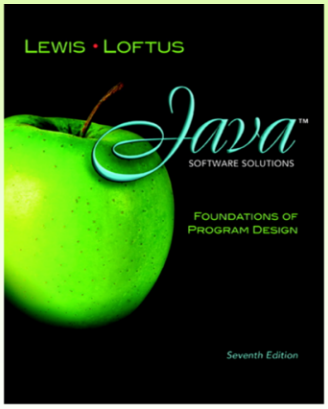
Chapter 10
Polymorphism



Java Software Solutions

Foundations of Program Design

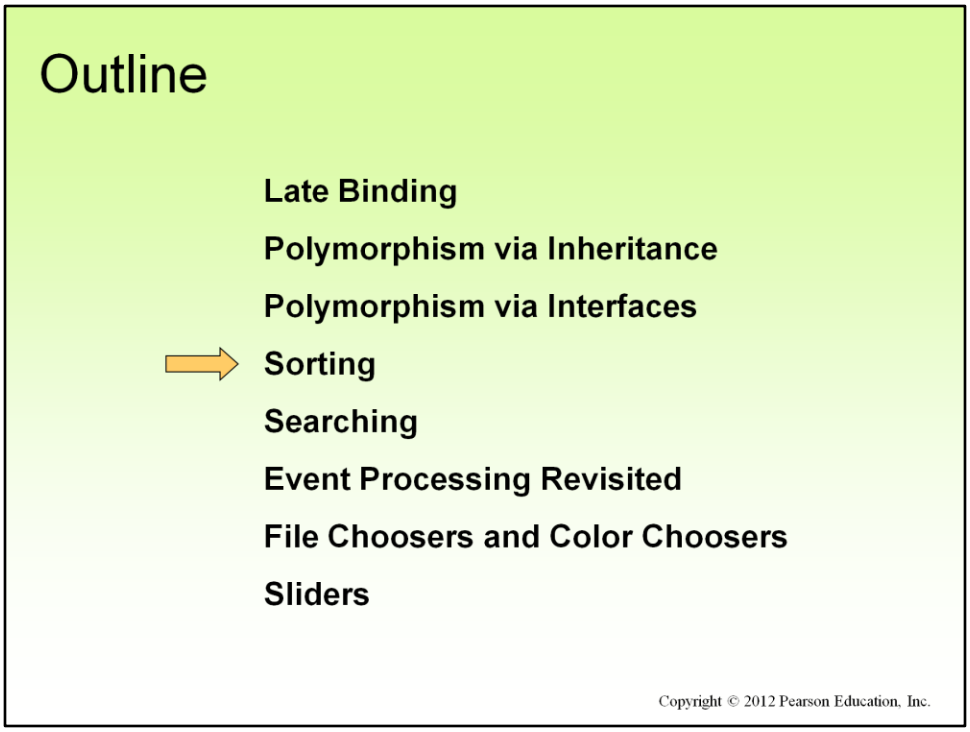
Seventh Edition

John Lewis
William Loftus

Addison-Wesley
is an imprint of
PEARSON

Polymorphism

- Polymorphism is an object-oriented concept that allows us to create versatile software designs
- Chapter 10 focuses on:
  - defining polymorphism and its benefits
  - using inheritance to create polymorphic references
  - using interfaces to create polymorphic references
  - using polymorphism to implement sorting and searching algorithms
  - additional GUI components



-We'll see polymorphism put into practice to implement two common operations in computer science

-These operations are ways in which we can **sort** and **search** multiple items in a collection

-Specifically, we'll see how we can use polymorphism with interfaces for these operations

-We'll begin in this section by looking at how we can implement ways to **sort** a number of items

-The key thing to remember in these examples is how we are not sorting **specific** types of class objects

-The methods used to sort items accept an array of **interface reference variables**, not specific class types

-As a result, the same method can be used for **any** class that implements the interface

-This allows us to write a sort method **one** time, and use it for lots of different class types!

-This is why polymorphism is so useful!

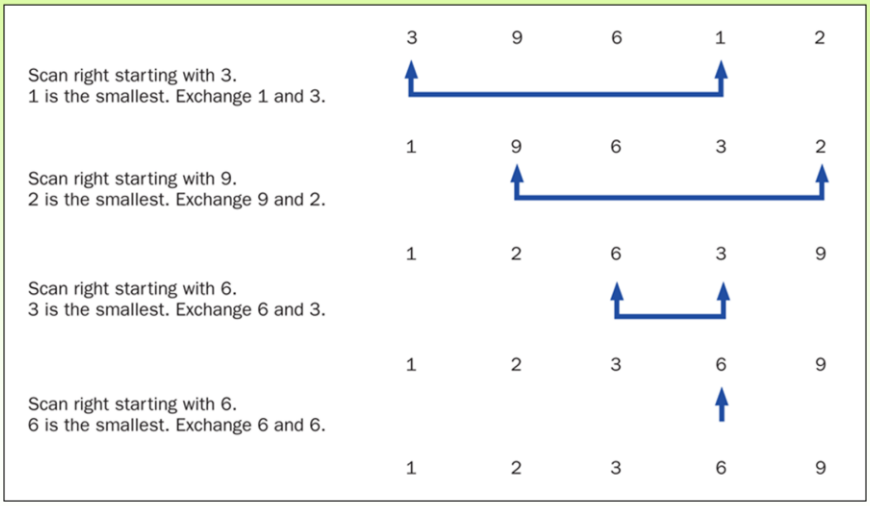
Sorting

- *Sorting* is the process of arranging a list of items in a particular order
- The sorting process is based on specific criteria:
  - sort test scores in ascending numeric order
  - sort a list of people alphabetically by last name
- There are many algorithms, which vary in efficiency, for sorting a list of items
- We will examine two specific algorithms:
  - Selection Sort
  - Insertion Sort

Selection Sort

- The strategy of Selection Sort:
  - select a value and put it in its final place in the list
  - repeat for all other values
- In more detail:
  - find the smallest value in the list
  - switch it with the value in the first position
  - find the next smallest value in the list
  - switch it with the value in the second position
  - repeat until all values are in their proper places

Selection Sort



Swapping

- The processing of the selection sort algorithm includes the *swapping* of two values
- Swapping requires three assignment statements and a temporary storage location
- To swap the values of `first` and `second`:

```
temp = first;
first = second;
second = temp;
```


-Note how we need a third temporary variable when swapping the value of two variables

-If we wanted to swap values in two variables named first and second, we perform it as follows

```
int first = 6;
int second = 5;
int temp;

temp = first;          // save the value of the first variable (stores 6)
first = second;  // replace first with second (stores 5)
second = temp;         // replace second with first that was saved in temp (stores 6)
```

Polymorphism in Sorting

- Recall that a class that implements the `Comparable` interface defines a `compareTo` method to determine the relative order of its objects
- We can use polymorphism to develop a generic sort for any set of `Comparable` objects
- The sorting method accepts as a parameter an array of `Comparable` objects
- That way, one method can be used to sort an array of `People`, or `Books`, or whatever

-The **Comparable** interface is an interface in the java.lang package in the Java API

-It is a simple interface with a single abstract method named **compareTo**

-If we write a class that **implements** this interface, we must provide a definition for this method

-In this method, we define how we want class objects to "compare" themselves to each other

-Since we write the method, we can define this comparison however we want

-We typically compare values of instance variables (class fields, or what it "knows")

-We compare values of the object that calls the method with the object passed to the method

-We return an integer value == zero (e.g. 0) to indicate objects are the same (or equal)

-We return an integer value < zero (e.g. -1) if the object calling the method is less than the object passed

-We return an integer value > zero (e.g. 1) if the object calling the method is greater than the object passed

Selection Sort

- This technique allows each class to decide for itself what it means for one object to be less than another
- Let's look at an example that sorts an array of `Contact` objects
- The `selectionSort` method is a static method in the `Sorting` class
- See `PhoneList.java`
- See `Sorting.java`
- See `Contact.java`

```
//********************************************************************
//  PhoneList.java       Author: Lewis/Loftus
//
//  Driver for testing a sorting algorithm.
//********************************************************************

public class PhoneList
{
   //-----------------------------------------------------------------
   //  Creates an array of Contact objects, sorts them, then prints
   //  them.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      Contact[] friends = new Contact[8];

      friends[0] = new Contact ("John", "Smith", "610-555-7384");
      friends[1] = new Contact ("Sarah", "Barnes", "215-555-3827");
      friends[2] = new Contact ("Mark", "Riley", "733-555-2969");
      friends[3] = new Contact ("Laura", "Getz", "663-555-3984");
      friends[4] = new Contact ("Larry", "Smith", "464-555-3489");
      friends[5] = new Contact ("Frank", "Phelps", "322-555-2284");
      friends[6] = new Contact ("Mario", "Guzman", "804-555-9066");
      friends[7] = new Contact ("Marsha", "Grant", "243-555-2837");
```

continue

continue

```
        Sorting.selectionSort(friends);

        for (Contact friend : friends)
           System.out.println (friend);
    }
}
```


-Note how we use a static method (selectionSort) of the Sorting class to sort the items

-This method accepts an array of Comparable interface reference variables

-To use this with the Contact class, we need to implement the Comparable interface in our Contact class

continue

```
      Sorting.select
      for (Contact f
         System.out.
   }
}
```

Output

```
Barnes, Sarah     215-555-3827
Getz, Laura       663-555-3984
Grant, Marsha     243-555-2837
Guzman, Mario     804-555-9066
Phelps, Frank     322-555-2284
Riley, Mark       733-555-2969
Smith, John       610-555-7384
Smith, Larry      464-555-3489
```

The static `selectionSort` method in the `Sorting` class:

```
//-----------------------------------------------------------------
//  Sorts the specified array of objects using the selection
//  sort algorithm.
//-----------------------------------------------------------------
public static void selectionSort (Comparable[] list)
{
   int min;
   Comparable temp;

   for (int index = 0; index < list.length-1; index++)
   {
      min = index;
      for (int scan = index+1; scan < list.length; scan++)
         if (list[scan].compareTo(list[min]) < 0)
            min = scan;

      // Swap the values
      temp = list[min];
      list[min] = list[index];
      list[index] = temp;
   }
}
```


-Note how this method accepts an array of Comparable interface reference variables

-Note how it uses the compareTo method of the variables to sort

-By using a generic interface, this method can be used with any class implementing the Comparable interface!

-We can say that this method is polymorphic since it can work with different types of class objects

-This is a great example of using polymorphism with interfaces!

```
//********************************************************************
//  Contact.java       Author: Lewis/Loftus
//
//  Represents a phone contact.
//********************************************************************

public class Contact implements Comparable
{
   private String firstName, lastName, phone;

   //-----------------------------------------------------------------
   //  Constructor: Sets up this contact with the specified data.
   //-----------------------------------------------------------------
   public Contact (String first, String last, String telephone)
   {
      firstName = first;
      lastName = last;
      phone = telephone;
   }
```

continue

-Note how we implement the Comparable interface so we can use the Sorting static method

continue

```
//-----------------------------------------------------------------
//  Returns a description of this contact as a string.
//-----------------------------------------------------------------
public String toString ()
{
   return lastName + ", " + firstName + "\t" + phone;
}

//-----------------------------------------------------------------
//  Returns a description of this contact as a string.
//-----------------------------------------------------------------
public boolean equals (Object other)
{
   return (lastName.equals(((Contact)other).getLastName()) &&
           firstName.equals(((Contact)other).getFirstName()));
}
```

continue

-Note how we are also overriding the equals method of the Object class

-This method returns whether objects are equal to one another

continue

```
//-----------------------------------------------------------------
//  Uses both last and first names to determine ordering.
//-----------------------------------------------------------------
public int compareTo (Object other)
{
   int result;

   String otherFirst = ((Contact)other).getFirstName();
   String otherLast = ((Contact)other).getLastName();

   if (lastName.equals(otherLast))
      result = firstName.compareTo(otherFirst);
   else
      result = lastName.compareTo(otherLast);

   return result;
}
```

continue

-Here, we write the definition for the compareTo method from the Comparable interface

-Note how this method accepts an object of the base Object type

-This is necessary so it can work with any type of class object

-As a result, note how we must type-cast this variable to call methods in our Contact class

((Contact)other).getFirstName()

-Note how we first check to see if the first and last names from the objects are equal

-If they are, we use the compareTo method of the String class to return the result of comparing first names

-If they are not, we use the compareTo method of the String class to return the result of comparing last names

continue

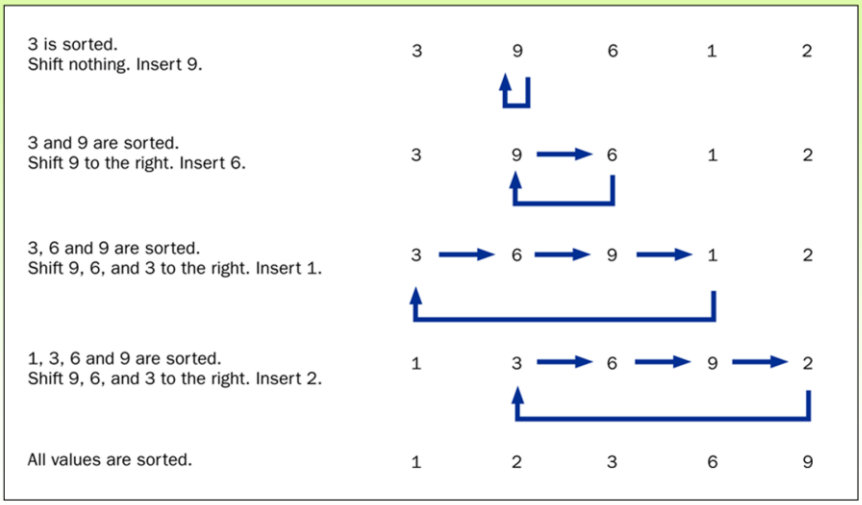
```
    //-----------------------------------------------------------------
    //  First name accessor.
    //-----------------------------------------------------------------
    public String getFirstName ()
    {
       return firstName;
    }

    //-----------------------------------------------------------------
    //  Last name accessor.
    //-----------------------------------------------------------------
    public String getLastName ()
    {
       return lastName;
    }
}
```

Insertion Sort

- The strategy of Insertion Sort:
  - pick any item and insert it into its proper place in a sorted sublist
  - repeat until all items have been inserted
- In more detail:
  - consider the first item to be a sorted sublist (of one item)
  - insert the second item into the sorted sublist, shifting the first item as needed to make room to insert the new one
  - insert the third item into the sorted sublist (of two items), shifting items as necessary
  - repeat until all values are inserted into their proper positions

-Insertion sort is another example of how we can sort items in a collection
-We can also implement this as a static method of our Sorting class accepting a Comparable array
-The only difference between using this method and the selection sort method is in how they sort the items

Insertion Sort



The static `insertionSort` method in the `Sorting` class:

```
//-----------------------------------------------------------------
//  Sorts the specified array of objects using the insertion
//  sort algorithm.
//-----------------------------------------------------------------
public static void insertionSort (Comparable[] list)
{
   for (int index = 1; index < list.length; index++)
   {
      Comparable key = list[index];
      int position = index;

      //  Shift larger values to the right
      while (position > 0 && key.compareTo(list[position-1]) < 0)
      {
         list[position] = list[position-1];
         position--;
      }

      list[position] = key;
   }
}
```

Comparing Sorts

- The Selection and Insertion sort algorithms are similar in efficiency
- They both have outer loops that scan all elements, and inner loops that compare the value of the outer loop with almost all values in the list
- Approximately n^2 number of comparisons are made to sort a list of size n
- We therefore say that these sorts are of *order* n^2
- Other sorts are more efficient: *order* $n \log_2 n$

-When there are many different ways (algorithms) to perform an operation, we need some way to compare them
-One way to compare is to look at how fast (or how efficient) algorithms perform to accomplish their tasks
-In the sorting methods, we can look at how many comparisons are made to perform a sort
-We represent a single comparison as **n** and determine how many **n**'s are needed in the algorithm
-We can determine that approximately **n*n** comparisons (or **n^2**) or needed in both algorithms
-We refer to the efficiency of these algorithms by saying they have **order n^2**
-If we find one that requires fewer than n^2, we can say that it is more efficient than these two methods
-Such notations are used to help compare and contrast relative efficiency of algorithms in computer science