CS-201 Introduction to Programming with Java

California State University, Los Angeles
Computer Science Department

Lecture XI: One-Dimensional Arrays

Introduction

- Suppose: You need to read one hundred numbers, compute the average, and find out how many numbers are above the average, below the average, and equal to the average.

- Naive Approach: Use 100 different variables
  - You have to declare 100 variables and repeatedly write almost identical code one hundred times.
  - Very impractical.
  - What if you have 1000 variables?
  - What if you have a changing number of values?

Introduction

- A better approach is to use an ***array***
  - a data structure that stores a fixed-size sequential collection of elements of the same data type

- Store all 100 numbers into an array and access the numbers through a ***single array variable***

Introduction

- An array is often thought of as a collection of variables of the same type.

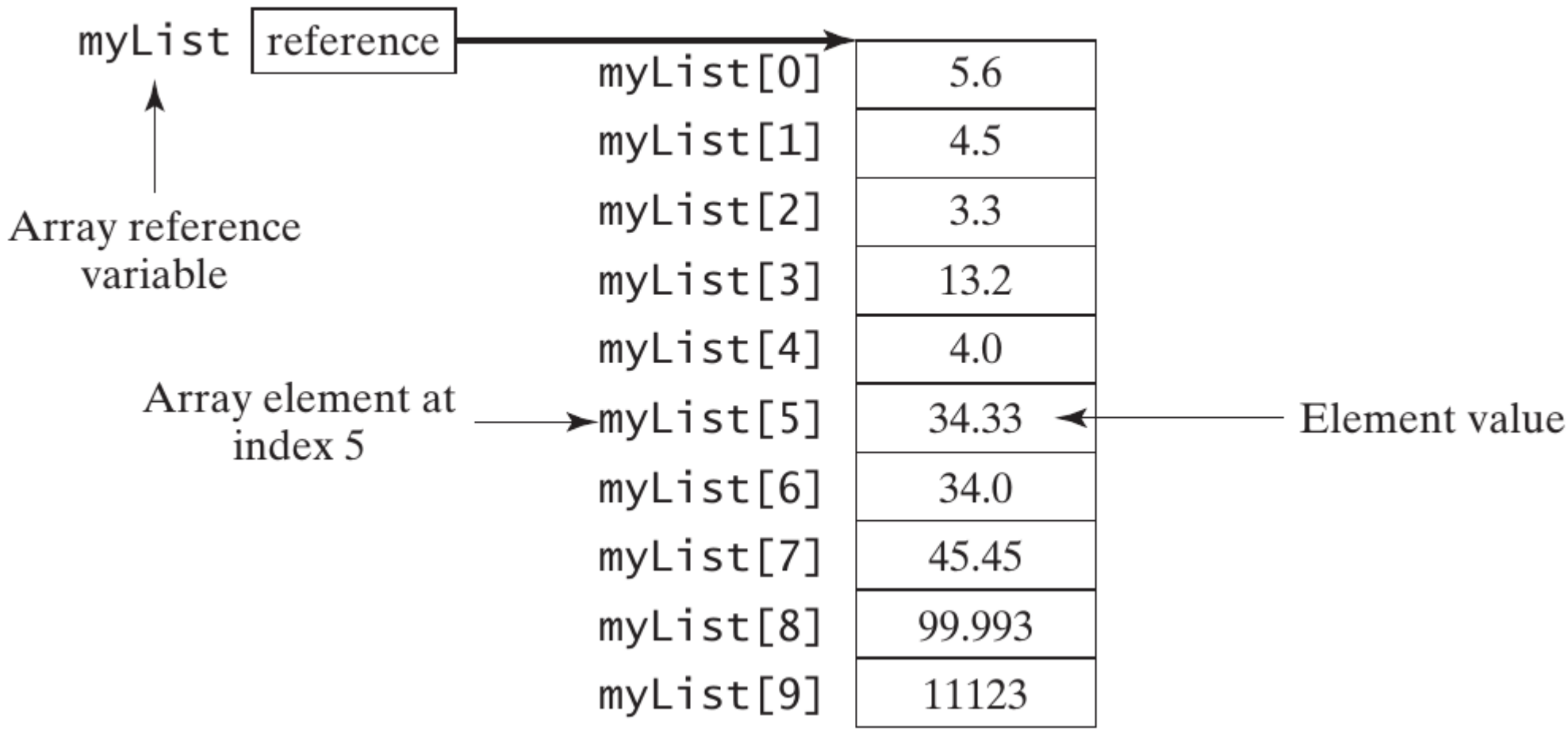
- Instead of declaring individual variables **`number0`**, **`number1`**, …, **`number99`**, we declare one array variable **`numbers`** and use **`numbers[0]`**, **`numbers[1]`**, …, **`numbers[99]`** to represent individual variables

Array Basics

Array Basics

- Example: A 10 element array of double values

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



Declaring Array Variables

- To use an array you have to declare a variable to ***reference*** the array.

- You also need to specify the ***type*** of the array.

- Syntax:

```
elementType[] arrayRefvar;
```

  - The **`elementType`** can be any type we have seen (**`int`**, **`double`**, **`char`**, etc...)
  - **`arrayRefvar`** can be any name you want (following rules for naming variables)
  - NOTE: that the **`[]`** are required because this distinguishes an array variable, from a regular variable (which can only hold one value)

Declaring Array Variables

- Example:

```
double[] myList;
```

  - declares a storage location for the ***reference*** to an array of **`double`** types.
  - This line **DOES NOT allocate any space in memory** for the array. only declares a ***reference variable*** which will reference an array at a future point in time.

Creating Arrays

- The previous slide illustrated how to create a reference to an array.
  - We only created a storage location for the reference variable NOT the array itself.
  - Again actual array has **NOT (again...NOT)** been created in memory yet.
  - If an array reference variable does not contain a reference to an array, the value of the variable is `null`
- We have to use the `new` operator to create the actual array in memory.

Creating Arrays

- Syntax:
  - Assign the result of **new** to the **arrayRefVar** created earlier

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

  - can create the reference variable, create the array, and assign it to the reference variable all in one step.

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

- Example:

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

  - Create an array of 10 **double** in memory
  - Assign the reference to the array, to the **myList** reference variable

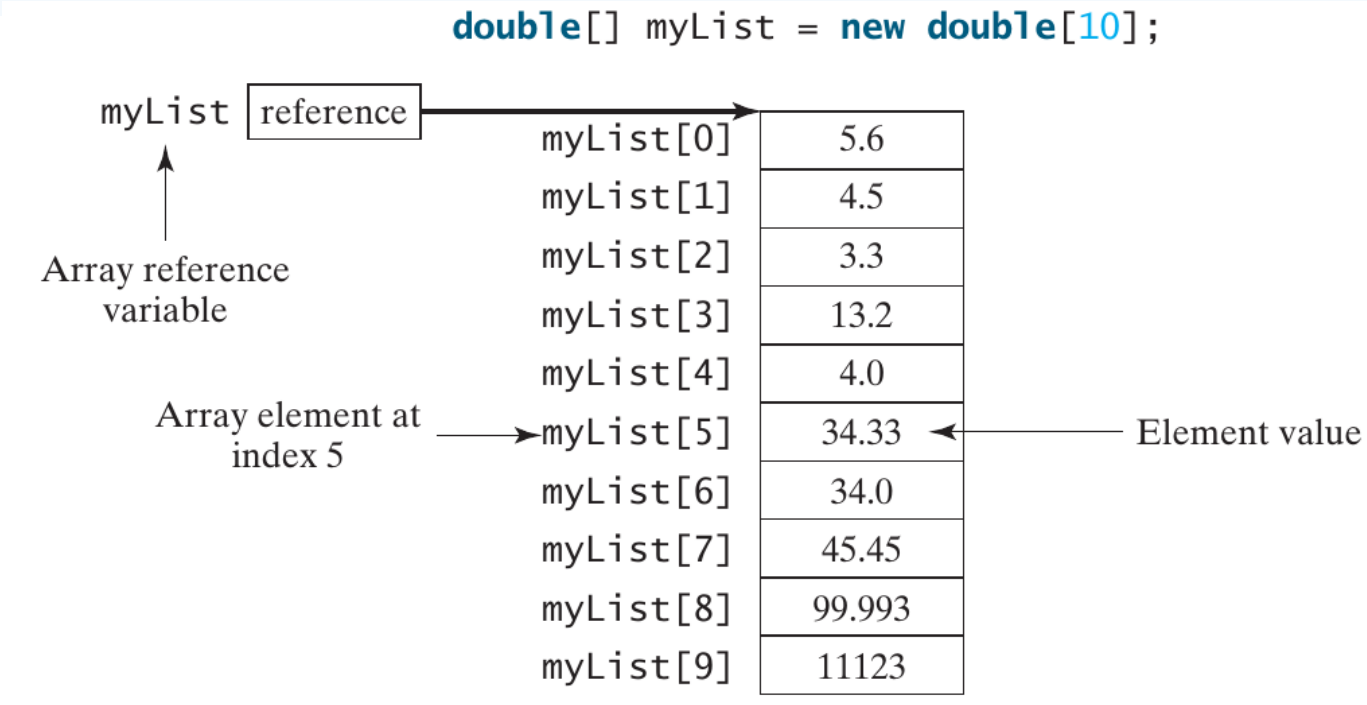
```
double[] myList = new double[10];
```

  - create a reference variable **myList**
  - create an array for 10 **double** elements in memory
  - Assign the reference to the array, to the **myList** reference variable

Creating Arrays

- Example: The following creates an array and assigns a value to each element of the array.

```
myList[0] = 5.6;
myList[1] = 4.5;
myList[2] = 3.3;
myList[3] = 13.2;
myList[4] = 4.0;
myList[5] = 34.33;
myList[6] = 34.0;
myList[7] = 45.45;
myList[8] = 99.993;
myList[9] = 11123;
```



Array Variables vs. the Actual Array

- An array variable only holds a **reference** to an array
- An array variable **IS NOT** the same thing as an array.
- To be technically correct, we would say "**`myList`** is a variable that contains a reference to an array of 10 double elements".

Array Size

- When you create an array using the **`new`** operator, you also have to specify the **size** of the array.

- Example: Specifying the size as an integer literal.
  - i.e. `int[] values = new int[5];`

- Example: Specifying the size as an integer variable.

```
int size = 5;
int[] values = new int[size];
```

Array Size

- Once you create an array in memory, **the size of the array is *fixed* and cannot be changed.**
  - NOTE: you may think something like the following will change the size of the array,

    ```
    values = new int[7];
    ```

    but you would be mistaken.
  - This would actually create an **entirely new** array in memory and assign its reference to **`values`**.
  - The old array (the one with a size of 5) now becomes ***garbage*** because nothing is referencing it anymore and it will eventually be freed up by the Garbage Collector.

Array Size

- Every array has a built in variable called **`length`** which gives you the size of the array.

- We can access the length variable using the array reference variable name, the dot **`.`** operator and the word **`length`**.

- Syntax: **`arrayRefVar.length`**

- Examples:
  - **`myList.length`** //this would return a value of 10
  - **`values.length`** // this would return a value of 5

Array Default Values

- When an array is created in memory, each of its elements are **always** assigned default values based on the type of the array. (This is before YOU assign anything to the array.)

- **`long`**, **`byte`**, **`int`**, **`short`**: 0
- **`double`**, **`float`**: 0.0
- **`char`**: \u0000
- **`boolean`**: **`false`**
- class/object types (including String): **`null`**

Accessing Array Elements

- If you want to access the value at a specific position in the array, you need to use the array reference variable name and the ***index*** or ***subscript*** where the value appears in the array.

- Arrays are **zero-indexed**
  - this means the first index of the array is **0 NOT 1**
  - this also means the values of the index range from **0** to **array.length - 1**

- The index must be a **positive integer**.
- The index can also be an **positive integer expression**
  - any expression which results in a positive integer value
  - Examples:

```
int a = 5;
c[a + 2] += 10; // This adds 2 + 5 to get an index of 7 and then adds 10 to the value stored in the array at index 7.
```

Accessing Array Elements

- When we access an element of the array using its index, we call the array reference variable + the index an ***indexed variable:***

```
arrayRefVar[indexNumber]
```

- These indexed variables can be used **just like any other variable with the same type**.

- Examples:

```
int index = 2;  //regular int variable, used as an index
myList[3] = 32.0; //assign 32.0 to index 3 of the array
myList[6] = myList[3] + 5; //assign 32.0 + 5 = 37. to index 6
System.out.println(myList[6]); //Print the value of index 6
myList[index+1] = 66.5; //assign 66.5 to the index + 1 = 3
```

Array Initializer Lists

- Another way to initialize an array without using the new operator, is to use an **initializer list** *or* **array initializer**

- Syntax:

```
elementType[] arrayRefVar = {value0, value1, …,
valuek};
```

- Element:

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

  - declares, creates, and initializes the array with four elements whose value are specified in the { }
  - this is equivalent to the following:

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

Array Initializer Lists

- NOTE: You do not use the **new** operator. You also must create the reference variable and use the initializer list all on one line. The following would cause a syntax error:

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

Passing Arrays to Methods

Passing Arrays to Methods

- Just like regular variables, you can pass an array to a method.
- Technically...you are passing the ***reference*** to the array, NOT a copy of the actual array.
- Arrays passed to a method use **pass-by-reference** same as class types.

Passing Arrays to Methods

- You can define a method which accepts an array argument.

```
public static void printArray(int[] array) {
  for (int i = 0; i < array.length; i++) {
    System.out.print(array[i] + " ");
  }
}
```

- You can then invoke the method by passing the reference variable to the method.

```
int[] list = {3, 1, 2, 6, 4, 2};
printArray(list);

printArray(new int[]{3, 1, 2, 6, 4, 2});
```

Anonymous Arrays

- The statement in the previous slide;

```
printArray(new int[]{3, 1, 2, 6, 4, 2});
```

  creates an array using the following syntax:

```
new dataType[]{value0, value1, ..., valuek};
```

- There is no explicit reference variable for the array.

- Such an array is called an anonymous array.
  - This array can never be reference beyond the line on which it is declared.
  - The only exception is if the anonymous array is passed to a method, then you can access the array through the method parameter.

Pass-by-Value vs. Pass-by-Reference

- Java uses ***pass-by-value*** to pass ***primitive arguments*** to a method.
  - a **copy of the value** is passed to the method parameter.
  - changing the value of the local parameter inside the method does NOT affect the value of the variable outside the method.

- Java uses ***pass-by-reference*** for ***array reference arguments***.
  - the reference to the array is passed to the method parameter
  - any changes to the array that occur inside the method body will affect the original array that was passed as the argument.

Example

```java
public class Test {
  public static void main(String[] args) {
    int x = 1; // x represents an int value
    int[] y = new int[10]; // y represents an array of int values

    m(x, y); // Invoke m with arguments x and y

    System.out.println("x is " + x);
    System.out.println("y[0] is " + y[0]);
  }

  public static void m(int number, int[] numbers) {
    number = 1001; // Assign a new value to number
    numbers[0] = 5555; // Assign a new value to numbers[0]
  }
}
```

```
x is 1
y[0] is 5555
```

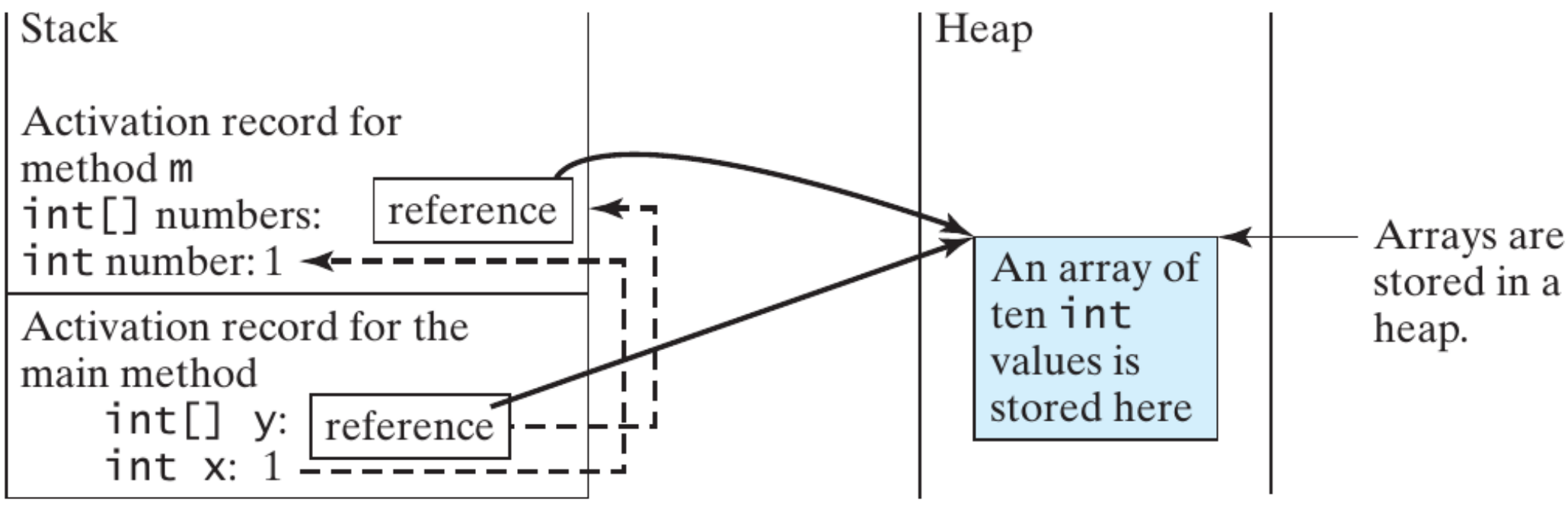
The Stack and Heap

- Why does x stay at 1, but y[0] changes to 5555?

- Recall that memory for methods and their local variables is allocated on the **stack.**

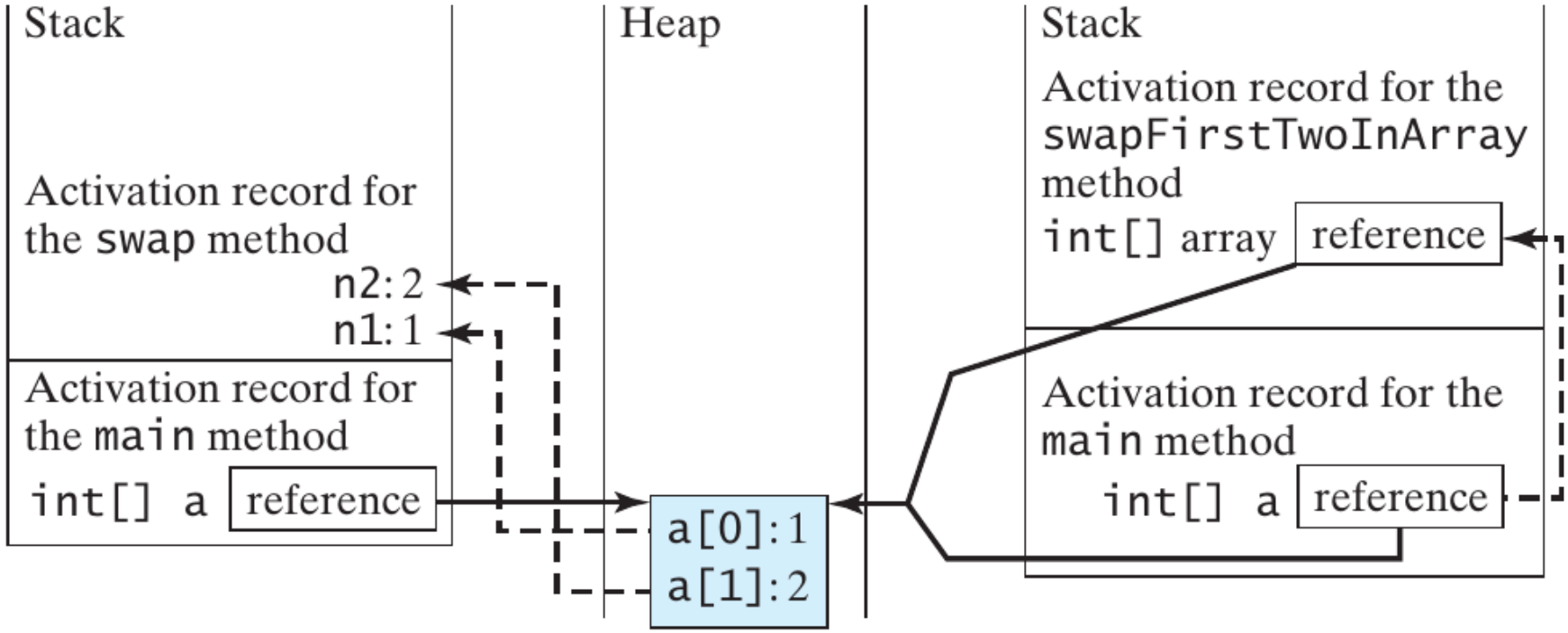
- Whenever you create an array, the memory which stores the array is allocated on the **heap.**
  - The **heap** is an area of memory used for **dynamic memory allocation**
  - Class types are also allocated memory on the **heap**

The Stack and Heap



Example

- See Code: **`TestPassArray.java`**



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

Returning an Array from a Method

- A method can return an array
  - The array should be created in the called method
  - The caller only needs to declare an array variable and assign the returned array to the variable

- Note: when passing an array to a method the array should be created BEFORE passing it to the method

```java
public static int[] reverse(int[] list) {
  int[] result = new int[list.length];

  for (int i = 0, j = result.length - 1;
       i < list.length; i++, j--) {
    result[j] = list[i];
  }

  return result;
}
```