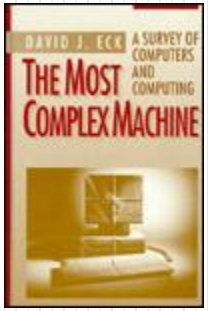
CIS2571 – Intro to Java

Chapter 6 → Single-Dimensional Arrays



R. Mukundan
Java Applets Centre
UC UNIVERSITY OF CANTERBURY

Information consolidated from a variety of textbook and online resources

Topic Objectives

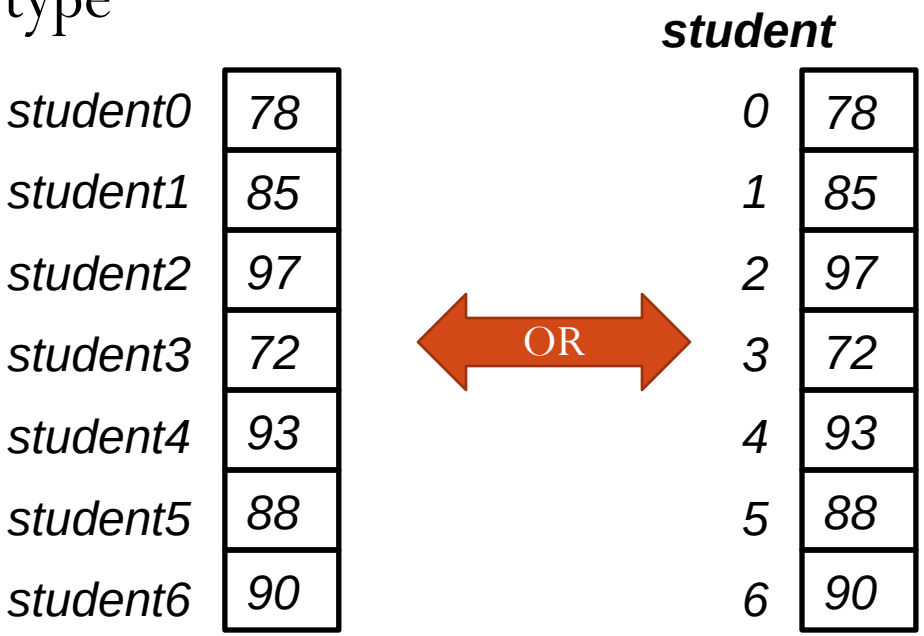
- Understand how an array is different than the primitive data types
- Know how to use arrays
  - Declare
  - Create
  - Access Elements
- Review some common array applications
- Know how to use the for-each looping construct for arrays
- Understand how arrays are:
  - Passed to methods
  - Returned from methods

Topic Objectives

- Recognize how variable-argument lists are interpreted as arrays
- Understand the different search techniques
  - Linear
  - Binary
- Understand the different sorting techniques
  - Selection
  - Insertion
- Know how to effectively use the Array class

What is an array?

- As programs get more complex, it often becomes necessary to store a large number of values during the execution of a program
- An array is used to store a fixed-size sequential collection of variables of the same type



Using Arrays

- Declaring Array Reference Variables
  - Declare a variable to reference the array and specify the array's element type

```
elementType[] arrayRefVar;
elementType arrayRefVar[]; // allowed, but !preferred
```

- Creating Arrays and Assigning Reference Variable
  - Cannot assign elements to array unless it has been created
  - Array size must be specified, cannot change after creation
  - Elements initialized to:
    - 0 for numerical
    - '\u0000' for char (nul)
    - false for boolean

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

unlike primitive data types, array declaration **does not allocate** any space in memory; it only creates a storage location for reference to an array

Using Arrays

- If an array reference variable does not contain a reference to an array, it is set to null
- Declaring, Creating, and Assigning Arrays

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

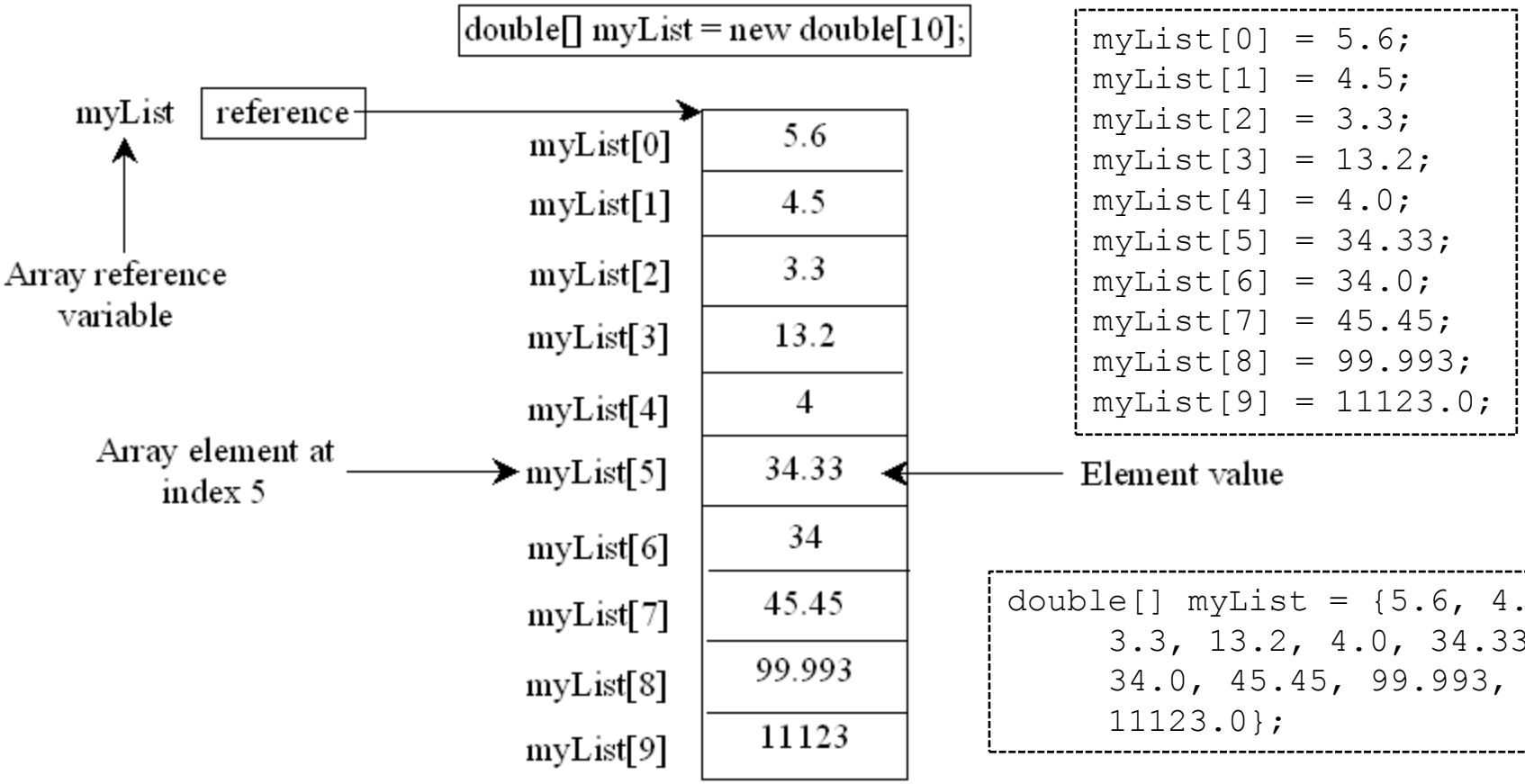
- Assigning values to array elements

```
arrayRefVar[index] = value;
```

- Array size can be obtained through **`arrayRefVar.length`**
- Array indexes are 0 based
  - **First** element has index **0**
  - **Last** element has index **`arrayRefVar.length-1`**
- Array initializer combines declaring, creating, and initializing into a single statement

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

Array Example




```
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] = 11123.0;
```

```
double[] myList = {5.6, 4.5,
     3.3, 13.2, 4.0, 34.33,
     34.0, 45.45, 99.993,
     11123.0};
```

Processing Arrays

- For loop is often used to process all elements of an array
  - All elements are of same type
  - Size of array is known
- Be careful of off-by-one error

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

Will throw "ArrayIndexOutOf BoundsException"

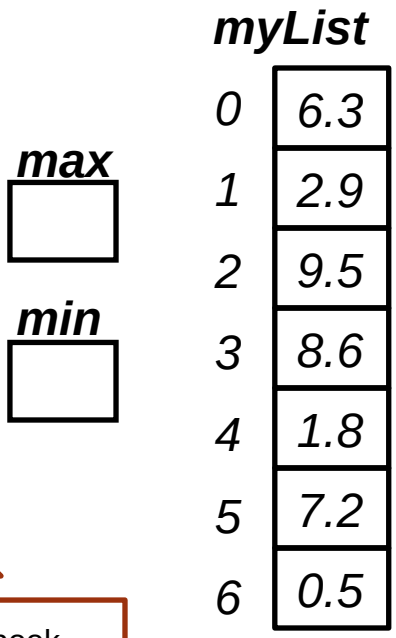
- Examples for processing arrays:
  - Initializing elements with values
  - Displaying elements
  - Summing elements
  - *Finding largest/smallest element*
  - *Copying elements*
  - *Random shuffling*
  - *Shifting elements*

See AnalyzeNumbers.java (p224)

Processing Arrays: Finding Largest/Smallest Element

- Use a variable to hold the largest/smallest number as comparisons are made with each element

```
// get largest element
double max = myList[0];
for (int i = 1; i < myList.length; i++) {
   if (myList[i] > max)
      max = myList[i];
}
// get smallest element
double min = myList[0];
for (int i = 1; i < myList.length; i++) {
   if (myList[i] < min)
      min = myList[i];
}
```

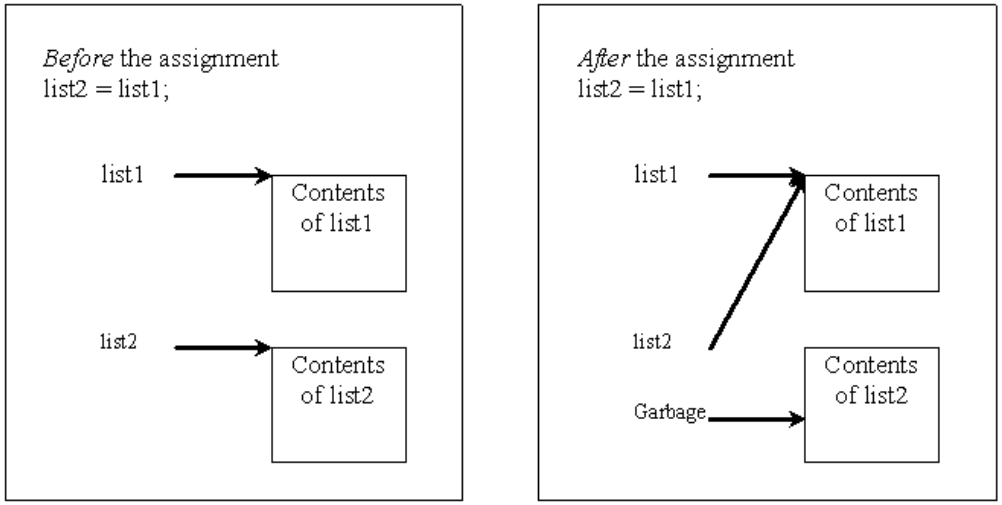


Used in textbook Listing 6.8 SelectionSort

Processing Arrays: Copying Elements

- Setting one array reference variable to another does not copy the elements, it will make both reference variables refer to the same array!

```
double[] list1 = {1.9, 2.9, 3.4, 3.5},
  list2 = new double[4];
```

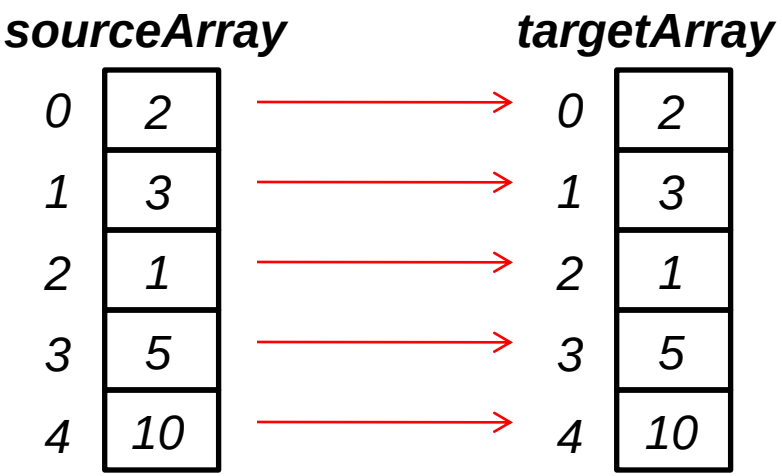


Processing Arrays: Copying Elements

- Use a loop to copy each array element

```
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];
```



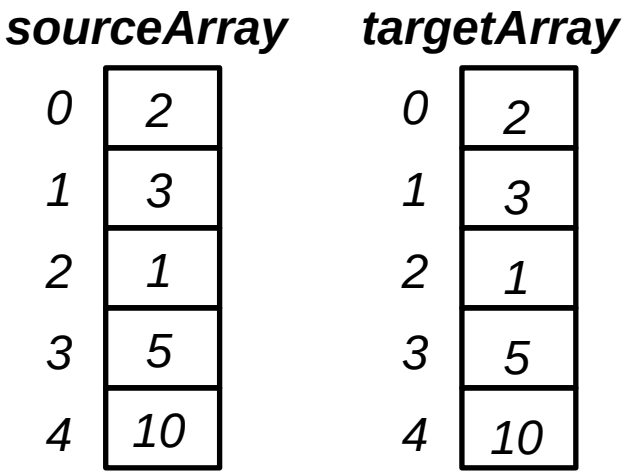
Processing Arrays: Copying Elements

- Use the arraycopy method in java.lang.System class
  - Target array must already be created and have enough space allocated
  - Format

```
arraycopy(sourceArray, src_pos, targetArray, tar_pos, length);
```

  - Examples

```
int[] sourceArray = {2, 3, 1, 5, 10};
int[] targetArray = new int[sourceArray.length];
System.arraycopy(sourceArray, 0, targetArray, 0,
    sourceArray.length);
```

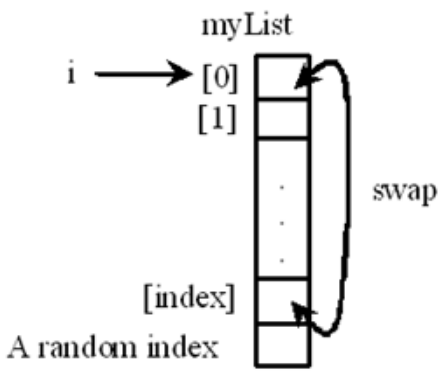


Processing Arrays: Random Shuffling

- Randomly reorder elements (shuffling)
  - Generate a random index for each array element and swap

```
for (int i = 0; i < myList.length; i++) {
   // Generate an index randomly
   int index = (int) (Math.random() * myList.length);

   // swap myList[i] with myList[index]
   double temp = myList[i];
   myList[i] = myList[index];
   myList[index] = temp;
}
```



Used in textbook
Listing 6.2 DeckOfCards

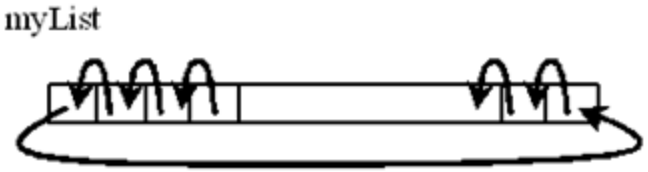
Processing Arrays: Shifting Elements

- Shift elements to the left or right
  - Fill last element with first element

```
// Retain first element
double temp = myList[0];

// Shift elements left
for (int i = 1; i < myList.length; i++) {
   myList[i - 1] = myList[i];
}

// Move first element to fill up the last position
myList[myList.length - 1] = temp
```



Used in textbook
Listing 6.9
InsertionSort

Processing Array: For-each Loops

- JDK 1.5 introduced a new for loop that enables sequential array traversal without using an index variable

```
for (elementType value: arrayRefVar) {
  // Process the value
}
```

- Use previous AnalyzeNumbers example to count number of elements above average:

```
for (double elVal: numbers) {
  if (elVal > average)
    count++;
}
```

- Accessing array elements in **different order,** or **changing element values** in array, **cannot** use for-each loop

Passing Arrays to Methods

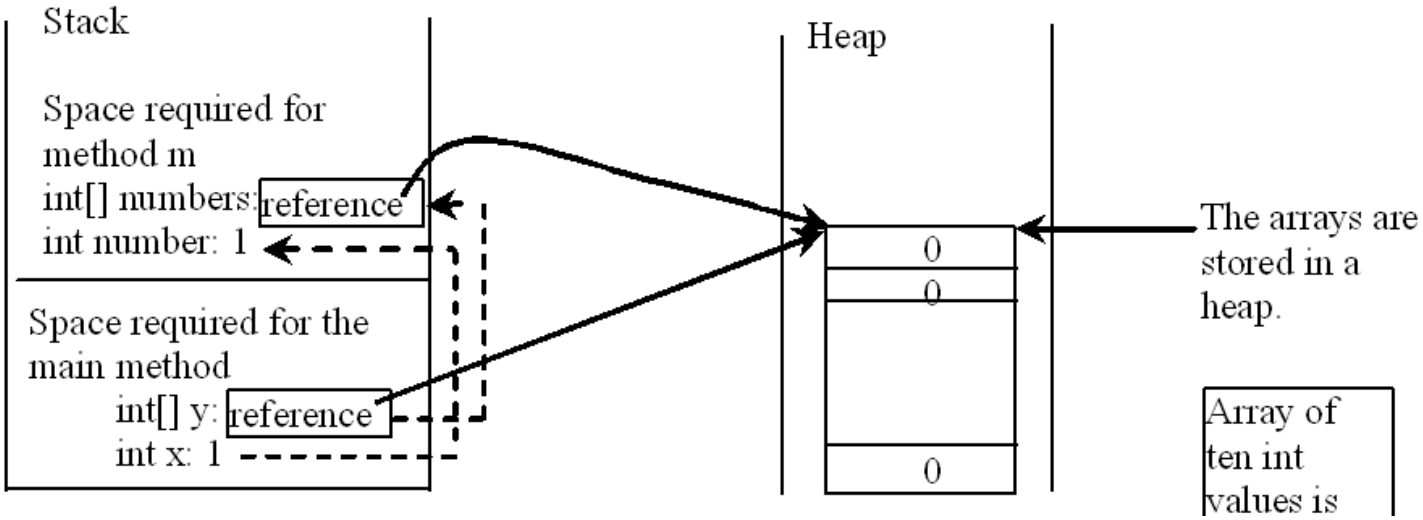
- Java uses pass-by-value when arguments are passed to method
  - Primitive data type → actual value is passed
  - Array data type → reference array variable is passed
- Pass-by-sharing used to describe array variable passing since underlying elements can be changed
  - No **direct** way to prevent modification of passed array elements
- When there is no explicit reference variable for an array, it is called an anonymous array

```
new elementType[]{val1, val2, . . . , valN}
```

- JVM stores array in area of memory called '**heap**', which is used for dynamic memory allocation

Passing Arrays to Method

- Call Stack contains variable that references an area in the **heap**
  - Used for dynamic memory allocation, and subsequent garbage collection, when memory is no longer accessible by **any** reference variable



Passing Arrays to Methods

Define the method

```
public static void printArray(int[] array) {
  for (int aVal: array) {
    System.out.print(aVal + " ");
  }
  print("\n");
}
```

Invoke the method

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

Invoke the method

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

anonymous array

See TestArrayMethods.java

Returning an Array from a Method

- Method can also return an array
- Return data type in method header:

```
elementType[]
```

- Create array in method definition
- Use created array in return statement
- Assign method return value to array reference variable:

```
elementType[] arrayRefVar = MethodName(methodArg);
```

- Memory allocated in method definition **will not** be reclaimed by garbage collection until **all** memory references are inaccessible

Returning an Array from a Method

Define the method

```
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;
}
```

new array created

array elements copied

array returned

Invoke the method

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

See TestArrayMethods.java

Variable-Length Argument Lists

- Can pass a variable number of arguments of the same type to a method
- **Only one** variable-length parameter may be specified in a method, and this parameter must be the **last** parameter.
- Format:

```
typeName... parameterName
```

- Java treats variable-length parameter as an array
  - Use `.length` to obtain array length
  - Use `[]` to access array elements

Variable-Length Argument Lists

Define the method

```
public static void printMax(double... numbers) {
  if (numbers.length == 0) {
    System.out.println("No argument passed");
    return;
  }
  double result = numbers[0];
  for (int i = 1; i < numbers.length; i++)
  if (numbers[i] > result)
    result = numbers[i];
  System.out.println("The max value is " + result);
}
```

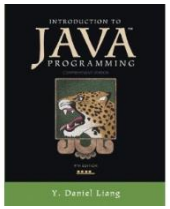
Invoke the method

```
printMax(34, 3, 3, 2, 56.5);
   // invoked with variable-length list
printMax(new double[]{1, 2, 3});
   // invoked with anonymous array
```

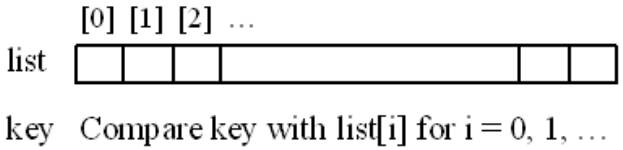
See 6.5 VarArgsDemo.java

Searching Arrays: Linear Search

- AKA sequential search
- Array elements are **unordered**
- Uses loop to sequentially step through an array
  - Compares each element with the value being searched
  - Stops when the value is found **or** the end of the array is reached




```
public class LinearSearch {
  /** The method for finding a key in the list */
  public static int linearSearch(int[] list, int key) {
    for (int i = 0; i < list.length; i++)
      if (key == list[i])
        return i;
    return -1;
  }
}
```

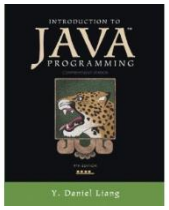


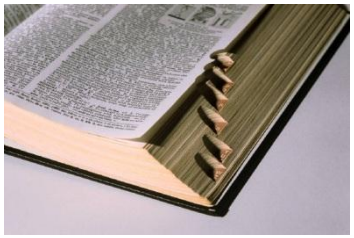
See 6.6 LinearSearch.java

Searching Arrays: Binary Search

- Locates item in an array by repeatedly dividing array in half and eliminating half of array to search.
  - Array elements must be in **sorted** order
  - It's more efficient than the sequential search
  - Continue dividing array in half, determine relevant half, stop when
    - match found
    - array location passed
  - Returns index of element if found, otherwise returns

    `-insertionPoint - 1`
    - `insertionPoint` is point at which key would be inserted into list
      - reverse sign and subtract 1 from returned value





Searching Arrays: Binary Search

- Binary Search Algorithm
  - Compare middle array element to search value
  - Middle element equals search value?
    - Match found!
  - Middle element greater than search value?
    - Eliminate second half of array
    - Check first half of array
  - Middle element less than the number?
    - Eliminate first half of array
    - Check second half of array

loop until match found or last index < first index

Searching Arrays: Binary Search

```
/** Use binary search to find the key in the list */
public static int binarySearch(int[] list, int key) {
  int low = 0;
  int high = list.length - 1;

  while (high >= low) {
    int mid = (low + high) / 2;
    if (key < list[mid])
      high = mid - 1;
    else if (key == list[mid])
      return mid;
    else
      low = mid + 1;
  }
  return -low - 1;
}
```

get lower and upper bounds

get array midpoint

eliminate upper half of array

value found

eliminate lower half of array

passed possible location

return –insertionPoint – 1

See 6.7 BinarySearch.java

Sorting Arrays: Selection Sort

- Finds the smallest number in unsorted list and places it as the first (sorted) element.
  - Continues to find smallest item in **shrinking unsorted** list, thus creating an **expanding sorted** list.

```
int[] myList = {2, 9, 5, 4, 8, 1, 6}; // Unsorted
```

Select 1 (the smallest) and swap it with 2 (the first) in the list	swap 2 9 5 4 8 1 6	
Select 2 (the smallest) and swap it with 9 (the first) in the remaining list	swap 1 9 5 4 8 2 6	The number 1 is now in the correct position and thus no longer needs to be considered.
Select 4 (the smallest) and swap it with 5 (the first) in the remaining list	swap 1 2 5 4 8 9 6	The number 2 is now in the correct position and thus no longer needs to be considered.
5 is the smallest and in the right position. No swap is necessary	1 2 4 5 8 9 6	The number 6 is now in the correct position and thus no longer needs to be considered.
Select 6 (the smallest) and swap it with 8 (the first) in the remaining list	swap 1 2 4 5 8 9 6	The number 5 is now in the correct position and thus no longer needs to be considered.
Select 8 (the smallest) and swap it with 9 (the first) in the remaining list	swap 1 2 4 5 6 9 8	The number 6 is now in the correct position and thus no longer needs to be considered.
Since there is only one element remaining in the list, sort is completed	1 2 4 5 6 8 9	The number 8 is now in the correct position and thus no longer needs to be considered.

Selection Sort: Algorithm

- Variables to hold **smallest value** and **index**

```
double currentMin;
int currentMinIndex;
```

- Nested loops
  - Outer loop goes iterates through all array elements

```
for (int i = 0; i < list.length; i++) {
```

  - Inner loop finds **smallest** of **remaining** array elements

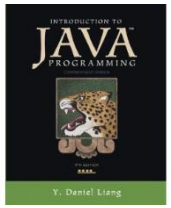
```
for (int j = i + 1; j < list.length; j++) {
    if (currentMin > list[j]) {
        currentMin = list[j];
        currentMinIndex = j;
    }
}
```

Selection Sort: Algorithm

- Swap elements if smaller item found in inner loop

```
if (currentMinIndex != i) {
    list[currentMinIndex] = list[i];
    list[i] = currentMin;
}
```

- Continue with next iteration of outer loop
- **Sorted** list increases (and **unsorted** list decreases) with each **full iteration** of inner loop



Selection Sort: Java Code

```
/** The method for sorting the numbers */
public static void selectionSort(double[] list) {
  for (int i = 0; i < list.length; i++) {
    // Find the minimum in the list[i..list.length-1]
    double currentMin = list[i];
    int currentMinIndex = i;
    for (int j = i + 1; j < list.length; j++) {
      if (currentMin > list[j]) {
        currentMin = list[j];
        currentMinIndex = j;
      }
    }
    // Swap list[i] with list[currentMinIndex] if necessary;
    if (currentMinIndex != i) {
      list[currentMinIndex] = list[i];
      list[i] = currentMin;
    }
  }
}
```

find smallest value and index

swap if smaller element found

See 6.8 SelectionSort.java

Sorting Arrays: Insertion Sort

- Work with **small sorted** sub-array to insert remaining **unsorted array elements** into the sorted sub-array until entire array is sorted

```
int[] myList = {2, 9, 5, 4, 8, 1, 6}; // Unsorted
```

Step 1: Initially, the sorted sublist contains the first element in the list. Insert 9 into the sublist.	2	9	5	4	8	1	6
Step2: The sorted sublist is {2, 9}. Insert 5 into the sublist.	2	9	5	4	8	1	6
Step 3: The sorted sublist is {2, 5, 9}. Insert 4 into the sublist.	2	5	9	4	8	1	6
Step 4: The sorted sublist is {2, 4, 5, 9}. Insert 8 into the sublist.	2	4	5	9	8	1	6
Step 5: The sorted sublist is {2, 4, 5, 8, 9}. Insert 1 into the sublist.	2	4	5	8	9	1	6
Step 6: The sorted sublist is {1, 2, 4, 5, 8, 9}. Insert 6 into the sublist.	1	2	4	5	8	9	6
Step 7: The entire list is now sorted.	1	2	4	5	6	8	9

Insertion Sort: Algorithm

- First array element is **sorted** list
- Nested loops
  - Outer loop goes iterates through **remaining** array elements

```
for (int i = 1; i < list.length; i++) {
```

  - Variables for **unsorted value** and **location**

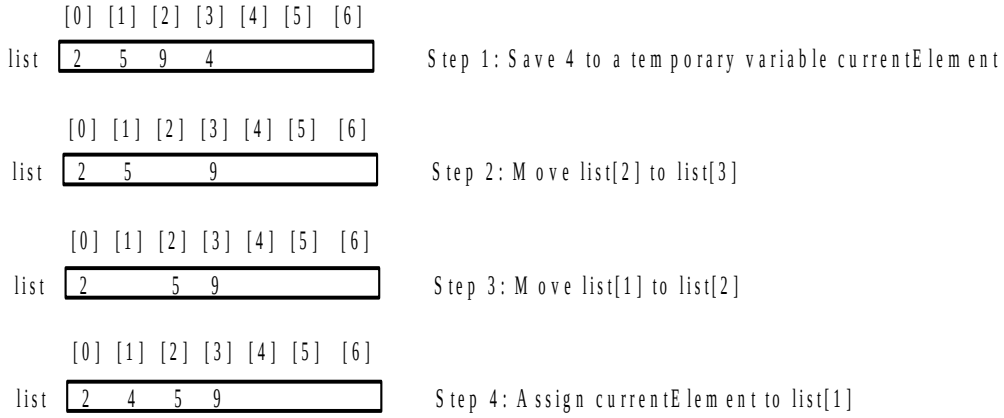
```
double currentElement = list[i];
int k;
```

Insertion Sort: Algorithm

- Inner loop finds location for **unsorted value** within **sorted** list by moving all **sorted** list elements **greater** than **unsorted value** starting at the end of the **sorted** list

```
for (k = i - 1;
        k >= 0 && list[k] > currentElement; k--) {
    list[k + 1] = list[k];
}
```

Example



Insertion Sort: Algorithm

- Insert **unsorted value** into proper **sorted** list location

```
list[k + 1] = currentElement;
```

- Continue with next iteration of outer loop
- **Sorted** list increases (and **unsorted** list decreases) with each **full iteration** of inner loop

Insertion Sort: Java Code

```
/** The method for sorting the numbers */
public static void insertionSort(double[] list) {
  for (int i = 1; i < list.length; i++) {
  /** insert list[i] into a sorted sublist list[0..i-1]
  so that list[0..i] is sorted. */
    double currentElement = list[i];
    int k;
    for (k = i - 1; k >= 0 &&
        list[k] > currentElement; k--) {
      list[k + 1] = list[k];
    }
    // Insert the current element into list[k+1]
    list[k + 1] = currentElement;
  }
}
```

find proper location for unsorted value while moving sorted values

insert unsorted value into sorted list

See 6.9 InsertionSort.java

Arrays Class

- **java.util.Arrays** class contains static methods for sorting and searching arrays
- Methods are overloaded for all primitive data types
  - sort → sorts a whole or partial array
  - binarySearch → searches array for element; returns index if found or - insertionPoint - 1
  - equals → check to see whether two arrays are equal (have the same contents)
  - fill → fill in all or part of an array

See TestArrayClass.java