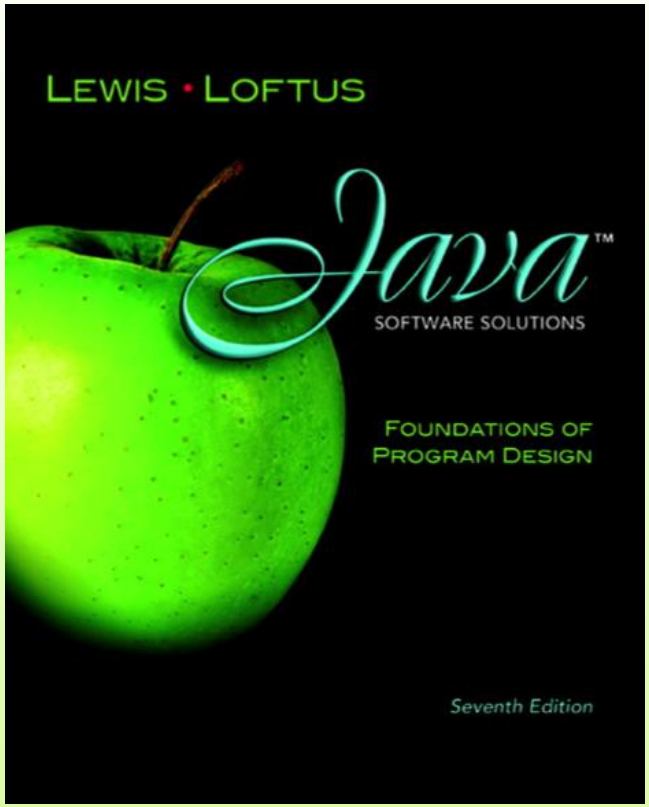
Week 9 & 10
Polymorphism



Java Software Solutions

Foundations of Program Design

Seventh Edition

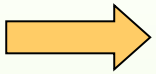
John Lewis
William Loftus

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

Outline



Late Binding

Polymorphism via Inheritance

Polymorphism via Interfaces

Sorting

Searching

Binding

- Consider the following method invocation:

```
obj.doIt();
```

- At some point, this invocation is *bound* to the definition of the method that it invokes

- , Java defers method binding until run time -- this is called *dynamic binding* or *late binding*

Polymorphism

- The term *polymorphism* literally means "having many forms"
- A *polymorphic reference* is a variable that can refer to different types of objects at different points in time
- The method called through a polymorphic reference can change from one invocation to the next
- All object references in Java are potentially polymorphic

Polymorphism

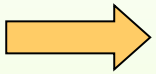
- Suppose we create the following reference variable:

```
Occupation job;
```

- This reference can point to an `Occupation` object, or to any object of any compatible type
- This compatibility can be established using inheritance or using interfaces
- Careful use of polymorphic references can lead to elegant, robust software designs

Outline

Late Binding



Polymorphism via Inheritance

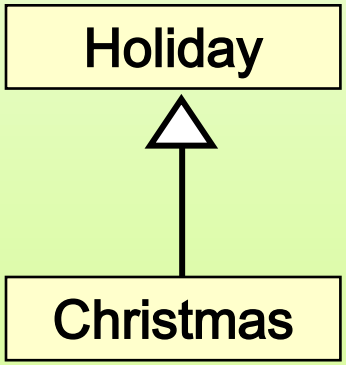
Polymorphism via Interfaces

Sorting

Searching

References and Inheritance

- An object reference can refer to an object of any class related to it by inheritance
- For example, if `Holiday` is the superclass of `Christmas`, then a `Holiday` reference could be used to refer to a `Christmas` object




```
Holiday day;
day = new Christmas();
```

References and Inheritance

- These type compatibility rules are just an extension of the is-a relationship established by inheritance
- Assigning a `Christmas` object to a `Holiday` reference is fine because Christmas is-a holiday
- Assigning a child object to a parent reference can be performed by simple assignment
- Assigning an parent object to a child reference can be done also, but must be done with a cast (now this is confusing??)
- After all, Christmas is a holiday but not all holidays are Christmas

Polymorphism via Inheritance

- Now suppose the `Holiday` class has a method called `celebrate`, and `Christmas` overrides it
- What method is invoked by the following?

```
day.celebrate();
```

- The type of the object being referenced, not the reference type, determines which method is invoked
- If `day` refers to a `Holiday` object, it invokes the `Holiday` version of `celebrate`; if it refers to a `Christmas` object, it invokes that version

Polymorphism via Inheritance

- Note that the compiler restricts invocations based on the type of the reference
- So if `Christmas` had a method called `getTree` that `Holiday` didn't have, the following would cause a compiler error:

```
day.getTree();  // compiler error
```

- Remember, the compiler doesn't "know" which type of holiday is being referenced
- A cast can be used to allow the call:

```
((Christmas)day).getTree();
```

Quick Check

If `MusicPlayer` is the parent of `CDPlayer`, are the following assignments valid?

```
MusicPlayer mplayer = new CDPlayer();
```

```
CDPlayer cdplayer = new MusicPlayer();
```

Quick Check

If `MusicPlayer` is the parent of `CDPlayer`, are the following assignments valid?

```
MusicPlayer mplayer = new CDPlayer();
```

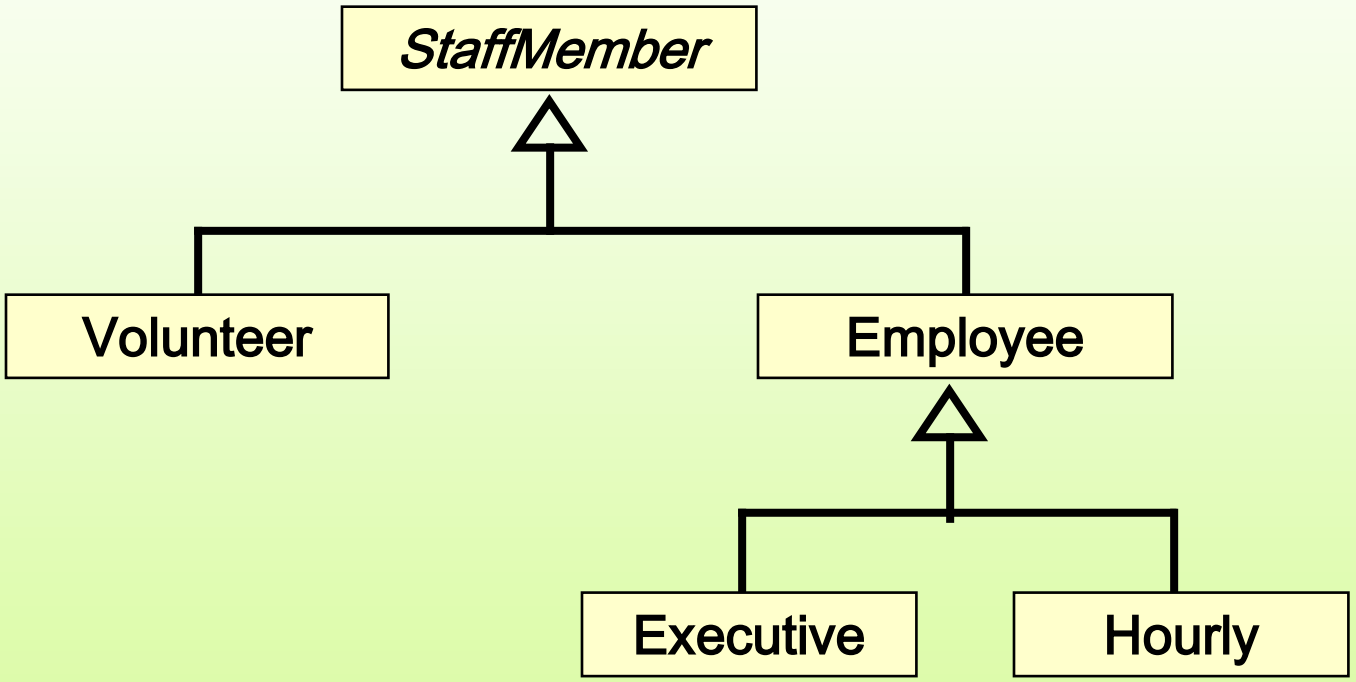
Yes, because a `CDPlayer` is-a `MusicPlayer`

```
CDPlayer cdplayer = new MusicPlayer();
```

No, you'd have to use a cast (and you shouldn't knowingly assign a super class object to a subclass reference)

Polymorphism via Inheritance

- Consider the following class hierarchy:



Polymorphism via Inheritance

- Let's look at an example that pays a set of diverse employees using a polymorphic method

- See `Firm.java`
- See `Staff.java`
- See `StaffMember.java`
- See `Volunteer.java`
- See `Employee.java`
- See `Executive.java`
- See `Hourly.java`

```
//********************************************************************
//  StaffMember.java       Author: Lewis/Loftus
//
//  Represents a generic staff member.
//********************************************************************

abstract public class StaffMember
{
   protected String name;
   protected String address;
   protected String phone;

   //-----------------------------------------------------------------
   //  Constructor: Sets up this staff member using the specified
   //  information.
   //-----------------------------------------------------------------
   public StaffMember (String eName, String eAddress, String ePhone)
   {
      name = eName;
      address = eAddress;
      phone = ePhone;
   }
```

continue

continue

```
   //-----------------------------------------------------------------
   //  Returns a string including the basic employee information.
   //-----------------------------------------------------------------
   public String toString()
   {
      String result = "Name: " + name + "\n";

      result += "Address: " + address + "\n";
      result += "Phone: " + phone;

      return result;
   }

   //-----------------------------------------------------------------
   //  Derived classes must define the pay method for each type of
   //  employee.
   //-----------------------------------------------------------------
   public abstract double pay();
}
```

```
//********************************************************************
//  Volunteer.java       Author: Lewis/Loftus
//
//  Represents a staff member that works as a volunteer.
//********************************************************************

public class Volunteer extends StaffMember
{
   //-----------------------------------------------------------------
   //  Constructor: Sets up this volunteer using the specified
   //  information.
   //-----------------------------------------------------------------
   public Volunteer (String eName, String eAddress, String ePhone)
   {
      super (eName, eAddress, ePhone);
   }

   //-----------------------------------------------------------------
   //  Returns a zero pay value for this volunteer.
   //-----------------------------------------------------------------
   public double pay()
   {
      return 0.0;
   }
}
```

```
//********************************************************************
//  Employee.java       Author: Lewis/Loftus
//
//  Represents a general paid employee.
//********************************************************************

public class Employee extends StaffMember
{
   protected String socialSecurityNumber;
   protected double payRate;

   //-----------------------------------------------------------------
   //  Constructor: Sets up this employee with the specified
   //  information.
   //-----------------------------------------------------------------
   public Employee (String eName, String eAddress, String ePhone,
                    String socSecNumber, double rate)
   {
      super (eName, eAddress, ePhone);

      socialSecurityNumber = socSecNumber;
      payRate = rate;
   }
```

continue

continue

```
    //-----------------------------------------------------------------
    //  Returns information about an employee as a string.
    //-----------------------------------------------------------------
    public String toString()
    {
       String result = super.toString();

       result += "\nSocial Security Number: " + socialSecurityNumber;

       return result;
    }

    //-----------------------------------------------------------------
    //  Returns the pay rate for this employee.
    //-----------------------------------------------------------------
    public double pay()
    {
       return payRate;
    }
}
```

```
//********************************************************************
//  Executive.java       Author: Lewis/Loftus
//
//  Represents an executive staff member, who can earn a bonus.
//********************************************************************

public class Executive extends Employee
{
   private double bonus;

   //-----------------------------------------------------------------
   //  Constructor: Sets up this executive with the specified
   //  information.
   //-----------------------------------------------------------------
   public Executive (String eName, String eAddress, String ePhone,
                     String socSecNumber, double rate)
   {
      super (eName, eAddress, ePhone, socSecNumber, rate);

      bonus = 0;  // bonus has yet to be awarded
   }
```

continue

continue

```
   //-----------------------------------------------------------------
   //  Awards the specified bonus to this executive.
   //-----------------------------------------------------------------
   public void awardBonus (double execBonus)
   {
      bonus = execBonus;
   }

   //-----------------------------------------------------------------
   //  Computes and returns the pay for an executive, which is the
   //  regular employee payment plus a one-time bonus.
   //-----------------------------------------------------------------
   public double pay()
   {
      double payment = super.pay() + bonus;

      bonus = 0;

      return payment;
   }
}
```

```
//********************************************************************
//  Hourly.java       Author: Lewis/Loftus
//
//  Represents an employee that gets paid by the hour.
//********************************************************************

public class Hourly extends Employee
{
   private int hoursWorked;

   //-----------------------------------------------------------------
   //  Constructor: Sets up this hourly employee using the specified
   //  information.
   //-----------------------------------------------------------------
   public Hourly (String eName, String eAddress, String ePhone,
                  String socSecNumber, double rate)
   {
      super (eName, eAddress, ePhone, socSecNumber, rate);

      hoursWorked = 0;
   }
```

continue

continue

```
    //-----------------------------------------------------------------
    //  Adds the specified number of hours to this employee's
    //  accumulated hours.
    //-----------------------------------------------------------------
    public void addHours (int moreHours)
    {
       hoursWorked += moreHours;
    }

    //-----------------------------------------------------------------
    //  Computes and returns the pay for this hourly employee.
    //-----------------------------------------------------------------
    public double pay()
    {
       double payment = payRate * hoursWorked;

       hoursWorked = 0;

       return payment;
    }
```

continue

continue

```
   //-----------------------------------------------------------------
   //  Returns information about this hourly employee as a string.
   //-----------------------------------------------------------------
   public String toString()
   {
      String result = super.toString();

      result += "\nCurrent hours: " + hoursWorked;

      return result;
   }
}
```

```
//********************************************************************
//  Staff.java       Author: Lewis/Loftus
//
//  Represents the personnel staff of a particular business.
//********************************************************************

public class Staff
{
   private StaffMember[] staffList;

   //-----------------------------------------------------------------
   //  Constructor: Sets up the list of staff members.
   //-----------------------------------------------------------------
   public Staff ()
   {
      staffList = new StaffMember[6];
```

continue

continue

```
      staffList[0] = new Executive ("Sam", "123 Main Line",
         "555-0469", "123-45-6789", 2423.07);

      staffList[1] = new Employee ("Carla", "456 Off Line",
         "555-0101", "987-65-4321", 1246.15);
      staffList[2] = new Employee ("Woody", "789 Off Rocker",
         "555-0000", "010-20-3040", 1169.23);

      staffList[3] = new Hourly ("Diane", "678 Fifth Ave.",
         "555-0690", "958-47-3625", 10.55);

      staffList[4] = new Volunteer ("Norm", "987 Suds Blvd.",
         "555-8374");
      staffList[5] = new Volunteer ("Cliff", "321 Duds Lane",
         "555-7282");

      ((Executive)staffList[0]).awardBonus (500.00);

      ((Hourly)staffList[3]).addHours (40);
   }
```

continue

continue

```
   //-----------------------------------------------------------------
   //  Pays all staff members.
   //-----------------------------------------------------------------
   public void payday ()
   {
      double amount;

      for (int count=0; count < staffList.length; count++)
      {
         System.out.println (staffList[count]);

         amount = staffList[count].pay();  // polymorphic

         if (amount == 0.0)
            System.out.println ("Thanks!");
         else
            System.out.println ("Paid: " + amount);

         System.out.println ("-----------------------------------");
      }
   }
}
```

```
//********************************************************************
//  Firm.java       Author: Lewis/Loftus
//
//  Demonstrates polymorphism via inheritance.
//********************************************************************

public class Firm
{
   //-----------------------------------------------------------------
   //  Creates a staff of employees for a firm and pays them.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      Staff personnel = new Staff();

      personnel.payday();
   }
}
```

Output

```
Name: Sam
Address: 123 Main Line
Phone: 555-0469
Social Security Number: 123-45-6789
Paid: 2923.07
-----------------------------------
Name: Carla
Address: 456 Off Line
Phone: 555-0101
Social Security Number: 987-65-4321
Paid: 1246.15
-----------------------------------
Name: Woody
Address: 789 Off Rocker
Phone: 555-0000
Social Security Number: 010-20-3040
Paid: 1169.23
-----------------------------------
```

```
**
Lo

in
**


--
s
--
[]

);
```

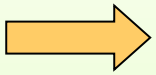
Output (continued)

```
Name: Diane
Address: 678 Fifth Ave.
Phone: 555-0690
Social Security Number: 958-47-3625
Current hours: 40
Paid: 422.0
-----------------------------------
Name: Norm
Address: 987 Suds Blvd.
Phone: 555-8374
Thanks!
-----------------------------------
Name: Cliff
Address: 321 Duds Lane
Phone: 555-7282
Thanks!
-----------------------------------
```

Outline

Late Binding

Polymorphism via Inheritance



Polymorphism via Interfaces

Sorting

Searching

Polymorphism via Interfaces

- Interfaces can be used to set up polymorphic references as well
- Suppose we declare an interface called `Speaker` as follows:

```
public interface Speaker
{
    public void speak();
    public void announce (String str);
}
```

Polymorphism via Interfaces

- An interface name can be used as the type of an object reference variable:

```
Speaker current;
```

- The `current` reference can be used to point to any object of any class that implements the `Speaker` interface

- The version of `speak` invoked by the following line depends on the type of object that `current` is referencing:

```
current.speak();
```

Polymorphism via Interfaces

- Now suppose two classes, `Philosopher` and `Dog`, both implement the `Speaker` interface, providing distinct versions of the `speak` method
- In the following code, the first call to `speak` invokes one version and the second invokes another:

```
Speaker guest = new Philospher();
guest.speak();
guest = new Dog();
guest.speak();
```

Polymorphism via Interfaces

- As with class reference types, the compiler will restrict invocations to methods in the interface
- For example, even if `Philosopher` also had a method called `pontificate`, the following would still cause a compiler error:

```
Speaker special = new Philospher();
special.pontificate();  // compiler error
```

- Remember, the compiler bases its rulings on the type of the reference

Quick Check

Would the following statements be valid?

```
Speaker first = new Dog();
Philosopher second = new Philosopher();
second.pontificate();
first = second;
```

Quick Check

Would the following statements be valid?

```
Speaker first = new Dog();
Philosopher second = new Philosopher();
second.pontificate();
first = second;
```

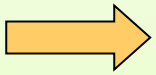
Yes, all assignments and method calls are valid as written

Outline

Late Binding

Polymorphism via Inheritance

Polymorphism via Interfaces



Sorting

Searching

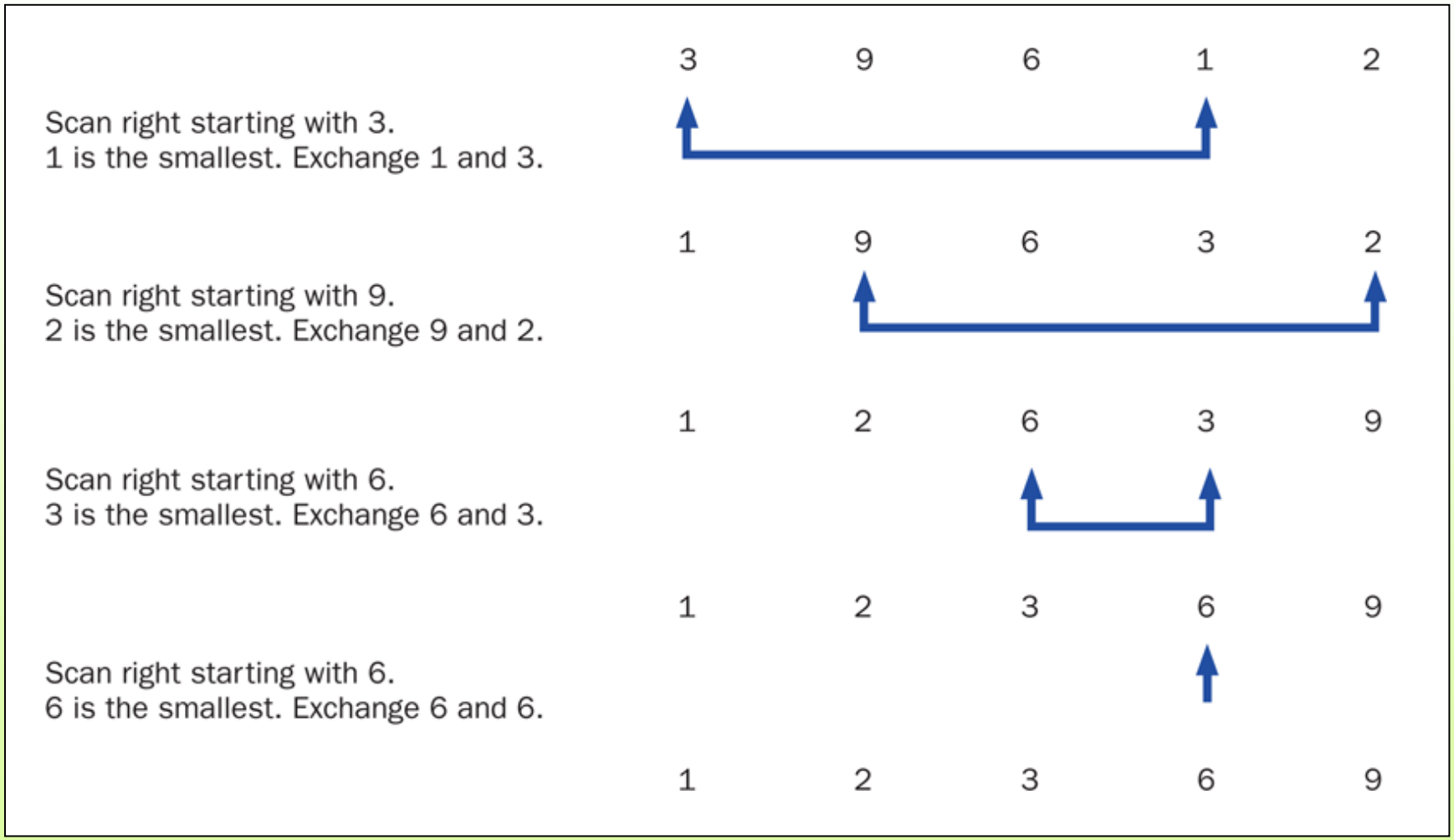
Sorting

- *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;
```

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

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`

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;
   }
}
```

```
//*******************************************************************
//  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

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

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

continue

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

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

```
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");

      Sorting.selectionSort(friends);

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

```
public class PhoneList
{
   //-----------------------------------------------------------------
   //  Creates an array of Contact objects, sorts them, then prints
   //  them.
   //-----------------------------------------------------------------
   public static
   {
      Contact[] 

      friends[0]                                        "610-555-7384");
      friends[1]                                        , "215-555-3827");
      friends[2]                                        "733-555-2969");
      friends[3]                                        "663-555-3984");
      friends[4]                                         "464-555-3489");
      friends[5]                                        , "322-555-2284");
      friends[6]                                        , "804-555-9066");
      friends[7]                                        , "243-555-2837");

      Sorting.selectionSort(friends);

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

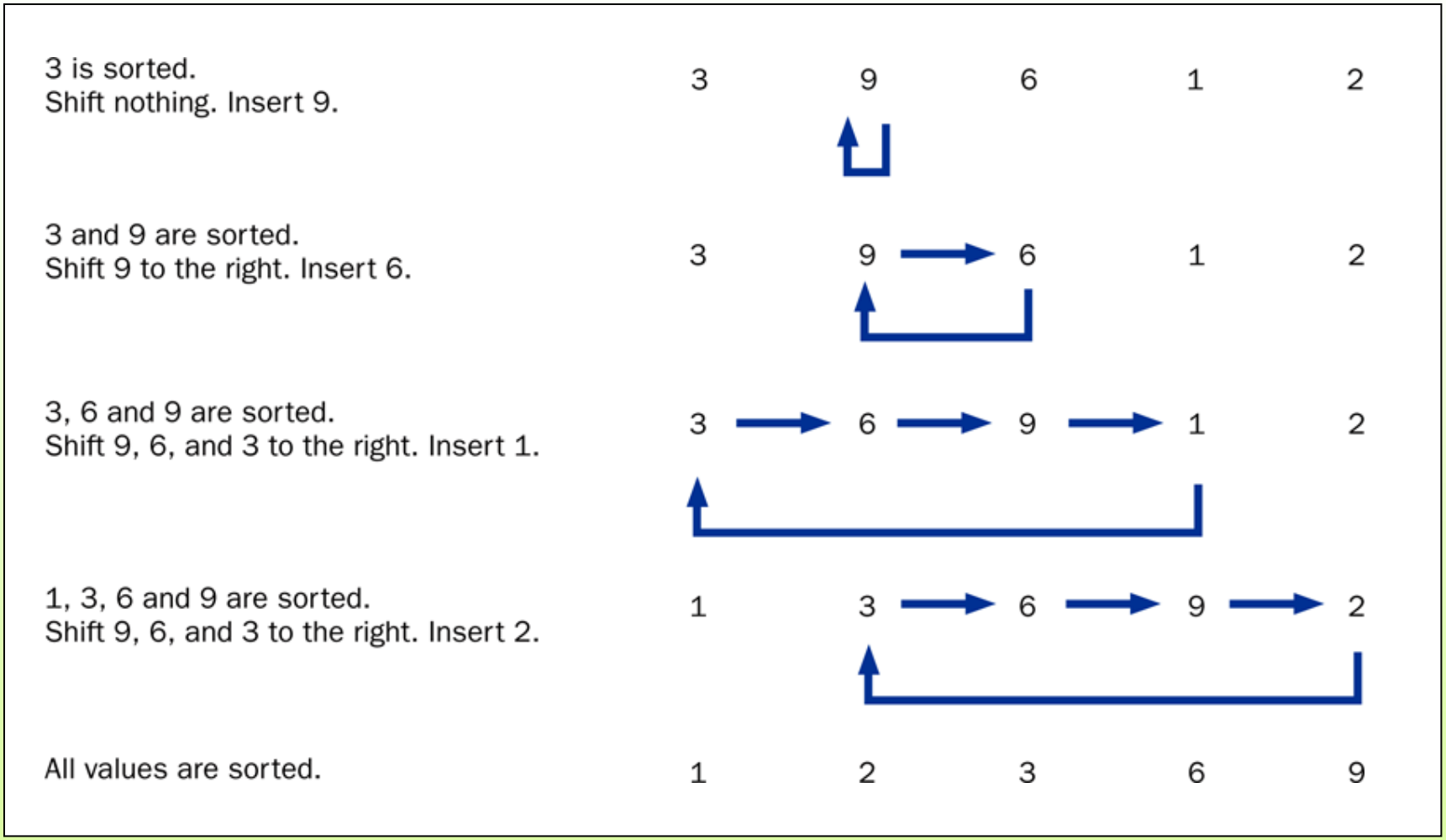
Output

```
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
```

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



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* $n \log_2 n$

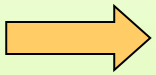
Outline

Late Binding

Polymorphism via Inheritance

Polymorphism via Interfaces

Sorting



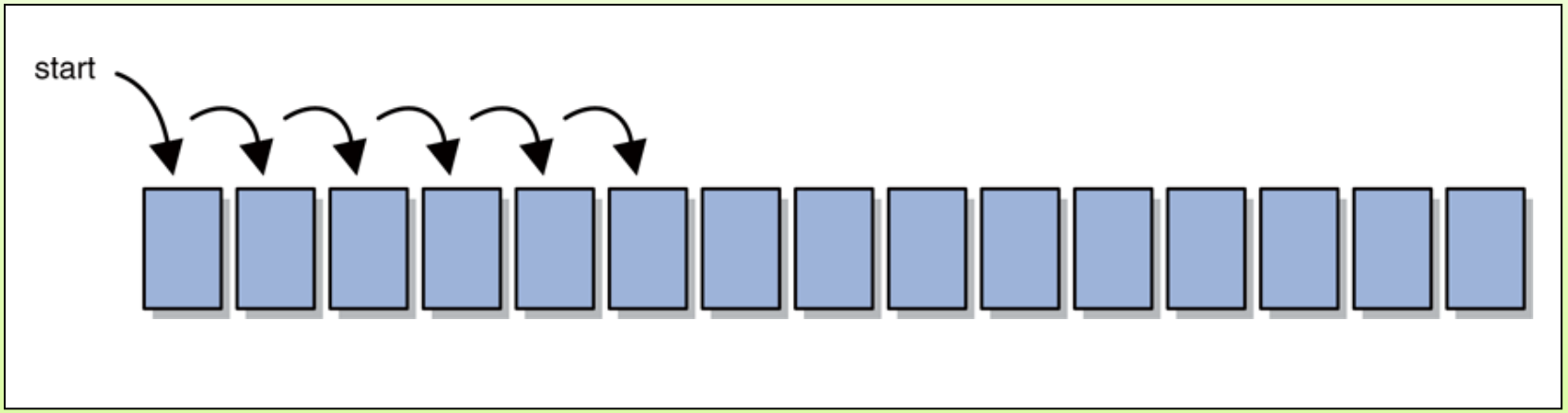
Searching

Searching

- *Searching* is the process of finding a *target element* within a group of items called the *search pool*
- The target may or may not be in the search pool
- We want to perform the search efficiently, minimizing the number of comparisons
- Let's look at two classic searching approaches: linear search and binary search
- As we did with sorting, we'll implement the searches with polymorphic `Comparable` parameters

Linear Search

- A linear search begins at one end of a list and examines each element in turn
- Eventually, either the item is found or the end of the list is encountered

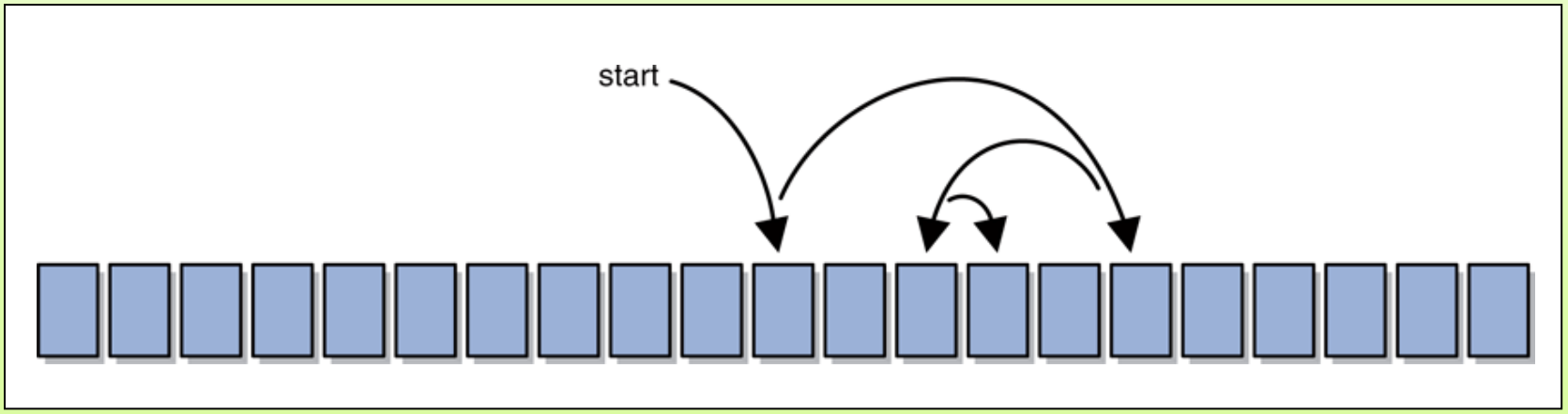


Binary Search

- A *binary search* assumes the list of items in the search pool is sorted
- It eliminates a large part of the search pool with a single comparison
- A binary search first examines the middle element of the list -- if it matches the target, the search is over
- If it doesn't, only one half of the remaining elements need be searched
- Since they are sorted, the target can only be in one half of the other

Binary Search

- The process continues by comparing the middle element of the remaining *viable candidates*
- Each comparison eliminates approximately half of the remaining data
- Eventually, the target is found or the data is exhausted



Searching

- The search methods are implemented as static methods in the `Searching` class

- See `PhoneList2.java`
- See `Searching.java`

The `linearSearch` method in the `Searching` class:

```
//-----------------------------------------------------------------
//  Searches the specified array of objects for the target using
//  a linear search. Returns a reference to the target object from
//  the array if found, and null otherwise.
//-----------------------------------------------------------------
public static Comparable linearSearch (Comparable[] list,
                                       Comparable target)
{
   int index = 0;
   boolean found = false;

   while (!found && index < list.length)
   {
      if (list[index].equals(target))
         found = true;
      else
         index++;
   }

   if (found)
      return list[index];
   else
      return null;
}
```

The `binarySearch` method in the `Searching` class:

```
//-----------------------------------------------------------------
//  Searches the specified array of objects for the target using
//  a binary search. Assumes the array is already sorted in
//  ascending order when it is passed in. Returns a reference to
//  the target object from the array if found, and null otherwise.
//-----------------------------------------------------------------
public static Comparable binarySearch (Comparable[] list,
                                       Comparable target)
{
   int min=0, max=list.length, mid=0;
   boolean found = false;

   while (!found && min <= max)
   {
      mid = (min+max) / 2;
      if (list[mid].equals(target))
         found = true;
      else
         if (target.compareTo(list[mid]) < 0)
            max = mid-1;
         else
            min = mid+1;
   }
```

continue

continue

```
        if (found)
            return list[mid];
        else
            return null;
    }
```

```
//********************************************************************
//  PhoneList2.java       Author: Lewis/Loftus
//
//  Driver for testing searching algorithms.
//********************************************************************

public class PhoneList2
{
   //-----------------------------------------------------------------
   //  Creates an array of Contact objects, sorts them, then prints
   //  them.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      Contact test, found;
      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

```
        test = new Contact ("Frank", "Phelps", "");
        found = (Contact) Searching.linearSearch(friends, test);
        if (found != null)
           System.out.println ("Found: " + found);
        else
           System.out.println ("The contact was not found.");
        System.out.println ();

        Sorting.selectionSort(friends);

        test = new Contact ("Mario", "Guzman", "");
        found = (Contact) Searching.binarySearch(friends, test);
        if (found != null)
           System.out.println ("Found: " + found);
        else
           System.out.println ("The contact was not found.");
    }
}
```

continue

```
        test = new
        found = (C                                    s, test);
        if (found
           System.out.println ("Found: " + found);
        else
           System.out.println ("The contact was not found.");
        System.out.println ();

        Sorting.selectionSort(friends);

        test = new Contact ("Mario", "Guzman", "");
        found = (Contact) Searching.binarySearch(friends, test);
        if (found != null)
           System.out.println ("Found: " + found);
        else
           System.out.println ("The contact was not found.");
    }
}
```

Output

```
Found: Phelps, Frank  322-555-2284

Found: Guzman, Mario  804-555-9066
```

Summary

- Chapter 10 has focused 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