can declare more than one variable in the same declaration:

```
int a[ ], b, c[ ], d;  // notice position of brackets
```

  - a and c are int arrays
  - b and d are just ints

- Another syntax:

```
int [ ] a, b, c, d;     // notice position of brackets
```

  - a, b, c and d are int arrays

  – When the brackets come before the first variable, they apply to **all variables** in the list

- But, in Java, we typically declare each variable **separately**

Creating Arrays

Creating Arrays

- You cannot assign elements to an array unless it has already been created.
- After an array variable is declared, you can create an array by using the new operator with the following syntax:
  arrayRefVar = new dataType[arraySize];
- This statement does two things:
  - (1) it creates an array using new **dataType[arraySize];**
  - (2) it assigns the reference of the newly created array to the variable arrayRefVar.

Creating Arrays

- Example:

  anArray = new int[10]; // create an array of integers

- This statement allocates an array with enough memory for ten integer elements and assigns the array to the anArray variable
- If this statement were missing, the compiler would print an error like the following, and compilation would fail:
  - Variable anArray might not have been initialized.

Declaring and Creating in One Step

- **Declaring** an array variable, **creating** an array, and **assigning** the reference of the array to the variable can be combined in one statement, as shown below:

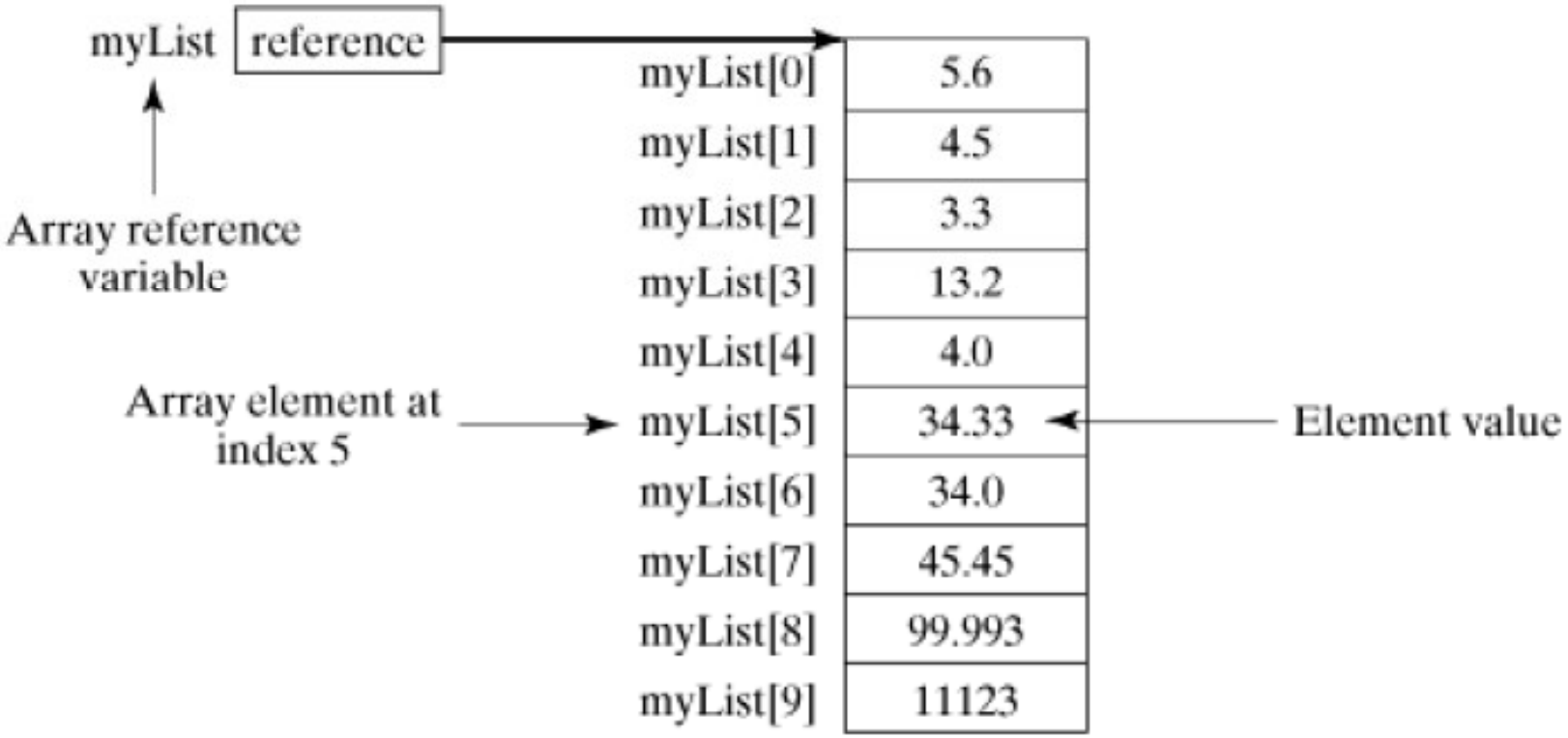
```
dataType[ ] arrayRefVar = new dataType[arraySize];
```

- Here is an example of such a statement:

```
double[ ] myList = new double[10];
```

Creating Arrays

```
double[] myList = new double[10];
```



Array Size

- Once an array is created, its size is fixed. It **cannot be changed**.
- you can use the built-in length property to determine the **size** of any array:

```
System.out.println(anArray.length);
```

- The code will print the array's size to standard output.

Default Values

- When an array is created, its elements are assigned the default value of:
  - 0 for the numeric primitive data types,
  - '\u0000' for char types, and
  - false for boolean types.

Accessing an Array

Accessing an Array

- The array elements are accessed through the index.
- The array indices are 0-based, i.e., it starts from 0 to arrayRefVar.length-1.
- Each element in the array is represented using the following syntax, known as an indexed variable:

```
arrayRefVar[index];
```

Accessing an Array

- Each array element is accessed by its numerical index.
- Example:

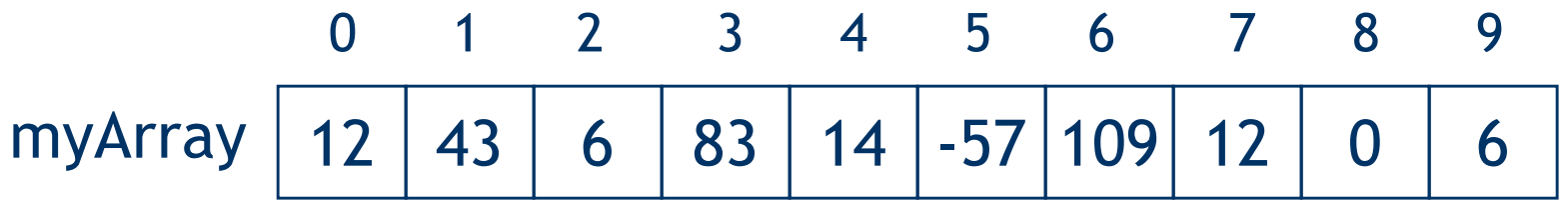
```
System.out.println("Element 1 at index 0: " + anArray[0]);
System.out.println("Element 2 at index 1: " + anArray[1]);
System.out.println("Element 3 at index 2: " + anArray[2]);
```

Initializing an Array

- The next few lines assign values to each element of the array:

```
anArray[0] = 100; // initialize first element
anArray[1] = 200; // initialize second element
anArray[2] = 300; // initialize third element
```

Accessing an Array



- Examples:
  - x = myArray[1]; // sets x to 43
  - myArray[4] = 99; // replaces 14 with 99
  - m = 5;
    y = myArray[m]; // sets y to -57
  - z = myArray[myArray[9]]; // sets z to 109

An Example

- An example:
  - TestDefaultValues.java

- Output:

```
11
1
3
6
10
```

Declare, Create and Initialize an Array

```
double[] myList = {1.9, 2.9, 3.4, 3.5};
```

- This shorthand notation is equivalent to the following statements:

```
double[] myList = new double[4];
myList[0] = 1.9;
myList[1] = 2.9;
myList[2] = 3.4;
myList[3] = 3.5;
```

- Here the length of the array is determined by the number of values provided between { and }.

Caution

- Using the shorthand notation, you have to **declare**, **create**, and **initialize** the array all in one statement.

```
double[] myList = {1.9, 2.9, 3.4, 3.5};
```

- Splitting it would cause a syntax error. For example, the following is wrong:

```
double[ ] myList;
myList = {1.9, 2.9, 3.4, 3.5};
```

Simple Processing on Arrays

Processing Arrays

- When processing array elements, you will often use a for loop.
- Here are the reasons why:
  - All of the elements in an array are of the same type.
  - They are evenly processed in the same fashion by repeatedly using a loop.
  - Since the size of the array is known, it is natural to use a **for** loop.

Initializing arrays

- The following loop initializes the array myList with random values between 0.0 and 99.0:

```
for (int i = 0; i < myList.length; i++)
{
    myList[i] = Math.random() * 100;
}
```

- Math.random() generates a random double value greater than or equal to 0.0 and less than 1.0 (0.0 <= Math.random() < 1.0).

Printing arrays

- To print an array:

```
for (int i = 0; i < myList.length; i++)
{
   System.out.print(myList[i] + " ");
}
```

- For an array of the char[ ] type, it can be printed using one print statement.
- For example, the following code displays Dallas:

```
char[ ] city = {'D', 'a', 'l', 'l', 'a', 's'};
System.out.println(city);
```

Finding the largest element

- Use a variable named max to store the largest element:

```
double max = myList[0];
for (int i = 1; i < myList.length; i++)
{
   if (myList[i] > max)
     max = myList[i];
}
```

Finding the smallest index of the largest element

- Often you need to locate the largest element in an array. If an array has more than one largest element, find the smallest index of such an element.

```
double max = myList[0];
int indexOfMax = 0;
for (int i = 1; i < myList.length; i++)
{
    if (myList[i] > max)
    {
      max = myList[i];
      indexOfMax = i;
    }
}
```

Enhanced for statement

- Enhanced for statement can be used to make your loops more compact and easy to read.
- The following program uses the enhanced for to loop through the array:
  - EnhancedForDemo.java
- Output:

```
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
Count is: 6
Count is: 7
Count is: 8
Count is: 9
Count is: 10
```

Copying Arrays

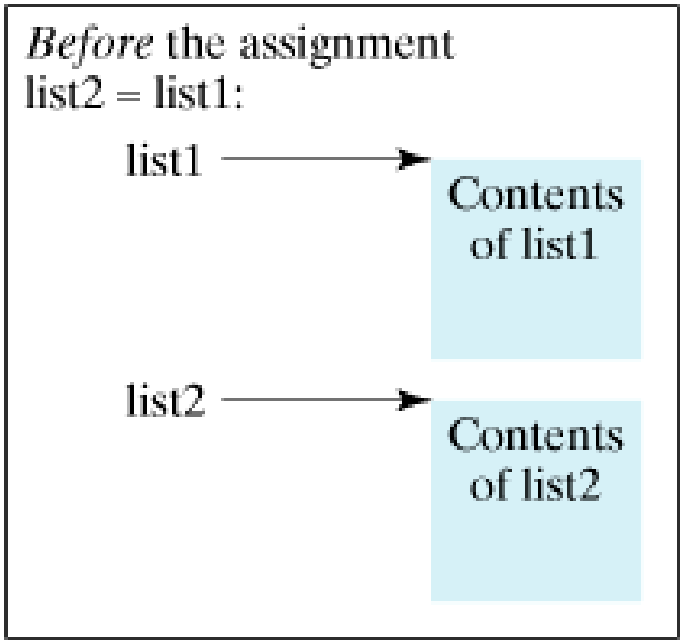
Copying Arrays

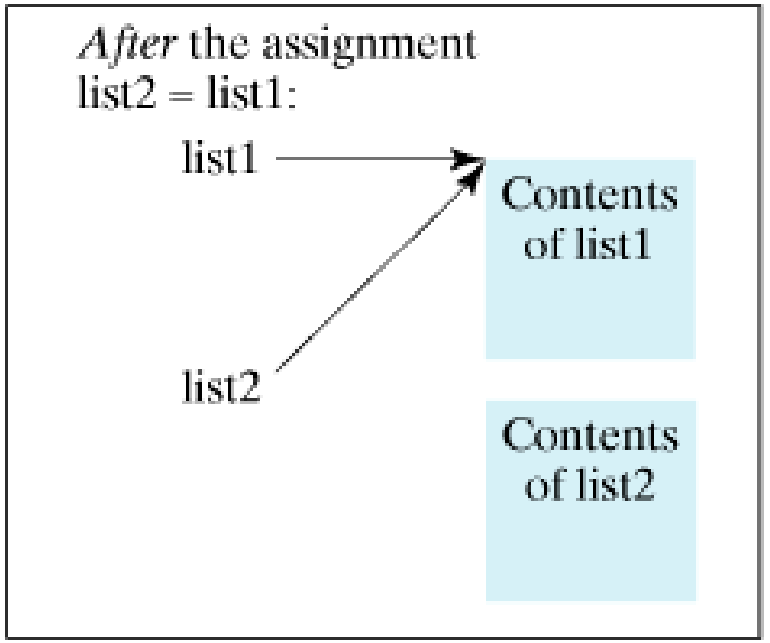
- Often you need to duplicate an array or a part of an array.
- In such cases you may attempt to use the assignment statement (=), as follows:

```
list2 = list1;
```

- This statement does not copy the contents of the array referenced
- It merely copies the reference value from list1 to list2.
- The array previously referenced by list2 is no longer referenced; it becomes garbage

Before and after assignment





Copying Arrays

- Using a loop:

```
int[] sourceArray = {2, 3, 1, 5, 10};
int[] targetArray = new int[sourceArray.length];
for (int i = 0; i < sourceArray.length; i++)
{
   targetArray[i] = sourceArray[i];
}
```

Copying Arrays

- The System class has an arraycopy method that you can use to efficiently copy data from one array into another:

arraycopy(sourceArray, srcPos, targetArray, tarPos, length);

- The arguments specify:
  - **`sourceArray`**: the array to copy from (source array)
  - **`srcPos`**: the array to copy to (destination array)
  - **`targetArray`**: the starting position in the source array
  - **`tarPos`**: the starting position in the destination array
  - **`length`**: the number of array elements to copy

An Example

- An example:
  - ArrayCopyDemo.java

- Output?

  caffein

ArrayCopyDemo Program

- The arraycopy method does not allocate memory space for the target array.
- The target array must have already been created with its memory space allocated.
- The arraycopy method **violates** the Java naming convention. By convention, this method should be named arrayCopy (i.e., with an uppercase C).

Arrays and Methods

Arrays and Methods

- Arrays and methods includes:
  - Passing Arrays to Methods
  - Returning an Array from a Method
  - Variable-Length Argument Lists

Passing Arrays to Methods

- Java uses **pass by value** to pass parameters to a method.
- There are important differences between passing the values of variables of primitive data types and passing arrays.

Passing Arrays to Methods

- **For a parameter of a primitive type value**
  - The actual value is passed.
  - Changing the value of the local parameter inside the method does not affect the value of the variable outside the method.
- **For a parameter of an array type**
  - The value of the parameter contains a reference to an array; this reference is passed to the method.
  - Any changes to the array that occur inside the method body will affect the original array that was passed as the argument.

Simple Example

- Simple Example
  - SimpleExample.java

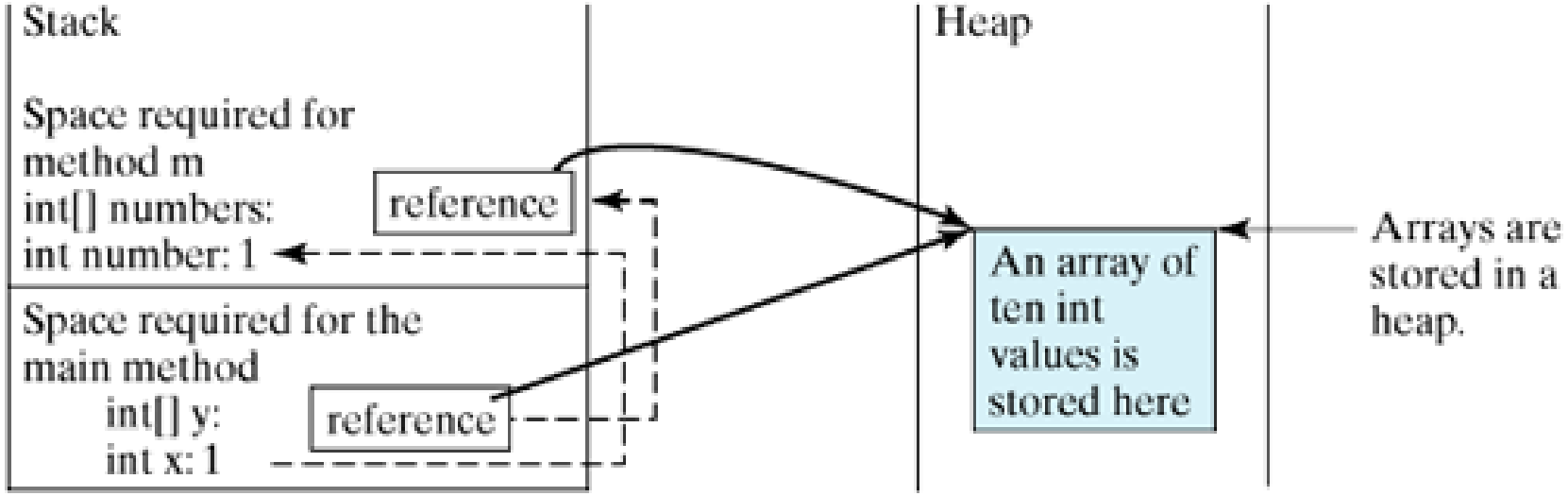
- Output:

  x is 1

  y[0] is 5555

Passing Arrays to Methods

- The JVM stores the array in an area of memory called the heap, which is used for dynamic memory allocation where blocks of memory are allocated and freed in an arbitrary order.



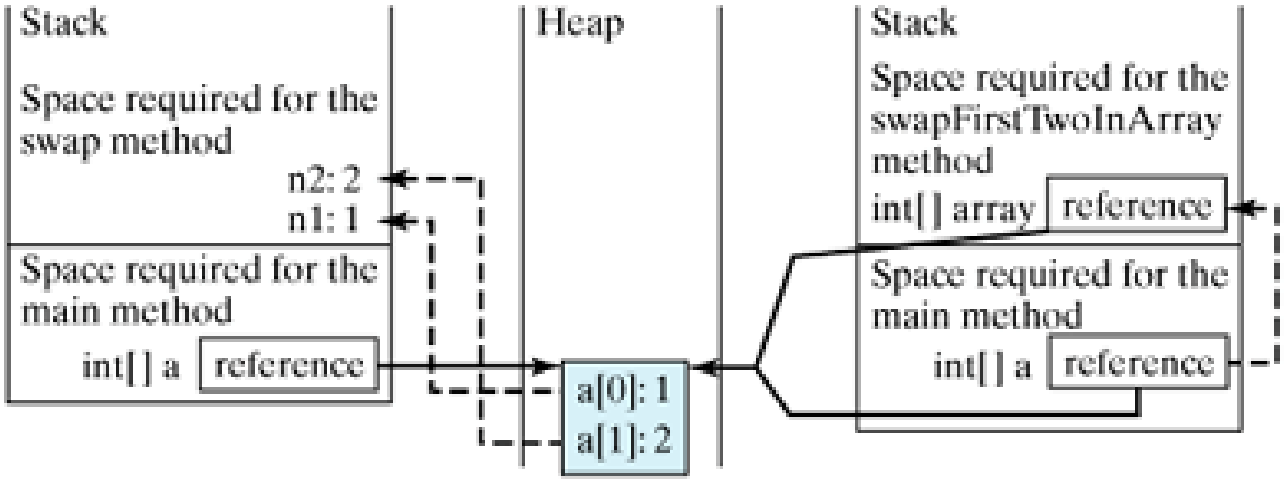
TestPassArray Program

- TestPassArray.java

- Output:

```
Before invoking swap
array is {1, 2}
After invoking swap
array is {1, 2}
Before invoking swapFirstTwoInArray
array is {1, 2}
After invoking swapFirstTwoInArray
array is {2, 1}
```

TestPassArray Program



Invoke `swap(int n1, int n2)`. The primitive type values in `a[0]` and `a[1]` are passed to the `swap` method.

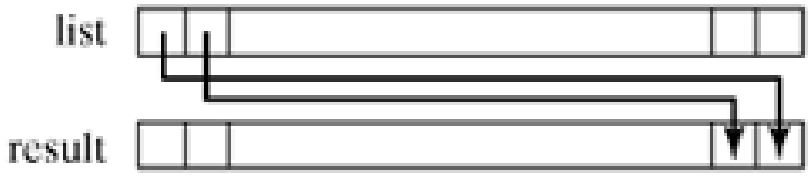
The arrays are stored in a heap.

Invoke `swapFirstTwoInArray(int[] array)`. The reference value in `a` is passed to the `swapFirstTwoInArray` method.

Returning an Array from a Method

- The method shown below returns an array that is the reversal of another array:

```
1 public static int[] reverse(int[] list) {
2   int[] result = new int[list.length];
3
4   for (int i = 0, j = result.length - 1;
5        i < list.length; i++, j--) {
6     result[j] = list[i];
7   }
8
9   return result;
10 }
```



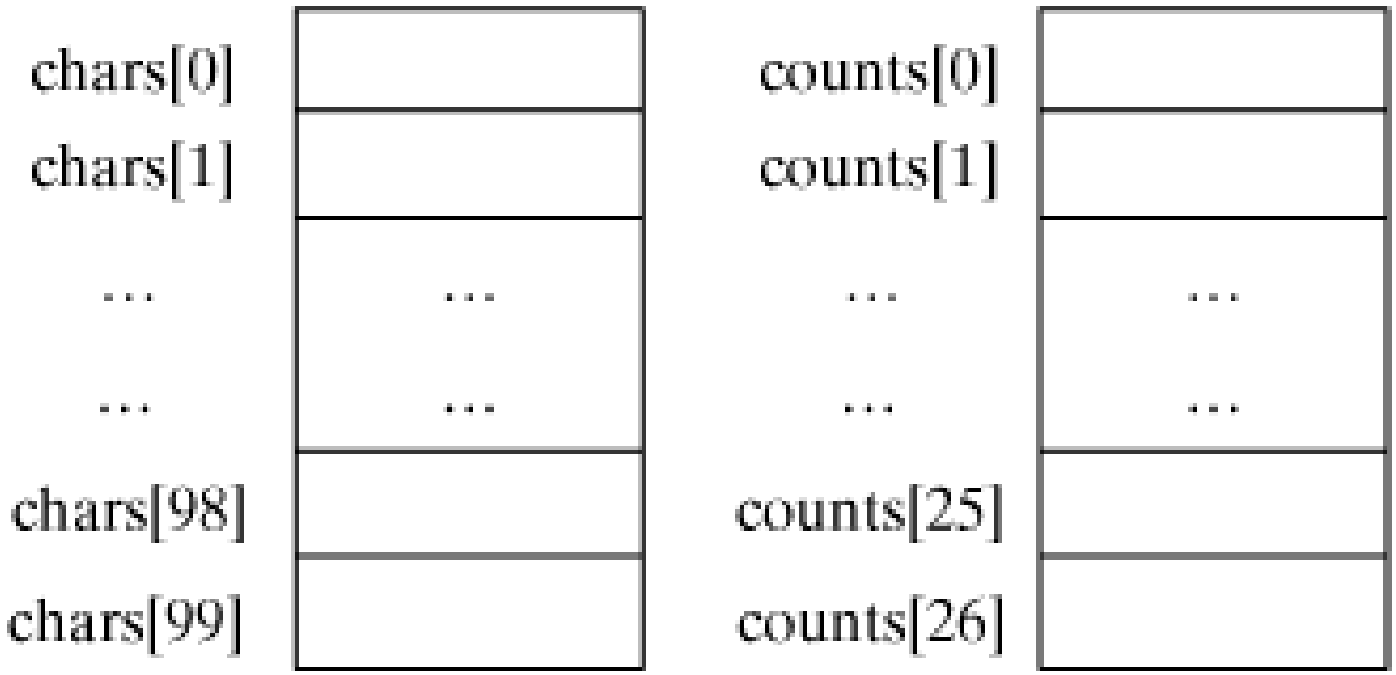
- The method can be invoked as below:

```
int[] list1 = {1, 2, 3, 4, 5, 6};
int[] list2 = reverse(list1);
```

- Trace the reverse method

Example: Counting the Occurrences of Each Letter

- Generate one hundred lowercase letters randomly and assign them to an array of characters and count the occurrences of each letter in the array.



Counting the Occurrences of Each Letter

- CountLettersInArray.java
- Output:

```
The lowercase letters are:
h y w f m f p h k r j r q l x d d o r f
o w w b f b r b c n z c y j z v t y q x
i e d q o f w o s c a n c d q d l s m e
y c d e r c g e m u o e i f d s f q b q
p w h p f w p u t w m h q z v f b o v f

The occurrences of each letter are:
1 a 5 b 6 c 7 d 5 e 10 f 1 g 4 h 2 i 2 j
1 k 2 l 4 m 2 n 6 o 4 p 7 q 5 r 3 s 2 t
2 u 3 v 7 w 2 x 4 y 3 z
```

Variable-Length Argument Lists

- Java enables you to pass a variable number of arguments of the same type to a method.
- The parameter in the method is declared as follows:

  ```
  typeName... parameterName
  ```

- Only one variable-length parameter may be specified in a method
- This parameter must be the last parameter.
- Any regular parameters must precede it.
- Java treats a variable-length parameter as an array.

Example

- Example:
  - VarArgsDemo.java
- Output:

```
The max value is 55.5
The max value is 3.0
No argument passed
```

References

References

- Y. Daniel Liang, **Introduction to Java Programming**, Sixth Edition, Pearson Education, 2007. (Chapter 5)
- S. Zakhour and et. al., **The Java Tutorial: A Short Course on the Basics**, 4th Edition, Prentice Hall, 2006. (Chapter 3)

The End