Lesson 11:

Arrays Continued

(Updated for Java 1.5
Modifications by Mr. Dave Clausen)

Lesson 11: Arrays Continued

Objectives:

- Use String methods appropriately.
- Write a method for searching an array.
- Understand why a sorted array can be searched more efficiently than an unsorted array.
- Write a method to sort an array.

Lesson 11: Arrays Continued

Objectives:

- Write methods to perform insertions and removals at given positions in an array.
- Understand the issues involved when working with arrays of objects.
- Perform simple operations with Java's ArrayList class.

Lesson 11: Arrays Continued

Vocabulary:

- Binary search
- Bubble sort
- Immutable object
- Insertion sort
- Linear search
- Modal
- Selection sort
- Substring
- Wrapper class
- Array List

11.1 Advanced Operations on Strings

- Consider the problem of extracting words from a line of text.
- To obtain the first word, we could copy the string's characters to a new string until we reach the first space character in the string (assuming the delimiter between words is the space) or we reach the length of the string.
- Here is a code segment that uses this strategy:

11.1 Advanced Operations on Strings

```
// Create a sample string
String original = "Hi there!";

// Variable to hold the first word, set to empty string
String word = "";

// Visit all the characters in the string
for (int i = 0; i < original.length(); i++)
{

   // Or stop when a space is found
   if (original.charAt(i) == ' ')
      break;  //DO NOT USE BREAK WITH LOOPS!

   // Add the non-space character to the word
   word += original.charAt(i);
}
```

11.1 Advanced Operations on Strings

Code Without Break Statements

```
int i = 0;   //set loop counter to zero
boolean notFound = true;   //prime the while loop
String original = "Hi there!"; // Create a sample string
String word = ""; // Variable to hold the first word, set to empty string

// Visit all the characters in the string
while (i < original.length()&& notFound)
{
    // Or stop when a space is found
    if (original.charAt(i) == ' ')
      notFound = false;
    else
      word += original.charAt(i); // Add the non-space char to word

    i++;
}
```

11.1 Advanced Operations on Strings

- As you can see, this code combines the tasks of finding the first space character and building a ***substring*** of the original string.
- The problem is solved much more easily by using two separate String methods that are designed for these tasks. TestStringMethods.java TestStringMethods.txt

```
String original = "Hi there!";

// Search for the position of the first space
int endPosition = original.indexOf(' ');

// If there is no space, use the whole string
if (endPosition == -1)
   endPosition = original.length();

// Extract the first word
String word = original.substring(0, endPosition);
```

11.1 Advanced Operations on Strings

- Most text-processing applications examine and manipulate the characters in strings.
  - Separating strings into segments
  - Searching for/replacing specific characters or substrings
  - Inserting text into a string
- `String` objects are **immutable (can't be changed)**.
  - No mutators in the `String` class

11.1 Advanced Operations on Strings

METHOD	DESCRIPTION
`charAt (anIndex)` returns char	Example: `chr = myStr.charAt(4);` Returns the character at the position `anIndex`. Remember that the first character is at position 0. An exception is thrown (i.e., an error is generated) if `anIndex` is out of range (i.e., does not indicate a valid position within `myStr`).
`compareTo (aString)` returns int	Example: `i = myStr.compareTo("abc");` Compares two strings alphabetically. Returns 0 if `myStr` equals `aString`, a value less than 0 if `myStr` string is alphabetically less than `aString`, and a value greater than 0 if `myStr` string is alphabetically greater than `aString`.
`equals (aString)` returns boolean	Example: boolean = `myStr.equals("abc");` Returns true if `myStr` equals `aString`; else returns false. Because of implementation peculiarities in Java, never test for equality like this: `myStr == aString`
`equalsIgnoreCase (aString)` returns boolean	Similar to equals but ignores case during the comparison.
`indexOf (aCharacter)` returns int	Example: `i = myStr.indexOf('z')` Returns the index within `myStr` of the first occurrence of `aCharacter` or –1 if `aCharacter` is absent.

11.1 Advanced Operations on Strings

METHOD	DESCRIPTION
`indexOf (aCharacter, beginIndex)` returns int	Example: `i = myStr.indexOf('z', 6);` Similar to the preceding method except the search starts at position `beginIndex` rather than at the beginning of `myStr`. An exception is thrown (i.e., an error is generated) if `beginIndex` is out of range (i.e., does not indicate a valid position within `myStr`).
`indexOf (aSubstring)` returns int	Example: `i = myStr.indexOf("abc")` Returns the index within `myStr` of the first occurrence of `aSubstring` or –1 if `aSubstring` is absent.
`indexOf (aSubstring, beginIndex)` returns int	Example: `i = myStr.indexOf("abc", 6)` Similar to the preceding method except the search starts at position `beginIndex` rather than at the beginning of `myStr`. An exception is thrown (i.e., an error is generated) if `beginIndex` is out of range (i.e., does not indicate a valid position within `myStr`).
`length()` returns int	Example: `i = myStr.length();` Returns the length of `myStr`.
`replace (oldChar, newChar)` returns String	Example: `str = myStr.replace('z', 'Z');` Returns a new string resulting from replacing all occurrences of `oldChar` in `myStr` with `newChar`. `myStr` is not changed.

11.1 Advanced Operations on Strings

METHOD	DESCRIPTION
`substring (beginIndex)` returns String	Example: `str = myStr.substring(6);` Returns a new string that is a substring of `myStr`. The substring begins at location `beginIndex` and extends to the end of `myStr`. An exception is thrown (i.e., an error is generated) if `beginIndex` is out of range (i.e., does not indicate a valid position within `myStr`).
`substring (beginIndex, endIndex)` returns String	Example: `str = myStr.substring(4, 8);` Similar to the preceding method except the substring extends to location `endIndex` – 1 rather than to the end of `myStr`.
`toLowerCase()` returns String	Example: `str = myStr.toLowerCase();` `str` is the same as `myStr` except that all letters have been converted to lowercase. **`myStr` is not changed.**
`toUpperCase()` returns String	Example: `str = myStr.toUpperCase();` `str` is the same as `myStr` except that all letters have been converted to uppercase. **`myStr` is not changed.**
`trim()` returns String	Example: `str = myStr.trim();` `str` is the same as `myStr` except that leading and trailing spaces, if any, are absent. **`myStr` is not changed.**

11.1 Advanced Operations on Strings

Example 11.2: Count the words and compute the average word length in a sentence. (Don't use Break)

SentenceStats.java SentenceStats.txt

SentenceStatsNoBreak.java SentenceStatsNoBreak.txt

```
import java.util.Scanner;

public class SentenceStats{

   public static void main(String[] args){

      Scanner reader = new Scanner(System.in);

      // Keep taking inputs
      while (true){
         System.out.print("Enter a sentence: ");
         String input = reader.nextLine();

       // Quit when the user just presses Enter
       if (input.equals(""))
          break;

       // Initialize the counters and indexes
       int wordCount = 0;
       int sentenceLength = 0;
       int beginPosition = 0;
       int endPosition = input.indexOf(' ');
```

```
// Continue until a blank is not seen
while (endPosition != -1){

   // If at least one nonblank character (a word) was seen
   if (endPosition > beginPosition){
      wordCount++;
      String word = input.substring(beginPosition, endPosition);
      sentenceLength += word.length();
   }

   // Update the indexes to go to the next word
   beginPosition = endPosition + 1;
   endPosition = input.indexOf(' ', beginPosition);
}
```

11.1 Advanced Operations on Strings

- Example 11.2: Count the words and compute the average word length in a sentence (cont.).

```
            // If at least one nonblank character was seen
            // at the end of the sentence, consider it a word
            if (beginPosition < input.length()){
               wordCount++;
               String word = input.substring(beginPosition, input.length());
               sentenceLength += word.length();
            }

            // Trap the case where there were no words
            if (wordCount > 0){
               System.out.println("Word count: " + wordCount);
               System.out.println("Sentence length: " + sentenceLength);
               System.out.println("Average word length: " +
                                  sentenceLength / wordCount);
            }
         }
      }
   }
```

11.2 Linear Search

Linear Search (LinearSearch.ppt)

- In Lesson 9, we developed the code for a method that searches an array of int for a given target value.
- The method returns the index of the first matching value or -1 if the value is not in the array.
- The method examines each element in sequence, starting with the first one, to determine if a target element is present.
- The loop ends if the target is found.

11.2 Linear Search

- The method must examine every element to determine the absence of a target.
- This method of searching is usually called ***linear search***.

```
int Linear_Search (int[] a, int searchValue)
{
   for (int i = 0; i < a.length; i++)
      if (a[i] == searchValue)
         return i;
   return -1;
}
```

11.2 Linear Search

SEARCHING AN ARRAY OF OBJECTS

- Suppose we have an array of names that we wish to search for a given name.
- A name is a String.
- The change in the loop is that two string elements must be compared with the **method equals** instead of the operator ==.

```
int Linear_Search (String[] a, String searchValue){
   for (i = 0; i < a.length; i++)
      if (a[i].equals(searchValue))
         return i;
   return -1;
}
```

11.2 Linear Search

- This method can be generalized to work with any objects, not just strings.
- We simply substitute Object for String in the formal parameter list.
- The method still works for strings, and we can also use it to search an array of Student objects for a target student, assuming that the Student class includes an appropriate equals method.
- Following is a code segment that uses this single method to search arrays of two different types:

11.2 Linear Search

```
String[]  stringArray = {"Hi", "there", "Martin"};
Student[] studentArray = new Student[5];
Student   stu = new Student("Student 1", 100, 100, 100);

for (int i = 0; i < studentArray.length; i++)
    studentArray[i] = new Student("Student " + (i + 1), 100, 100, 100);

int stringPos = Linear_Search(stringArray, "Martin");
// Returns 2
int studentPos = Linear_Search(studentArray, stu);
// Returns 0
```

11.2 Binary Search

- **Binary search:** An efficient search algorithm based on eliminating half of the data from the search at each iteration
  - **The Data must be sorted first!**
  - Examine midpoint of data, then decide which half of the data to continue searching on.
    - Discard other half of data.
    - Repeat this process

GuessingGame.exe

11.2 Binary Search

Binary Search (using Iteration NOT Recursion)

- The method of linear search works well for arrays that are fairly small (a few hundred elements).
- As the array gets very large (thousands or millions of elements), the behavior of the search degrades.
- When we have an array of elements that are **in** ascending **order**, such as a list of numbers or names, there is a much better way to proceed, using an algorithm known as ***binary search***.
- This method is much faster than linear search for very large arrays.

11.2 Binary Search

- The basic idea of binary search is to examine the element at the array's midpoint on each pass through the search loop.
- If the current element matches the target, we return its position.
- If the current element is less than the target, then we search the part of the array to the right of the midpoint (containing the positions of the greater items).

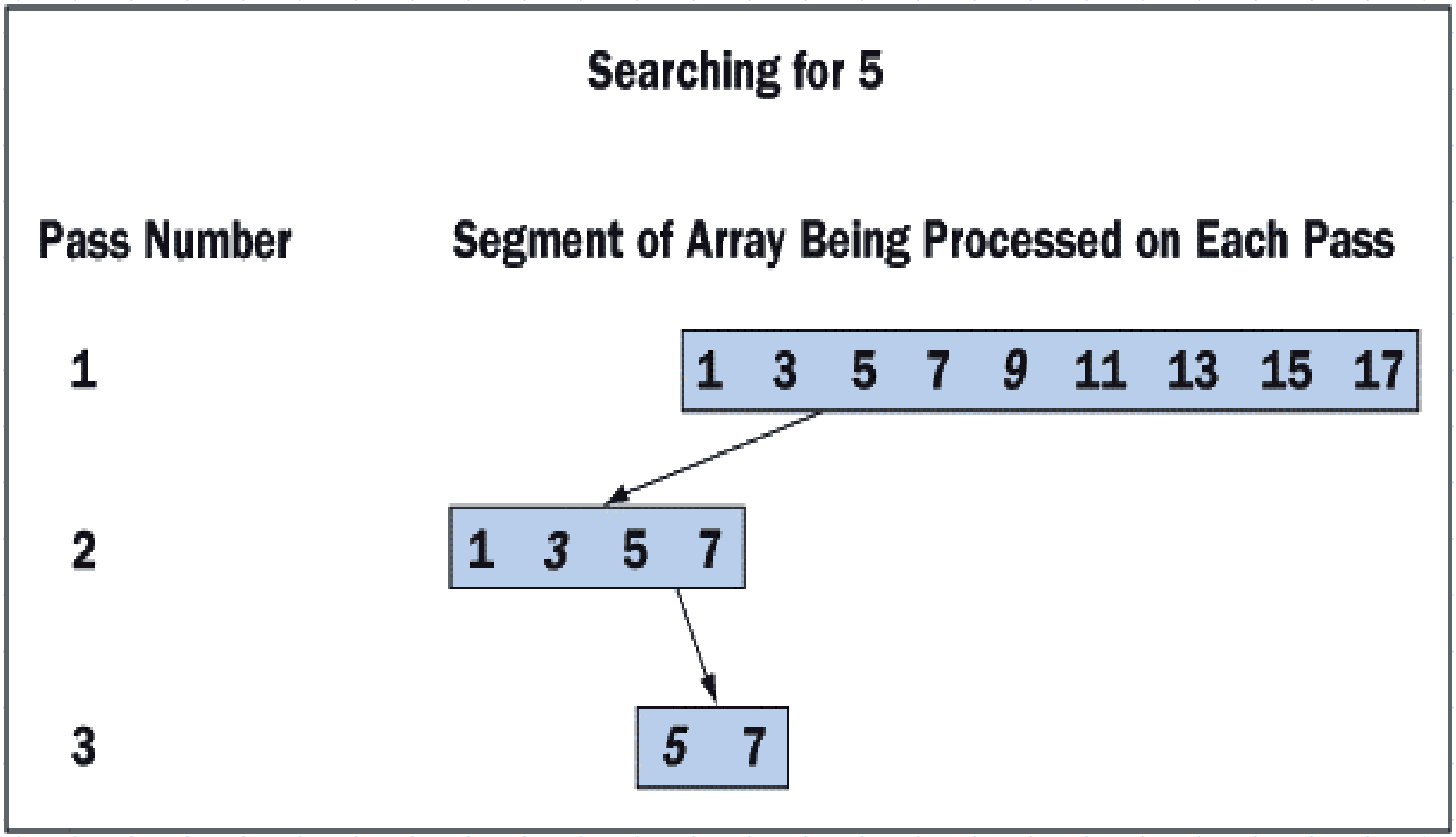
11.2 Binary Search

- Otherwise, we search the part of the array to the left of the midpoint (containing the positions of the lesser items).
- On each pass through the loop, the current leftmost position or the current rightmost position is adjusted to track the portion of the array being searched.

11.2 Binary Search

- Figure 11-1 shows a trace of a binary search for the target value 5 in the array 1 3 5 7 9 11 13 15 17.
- Note that on each pass through the loop, the number of elements yet to be examined is reduced by half.
- Herein lies the advantage of binary search over linear search for very large arrays.

11.2 Binary Search



Binary Search for Integers

(Using Iteration NOT Recursion)

```
//Binary Search for Integers –Finds 1st Occurrence Only
// Iterative binary search of an ascending array
int Binary_Search (int[ ] a, int searchValue){
   int left = 0;              // Establish the initial
   int right = a.length - 1;  // endpoints of the array
   while (left <= right){                  // Loop until the endpoints cross
      int midpoint = (left + right) / 2;  // Compute the current midpoint
      if (a[midpoint] == searchValue)  // Target found; return its index
         return midpoint;
      else if (a[midpoint] < searchValue) // Target to right of midpoint
         left = midpoint + 1;
      else                  // Target to left of midpoint
         right = midpoint - 1;
   }
   return -1;              // Target not found
}
```

11.2 Binary Search

COMPARING OBJECTS AND THE COMPARABLE INTERFACE

- When using binary search with an array of objects, we must compare two objects.
- But objects do not understand the < and > operators, and we have seen that == is not a wise choice for comparing two objects for equality.
- Classes that implement the `Comparable` interface include the method `compareTo`, which performs the three different comparisons.
- Here is the signature of `compareTo`:

```
public int compareTo(Object other)
```

11.2 Binary Search

- The behavior of compareTo is summarized in Table 11-2.

USAGE OF `compareTo`	VALUE RETURNED
`obj1.compareTo(obj2)`	0 if `obj1` is equal to `obj2`, using `equals`
`obj1.compareTo(obj2)`	A negative integer, if `obj1` is less than `obj2`
`obj1.compareTo(obj2)`	A positive integer, if `obj1` is greater than `obj2`

11.2 Binary Search

- For example, the String class implements the Comparable interface; thus, the second output of the following code segment is 0:

```
String str = "Mary";
System.out.println(str.compareTo("Suzanne"));
// Outputs -6
System.out.println(str.compareTo("Mary"));
// Outputs 0
System.out.println(str.compareTo("Bob"));
// Outputs 11
```

11.2 Binary Search

- The other output integers are system dependent, but the first should be negative while the third should be positive (in the example given, they are -6 and 11, respectively).
- Before sending the `compareTo` message to an arbitrary object, that object must be cast to `Comparable`, because `Object` does not implement the `Comparable` interface or include a `compareTo` method.
- Here is the code for the binary search of an array of objects:

11.2 Binary Search

(Using Iteration NOT Recursion)

```
//Binary Search for Objects – Finds 1st Occurrence Only
int Binary_Search (Object[] a, Object searchValue){
   int left = 0;
   int right = a.length – 1;
   while (left <= right){
    int midpoint = (left + right) / 2;
    int result =((Comparable)a[midpoint]).compareTo(searchValue);
      if (result == 0)
         return midpoint;
      else if (result < 0)
         left = midpoint + 1;
      else
         right = midpoint – 1;
   }
   return -1;
}
```

11.2 Search

IMPLEMENTING THE METHOD compareTo

- Objects that are ordered by the relations less than, greater than, or equal to must understand the `compareTo` message.
- Their class must implement the `Comparable` interface and their interface, if there is one, should also include the method `compareTo`.
- The Student class of Lesson 5, which has no interface, is modified to support comparisons of students' names.
- Below are the required changes to the code:

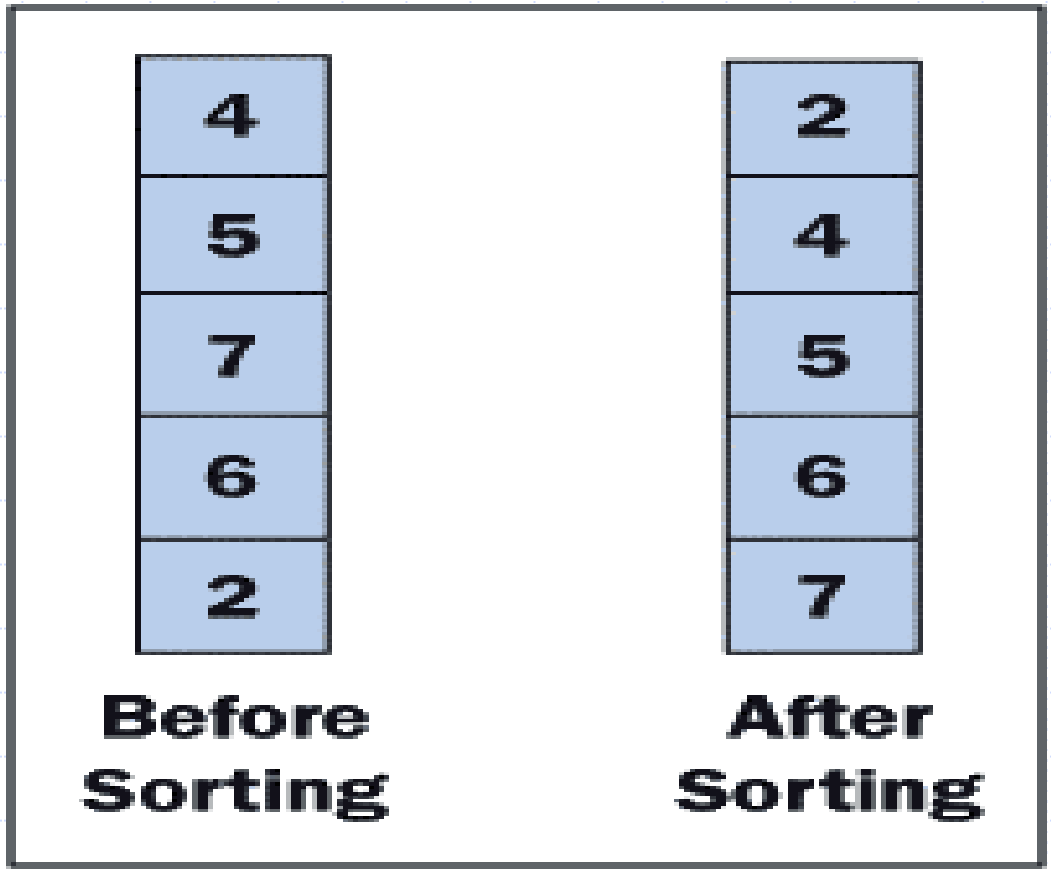
11.2 Searching

```
public class Student implements Comparable{

   <data declarations>

   public int compareTo(Object other){

      // The parameter must be an instance of Student
      if (! (other instanceof Student))
         throw new IllegalArgumentException("Parameter must be a Student");

      // Obtain the student's name after casting the parameter
      String otherName = ((Student)other).getName();

      // Return the result of comparing the two students' names
      return name.compareTo(otherName);
   }

   <other methods>
}
```

11.3 Sorting

- When the elements are in random order, we need to rearrange them before we can take advantage of any ordering.
- This process is called ***sorting***.
- Suppose we have an array **"a"** of five integers that we wish to sort from smallest to largest.
- In Figure 11-2, the values currently in **"a"** are as depicted on the left; we wish to end up with values as on the right.

11.3 Sorting



11.3 Sorting

Selection Sort

Selection Sort (SelectionSort.ppt)

- The basic idea of a selection sort is:

```
For each index position i
   Find the smallest data value in the array from positions i
   through length - 1, where length is the number of data
   values stored.

 Exchange the smallest value with the value at position i.
```

11.3 Sorting

Selection Sort

- Table 11-3 shows a trace of the elements of an array after each exchange of elements is made.
- The items just swapped are marked with asterisks, and the sorted portion is shaded.
- Notice that in the second and fourth passes, since the current smallest numbers are already in place, we need not exchange anything.
- After the last exchange, the number at the end of the array is automatically in its proper place.

11.3 Sorting

Selection Sort

UNSORTED ARRAY	AFTER 1ST PASS	AFTER 2ND PASS	AFTER 3RD PASS	AFTER 4TH PASS
4	1*	1	1	1
2	2	2*	2	2
5	5	5	3*	3
1	4*	4	4	4*
3	3	3	5*	5

11.3 Sorting

Selection Sort

- Before writing the algorithm for this sorting method, note the following:
  - **If the array is of length n, we need n - 1 steps.**
  - We must be able to find the smallest number.
  - We need to exchange appropriate array items.
- When the code is written for this sort, note that strict inequality (<) rather than weak inequality (<=) is used when looking for the smallest remaining value.

11.3 Sorting

Selection Sort

- The algorithm for selection sort is:
  - Note: The following algorithms and source codes assume that we are sorting the entire array (for all of the sorts in the rest of this lesson).
  - In other words, the logical size and the physical size are the same in these examples.
  - If this is not the case, don't forget to pass the variable logical_size and use this as the upper limit in loops (Replace a.length with logical_size).

```
For each i from 0 to n - 1 do
   Find the smallest value among a[i], a[i + 1],. . .a[n - 1]
   and store the index of the smallest value in minIndex
 Exchange the values of a[i] and a[index], if necessary
```

11.3 Sorting

Selection Sort

- Here we need to find the smallest value of array **“a”** .
- We will incorporate this segment of code in a method, findMinimum, for the selection sort.
- We will also use a method swap to exchange two elements in an array.

11.3 Sorting

Selection Sort

- Using these two methods, the implementation of a selection sort method is:

```
public static void SelectionSort(int[] a)
{
   for (int i = 0; i < a.length - 1; i++)
   {
      int minIndex = FindMinimum(a, i);
      if (minIndex != i)
         Swap(a, i, minIndex);
   }
}
```

11.3 Sorting

Selection Sort

- The method for finding the minimum value in an array takes two parameters:
  - the array
  - and the position to start the search.
- The method returns the index position of the minimum element in the array.
- Its implementation uses a for loop:

```
public static int FindMinimum(int[] a, int first){
   int minIndex = first;

   for (int i = first + 1; i < a.length; i++)
      if (a[i] < a[minIndex])
         minIndex = i;

   return minIndex;
}
```

11.3 Sorting

Selection Sort

- The swap method exchanges the values of two array cells:

```
public static void Swap(int[] a, int x, int y)
{
   int temp = a[x];
   a[x] = a[y];
   a[y] = temp;
}
```

11.3 Sorting

Bubble Sort

Bubble Sort (BubbleSort.ppt)

- Given a list of items stored in an array, a bubble sort causes a pass through the array to compare adjacent pairs of items.
- Whenever two items are out of order with respect to each other, they are swapped.
- The effect of such a pass through an array of items is traced in Table 11-4.
- The items just swapped are marked with asterisks, and the sorted portion is shaded.

11.3 Sorting

Bubble Sort

Bubble Sort

- Notice that after such a pass, we are assured that the array will have the item that comes last in order in the final array position.
- That is, the last item will "sink" to the bottom of the array, and preceding items will gradually "percolate" or "bubble" to the top.

11.3 Sorting

Bubble Sort

UNSORTED ARRAY	AFTER 1ST PASS	AFTER 2ND PASS	AFTER 3RD PASS	AFTER 4TH PASS
5	4*	4	4	4
4	5*	2*	2	2
2	2	5*	1*	1
1	1	1	5*	3*
3	3	3	3	5*

11.3 Sorting

Bubble Sort

- A complete Java method to implement a bubble sort for an array of integers is shown next:

11.3 Sorting

Bubble Sort

```
public static void BubbleSort(int[] a){
   int k = 0;
   boolean exchangeMade = true;

   // Make up to n - 1 passes through array, exit early if no exchanges
   // are made on previous pass

   while ((k < a.length - 1) && exchangeMade){
      exchangeMade = false;
      k++;
      for (int j = 0; j < a.length - k; j++)
         if (a[j] > a[j + 1]){
            Swap(a, j, j + 1);
            exchangeMade = true;
         }
       }
}
```

11.3 Sorting

Insertion Sort

Insertion Sort (InsertionSort.ppt)

- The ***insertion sort*** attempts to take greater advantage of an array's partial ordering.
- The goal is that on the *kth* pass through, the *kth* item among:

```
a[0], a[1], ..., a[k]
```

- should be inserted into its rightful place among the first k items in the array.
- This is analogous to the fashion in which many people pick up playing cards and order them in their hands.

11.3 Sorting

Insertion Sort

```
For each k from 1 to n - 1 (k is the index of array element to insert)
   Set itemToInsert to a[k]
   Set j to k - 1
   (j starts at k - 1 and is decremented until insertion position is
found)
   While (insertion position not found) and (not beginning of array)
      If itemToInsert < a[j]
         Move a[j] to index position j + 1
         Decrement j by 1
      Else
         The insertion position has been found
    itemToInsert should be positioned at index j + 1
```

11.3 Sorting

Insertion Sort

- An insertion sort for each value of *k* is traced in Table 11-5.
- In each column of this table, the data items are sorted in order relative to each other above the item with the asterisk; below this item, the data are not affected.

UNSORTED ARRAY	AFTER 1ST PASS	AFTER 2ND PASS	AFTER 3RD PASS	AFTER 4TH PASS
2	2	1*	1	1
5 ←	5 (no insertion)	2	2	2
1	1←	5	4*	3*
4	4	4 ←	5	4
3	3	3	3 ←	5

11.3 Sorting

Insertion Sort

```
public static void insertionSort(int[ ] list){
    int itemToInsert, j; // On the kth pass, insert item k into its correct position among
    boolean stillLooking; // the first k entries in array.
    for (int k = 1; k < list.length; k++){
          // Walk backwards through list, looking for slot to insert list[k]
      itemToInsert = list[k];
      j = k - 1;
      stillLooking = true;
      while ((j >= 0) && stillLooking )
        if (itemToInsert  < list[j]) {
          list[j + 1] = list[j];
          j--;
        }else
          stillLooking = false;
       // Upon leaving loop, j + 1 is the index
      // where itemToInsert  belongs
     list[j + 1] = itemToInsert;
    }
}
```

11.3 Sorting

Insertion Sort

Sorting Arrays of Objects

- Any of the sort methods can be modified to sort arrays of objects.
- We assume that the objects implement the `Comparable` interface and support the method `compareTo`.
- Then, we simply replace the type of all array parameters with `Object` and make the appropriate use of `compareTo` where the comparison operators are used.

Find Minimum for Objects

```
//Change in selection sort call to FindMinimum
int FindMinimum(Object[ ] a, int first{
  int minIndex =first;



//Change in FindMinimum source code

   for (int i = first + 1; i < a.length; i++)
      if (((Comparable)a[i]).compareTo(a[minIndex])<0)
         minIndex = i;

   return minIndex;
}
```

Testing Sort Algorithms

TestSortAlgorithms.java

TestSortAlgorithms.txt

11.4 Insertions and Removals

- In this section, we show how to add or remove an element at the end of an array that is not full at arbitrary positions within an array.
- For simplicity, we make four assumptions:
  1. Arrays are of fixed size; thus, when an array becomes full, insertions are not performed.
  2. We are working with an array of objects, although we can modify the code to cover arrays of integers, employees, or whatever element type is desired.

11.4 Insertions and Removals

3. For a successful insertions,
   0 <= target index <= logical size.
   - The new element is inserted before the element currently at the target index, or after the last element if the target index equals the logical size.
4. For successful removals,
   0 <= target index < logical size.

- When an assumption is not satisfied, the operation is not performed and we return `false`; otherwise, the operation is performed and we return `true`.

11.4 Insertions and Removals

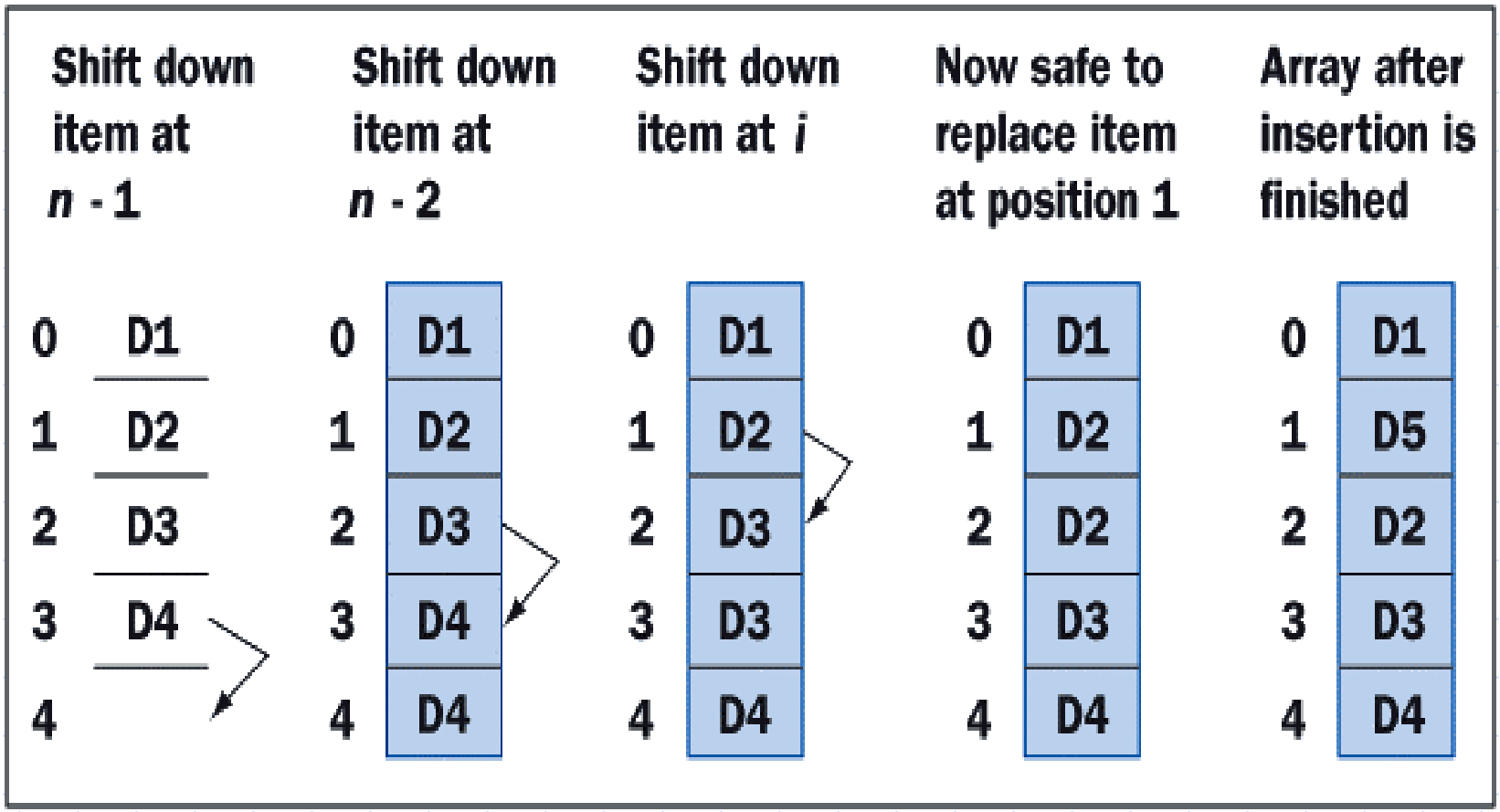
Inserting an Item into an Array at an Arbitrary Position

- Inserting an item into an array differs from replacing an item in an array.
- In the case of a replacement:
  - an item already exists at the given index position and a simple assignment suffices.
  - and the logical size of the array does not change.

11.4 Insertions and Removals

- In the case of an insertion, we must do six things:
  1. Check for available space before attempting an insertion; if there is no space, return false.
  2. Check the validity of the target index and return false if it is not >= 0 and <= logical size.
  3. Shift the items from the logical end of the array to the target index down by one position.
  4. Assign the new item to the cell at the target index.
  5. Increment the logical size by one.
  6. Return true.
- Figure 11-3 shows these steps for the insertion of an item at position 1 in an array of four items.

11.4 Insertions and Removals



11.4 Insertions and Removals

- As you can see, the order in which the items are shifted is critical.
- If we had started at the target index and copied down from there, we would have lost two items.
- Thus, we must start at the logical end of the array and work back up to the target index, copying each item to the cell of its successor.
- Next is the Java code for the insertion operation:

11.4 Insertions and Removals

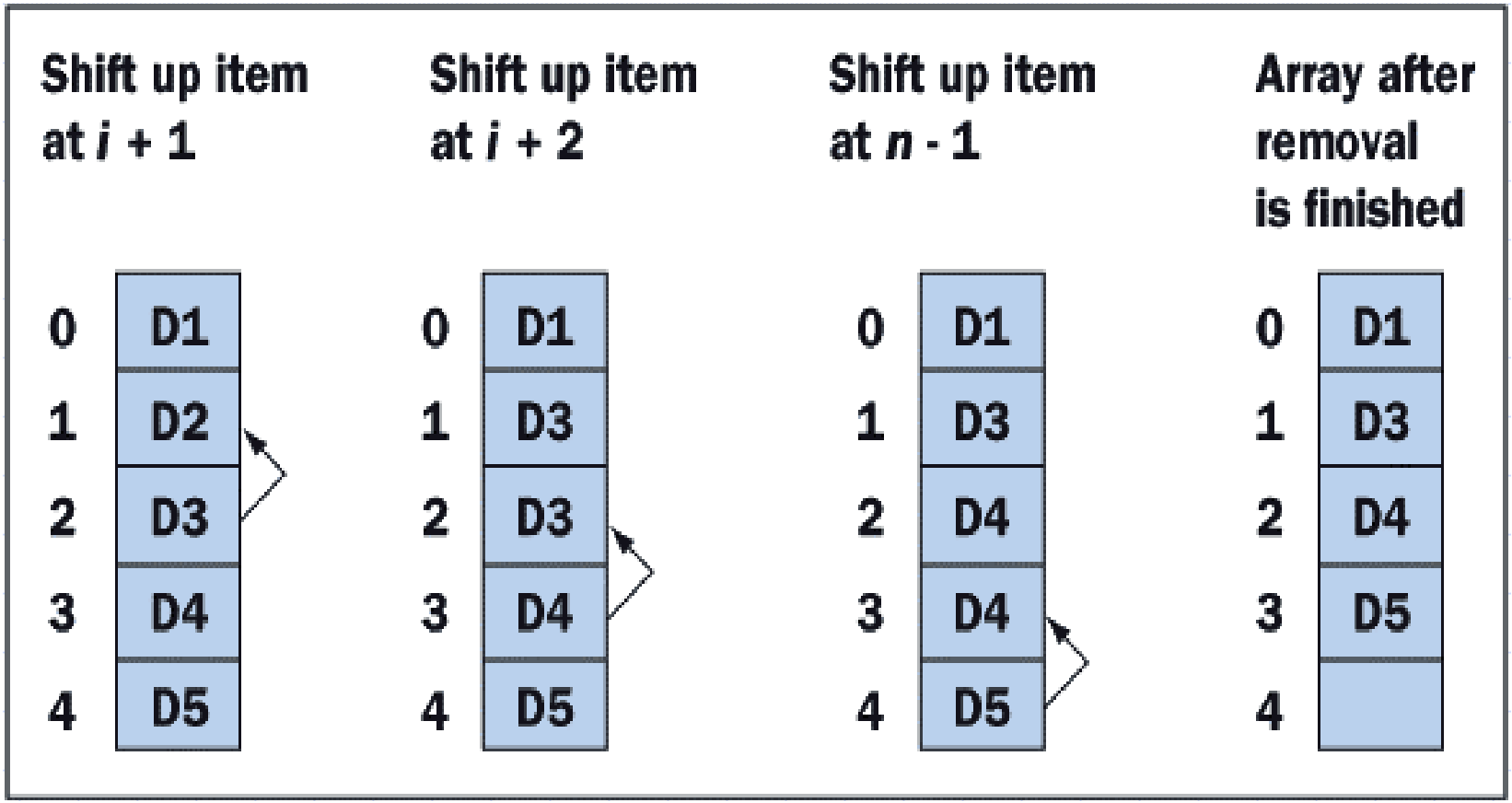
```
// Check for a full array and return false if full
if (logicalSize == array.length)
   return false;

// Check for valid target index return false if not valid
if (targetIndex < 0 || targetIndex > logicalSize)
   return false;

// Shift items down by one position
for (int i = logicalSize; i > targetIndex; i--)
   array[i] = array[i - 1];

// Add new item, increment logical size, and return true
array[targetIndex] = newItem;
logicalSize++;
return true;
```

11.4 Insertions and Removals

Removing an Item from an Array

- Removing an item from an array involves the inverse process of inserting an item into the array.
- Here are the steps in this process:
  1. Check the validity of the target index and return false if it is not >= 0 and < logical size.
  2. Shift the items from the target index to the logical end of the array up by one position.
  3. Decrement the logical size by one.
  4. Return true.
- Figure 11-4 shows these steps for the removal of an item at position 1 in an array of five items.

11.4 Insertions and Removals



11.4 Insertions and Removals

- As with insertions, the order in which we shift items is critical.
- For a removal, we begin at the item following the target position and move toward the logical end of the array, copying each item to the cell of its predecessor.
- Here is the Java code for the removal operation:

11.4 Insertions and Removals

```
// Check for valid target index return false if not valid
if (targetIndex < 0 || targetIndex >= logicalSize)
   return false;

// Shift items up by one position
for (int i = targetIndex; i < logicalSize - 1; i++)
   array[i] = array[i + 1];

// Decrement logical size and return true
logicalSize--;
return true;
```

TestInsertAndRemove.java TestInsertAndRemove.txt

11.5 Working with Arrays of Objects

Polymorphism, Casting, and `instanceof`

- It is quite common to declare and instantiate an array of some interface type.
- For instance, below is code that reserves 10 cells for shapes:

```
Shape[] shapes = new Shape[10];
```

11.5 Working with Arrays of Objects

- We can now store in this array instances of any class that implements `Shape`, such as `Rect`, `Circle`, or `Wheel`, as follows:

```
shapes[0] = new Rect(20, 20, 40, 40);// Cell 0 refers to a Rect
shapes[1] = new Circle(100, 100, 20);// Cell 1 refers to a Circle
shapes[2] = new Wheel(200, 200, 20, 6);// Cell 2 refers to a Wheel
```

- As long as we send `Shape` messages to the elements of this array, we can ignore the fact that they are of different concrete classes (that's what polymorphism is all about).

11.5 Working with Arrays of Objects

- Let us now draw all the shapes in the array:

```
Pen pen = new StandardPen();
for (int i = 0; i < 3; i++)
  shapes[i].draw(pen);
```

11.5 Working with Arrays of Objects

- Let us assume that we know the position of the wheel object in the array (in our example, it's at position 2).
- Then, to set its spokes to 5, we perform the following steps:
  1. Access the array element with the subscript.
  2. Cast the element, which is masquerading as a `Shape`, to a `Wheel`.
  3. Send the `setSpokes`(5) message to the result.
- Here is the code:

```
((Wheel) shapes[2]).setSpokes(5);
```

11.5 Working with Arrays of Objects

- Note the use of parentheses to override the precedence of the method selector, which would otherwise be run before the cast operation.
- Failure to cast in this code causes a compile-time error.
- Suppose we don't know the position of a wheel in the array of shapes, but we wish to set the spokes of each wheel to 5.
- We must first determine that a shape is a wheel before casting, and Java's instanceof operator comes to our rescue.

11.5 Working with Arrays of Objects

- Here is a loop that solves the problem:

```
for (int i = 0; i < shapes.length; i++)
   if (shapes[i] instanceof Wheel)
   ((Wheel) shapes[i]).setSpokes(5);
```

11.5 Working with Arrays of Objects

- Let us now summarize the use of objects in an array by making two fundamental points:
  1. When the element type of an array is a reference type or interface, objects of those types or any subtype (subclass or implementing class) can be directly inserted into the array.
  2. After accessing an object in an array, care must be taken to send it the appropriate messages or to cast it down to a type that can receive the appropriate messages.

11.7 The Class java.util.ArrayList

- Arrays are easiest to use when:
  - We know how many data elements will be added to them, so we don't run out of cells or waste any cells
  - We know they are full

    When these conditions do not hold, clients must
  - Find a way to increase the length of the array when it becomes full or shrink it when many cells become unoccupied
  - Track the array's logical size with a separate variable

11.7 The Class java.util.ArrayList

- Like most object-oriented languages, Java provides a wide range of classes for maintaining collections of objects.
- The collection class most like an array is called ArrayList and is included in the package java.util.

11.7 The Class java.util.ArrayList

- **Java 5 allows:**
  - **Generic array list:** Programmer must specify element type for the list
  - **Raw array list:** Can contain objects of any reference type
- Declaring/instantiating a generic array list:

```
import java.util.ArrayList;

ArrayList<String> list = new ArrayList<String>();
```

11.7 The Class java.util.ArrayList

Declaring and Instantiating an Array List

- An array list is an object, so it is instantiated like any other object, as in the following example:

```
import java.util.ArrayList;

ArrayList list = new ArrayList();
```

- Note that no initial length is specified in the array.
- An array list tracks its own physical size and logical size, which initially is 0.
- When a client inserts objects into an array list, the list automatically updates its logical size and adds cells to accommodate new objects if necessary.

11.7 The Class java.util.ArrayList

Using ArrayList Methods

- The programmer manipulates an array list by sending it messages.
- There are methods for examining an array list's logical size, testing it for emptiness (it's never full, at least in theory), insertions, removals, examining or replacing elements at given positions, and searching for a given element.
- Table 11-6 contains descriptions of these commonly used methods.

11.7 The Class java.util.ArrayList

ArrayList METHOD	WHAT IT DOES
`ArrayList<E>()`	Constructor builds an empty array list
`boolean isEmpty()`	Returns true if the list contains no elements and false otherwise
`int size()`	Returns the number of elements currently in the list
`E get(int index)`	Returns the element at index
`E set(int index, E obj)`	Replaces the element at index with `obj` and returns the old element
`void add(int index, E obj)`	Inserts `obj` before the element at index or after the last element if index equals the size of the list
`E remove(int index)`	Removes and returns the element at index
`int indexOf(E obj)`	Returns the index of the first instance of `obj` in the list or –1 if `obj` is not in the list

11.7 The Class java.util.ArrayList

- The items in an array list must all be objects:

```
ArrayList list = new ArrayList();
list.add (3.14);      // Invalid (compile-time error), 3.14 is not
                      // an object
list.add (new Student()); // Valid
```

11.7 The Class java.util.ArrayList

```
import java.util.ArrayList;
ArrayList <String> list = new ArrayList<String>();

for(int i=0; i<5; i++)
  list.add(i, “Item” + (i+1));
//The above list contains: Item1 Item2 Item3 Item4 Item5

for(int i=0; i<list.size(); i++)
  System.out.println(list.get(i)); //To display the list

//To display the list using the enhanced for loop
for(String str : list)
  System.out.println(str);
```

11.7 The Class java.util.ArrayList

- The next code segment performs some example searches:

```
System.out.println(list.indexOf("Item3")); // Displays 2
System.out.println(list.indexOf("Martin"));// Displays -1
```

- Our final code segment removes the first element from the list and displays that element and the list's size after each removal, until the list becomes empty:

```
while (! list.isEmpty()){
   Object obj = list.remove(0);
   System.out.println(obj);
   System.out.println("Size: " +
list.size());
}
```

11.7 The Class java.util.ArrayList

Array Lists and Primitive Types

- The array list is a powerful data structure, however, there are some restrictions on their use:
  1. an array list can contain only objects, not primitive types.
  2. although it's easy to insert any object into an array list, care must be taken when manipulating objects extracted from an array list.
- Remember that the elements come out as objects, which must be cast down to the appropriate classes before they are sent messages.

11.7 The Class java.util.ArrayList

Primitive Types and Wrapper Classes

- Java distinguishes between primitive data types (numbers, characters, Booleans) and objects (instances of `String`, `Employee`, `Student`, etc.).
- Variables and arrays can refer either to primitive data types or to objects, as in:

```
int x;                              // An integer
variable
int nums[];                       // An array of integers
Student student;                 // A Student variable
Student students[];             // An array of Students
```

11.7 The Class java.util.ArrayList

- There are occasions when we desire to store primitive data types in lists, which we can do if we first convert them to objects.
- Java provides wrapper classes for this purpose.
- Here is an example that illustrates how to convert an integer to and from its wrapper class Integer:

```
Integer intObj3 = new Integer(3);          // An Integer object containing 3
Integer intObj4 = new Integer(4);          // An integer object containing 4
int x = intObj3.intValue();                // Extracts 3 and saves in x
System.out.println(intObj3);               // Displays 3 using toString()
System.out.println(intObj3.equals(intObj4));       // Displays false
System.out.println(intObj3.compareTo(intObj4));    // Displays a negative
                                                   // number
```

11.7 The Class java.util.ArrayList

- `ArrayList` objects automatically "box" and "unbox" primitive values when used with `ArrayList` methods.

```
// Create a list of Integers
ArrayList<Integer> list = new ArrayList<Integer>();

// Add the ints 1-100 to the list
for (int i = 1; i <= 100; i++)
   list.add(i);

// Increment each int in the list
for (int i = 0; i < list.size(); i++)
   list.set(i, list.get(i) + 1);
```

11.7 The Class java.util.ArrayList

- As you can see from this code, the wrapper object intObject serves as a container in which the integer value 3 is stored or wrapped.
- The intValue method retrieves or unwraps the value in intObject.
- All wrapper objects understand the equals and compareTo messages, so they can be included in arrays that are searched and sorted.
- However, they cannot be used directly as operands in standard arithmetic expressions.
- Usually, casting as well as unwrapping is necessary.

10.7 The Class java.util.ArrayList

- Example 11.5: Test an array list of integers

TestArrayList.java

TestArrayList.txt

10.7 The Class java.util.ArrayList

- Like strings, wrapper objects are immutable; once they have been instantiated, the primitive values they contain cannot be changed.
- The wrapper classes for all the primitive data types are listed in following table.

10.7 The Class java.util.ArrayList

PRIMITIVE DATA TYPE	WRAPPER CLASS	METHOD FOR UNWRAPPING
boolean	Boolean	boolean booleanValue()
char	Character	char charValue()
byte	Byte	byte byteValue()
short	Short	short shortValue()
int	Integer	int intValue()
long	Long	long longValue()
float	Float	float floatValue()
double	Double	double doubleValue()

Arrays vs. ArrayList

- Advantages of `ArrayList` over arrays:
  - 1. Includes many methods for tasks such as insertions, removals, and searches.
  - 2. Tracks own logical size and grows or shrinks automatically with the number of elements contained in it.

Arrays vs. ArrayList

- An array might best be used when:
  - 1. We know the exact number of data elements to be inserted in advance.
  - 2. The operations to be performed are more specialized than the ArrayList provides.
    - Arrays have been around a long time, so legacy code is likely to use arrays.

Case Study

TestDesk.java

TestSuit.java

TestCard.java

Card.java

Deck.java

Suit.java

Summary

- **Linear search:** Simple search that works well for small- and medium-sized arrays
- **Binary search:** Clever search that works well for large arrays but assumes that the elements are sorted
- Comparisons of objects are accomplished by implementing the `Comparable` interface, which requires the `compareTo` method.

Summary

- Selection sort, bubble sort, and insertion sort are simple sort methods that work well for small- and medium-sized arrays.
- Insertions and removals of elements at arbitrary positions are complex operations that require careful design and implementation.

Summary

- One can insert objects of any class into an array of `Object`. When retrieved from the array, objects must be cast down to their classes before sending them most messages.
- The limitation of a fixed-size array can be overcome by using Java's `ArrayList` class.

Summary

- An array list tracks and updates its logical size and provides many useful client methods.
- Wrapper class, such as `Integer`, provides a way of packaging a value of a primitive type, such as `int`, in an object so that it can be stored in an array of `Object` or an array list.