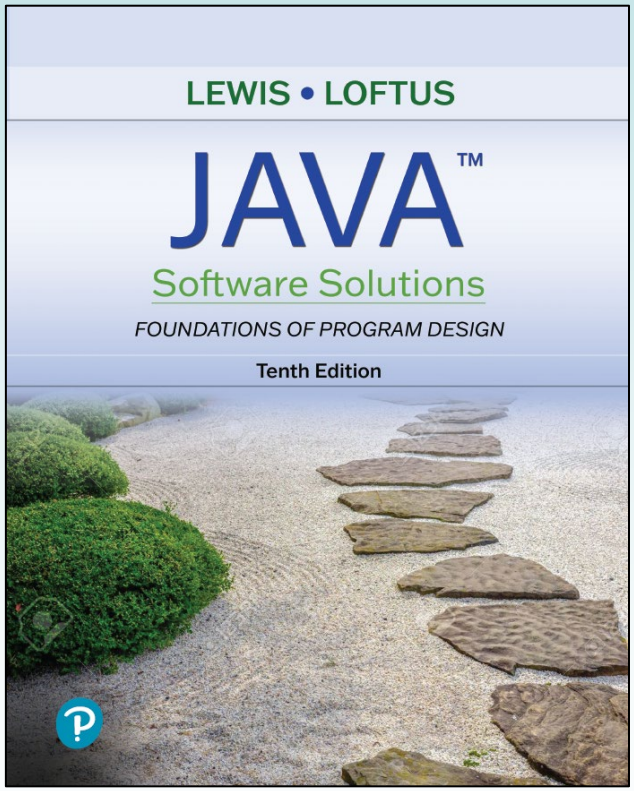
Chapter 10
Polymorphism



Java Software Solutions
Foundations of Program Design
10th Edition

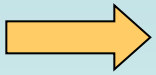
(edits 2/25/2019; 3/7/2019; 06/01/2024, 08/12/2026)

John Lewis
William Loftus

Polymorphism

- Chapter 10 focuses on:
  - polymorphism and its benefits
  - using inheritance to create polymorphic references
  - using interfaces to create polymorphic references
  - using polymorphism to implement sorting and searching algorithms
  - ☺Property binding
  - ☺additional GUI components

Outline



Late Binding

Polymorphism via Inheritance

Polymorphism via Interfaces

Sorting

Searching

Properties

Sliders

Spinners

Binding

- Consider the following method invocation:
  `obj.doIt();`
- At some point, this invocation is *bound* to the definition of `doIt()`
  - but there might be a `doIt()` in the parent and a different one in a child class
  - `obj` might point to a parent at one point in a program, and to a child at another point
  - Which `doIt()` gets run?

Binding

- `obj.doIt()`
  - Which `doIt()`? parent? child?
- If binding occurs at compile time, then that line of code would call the same method every time
- However, 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 times
- The method called through a polymorphic reference can change from one time to the next
  - `obj.doIt()` may call different methods depending on what type of object `obj` is currently pointing to
- All object references in Java are potentially polymorphic

Polymorphism

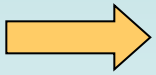
- Suppose we create the following reference variable:

```
Occupation job;
```

- This reference can point to an `Occupation` object, or to any object of any compatible type
- Objects are compatible by inheritance or by interfaces

Outline

Late Binding



Polymorphism via Inheritance

Polymorphism via Interfaces

Sorting

Searching

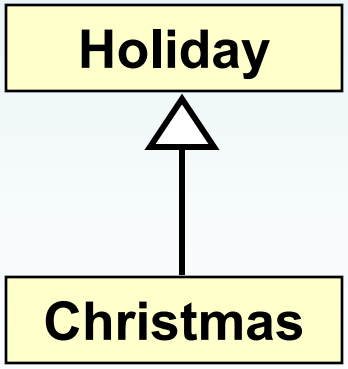
Properties

Sliders

Spinners

References via Inheritance

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




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

References via Inheritance

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

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

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

```
class Holiday
{
  void celebrate()
  {
    System.out.println("Happy Holiday");
  }
}

class Christmas extends Holiday
{
  void celebrate()
  {
    System.out.println("Merry Christmas");
  }
}

class HolidayTester
{
  public static void main (String[] args)
  {
    Holiday day; // polymorphic reference
    day = new Holiday();
    day.celebrate();

    day = new Christmas();
    day.celebrate();
  }
}
```

```
PS C:\Users\OneDrive - CCSU\AA151> javac HolidayTester.java
PS C:\Users\OneDrive - CCSU\AA151> java HolidayTester
Happy Holiday
Merry Christmas
PS C:\Users\OneDrive - CCSU\AA151>
```

Polymorphism via Inheritance

- The compiler restricts method names to those had by the type of the reference variable
- So if `Christmas` had a method called `getTree` that `Holiday` didn't have, the following would cause a compiler error

```
Holiday day = new Holiday();
day.getTree();  // compiler error
```

- The compiler can’t predict which type of holiday is being referenced since that happens at run-time
- 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`

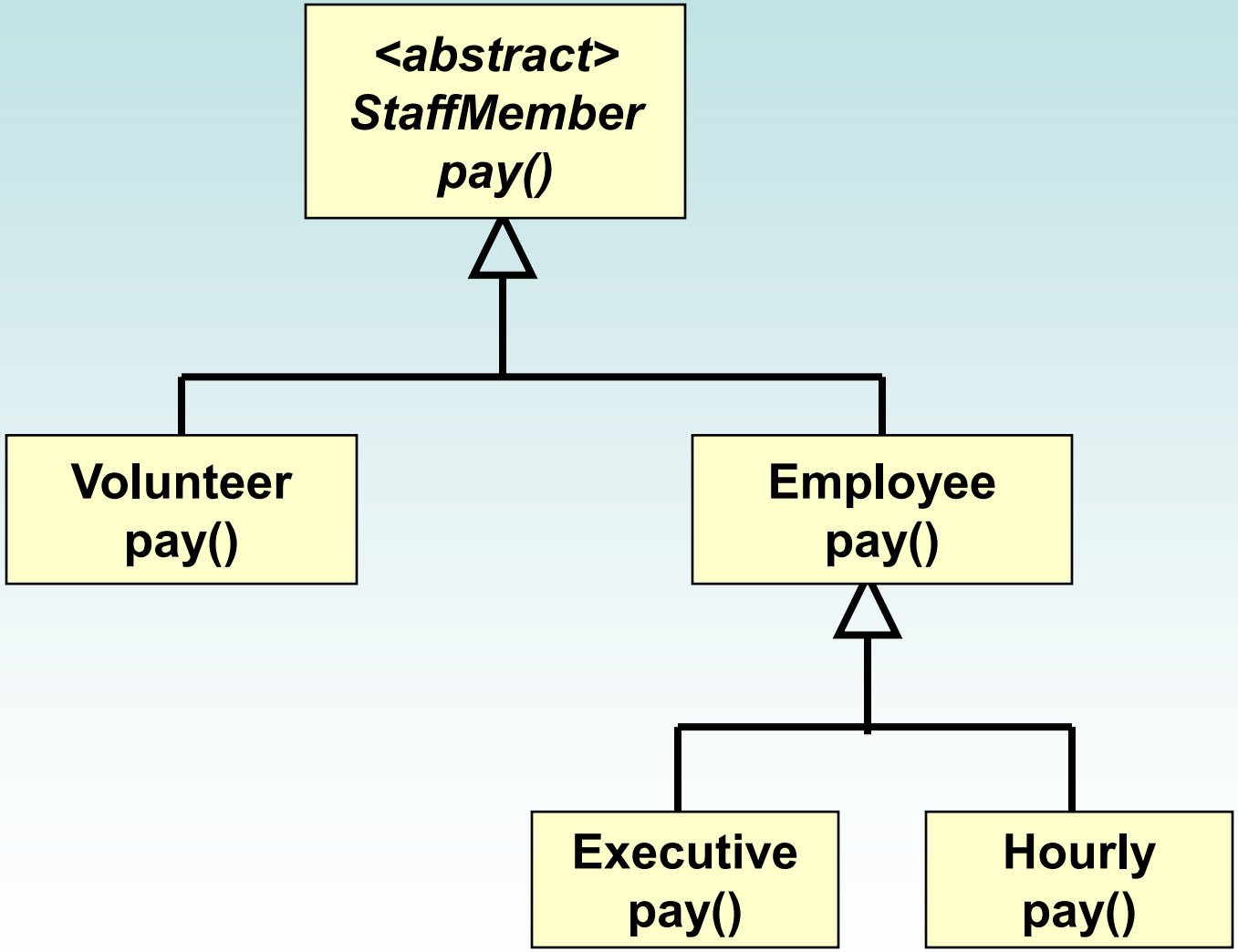
```
CDPlayer cdplayer = new MusicPlayer();
```

No, you'd have to use a cast (and you shouldn't usually assign a reference to a parent object to a child reference variable)

Polymorphism via Inheritance

- Let's look at an example that pays a set of employees of various types

Polymorphism via Inheritance



```
//********************************************************************
//  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. Weird: even though you can't construct an object
   //  from an abstract class, it does have a constructor, which
   //  its children access with super.
   //-----------------------------------------------------------------
   public StaffMember(String eName, String eAddress, String ePhone)
   {
      name = eName;
      address = eAddress;
      phone = ePhone;
   }
```

continue StaffMember

```
   //-----------------------------------------------------------------
   //  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();
}
```

```java
//********************************************************************
//  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 Employee

```
    //-----------------------------------------------------------------
    //  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 Executive

```
   //-----------------------------------------------------------------
   //  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 Hourly

```
    //-----------------------------------------------------------------
    //  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 Hourly

```
    //-----------------------------------------------------------------
    //  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();
   }
}
```

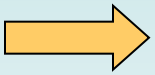
Output

```
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
-----------------------------------
```

Output

```
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
- ☺ Properties
- ☺ Sliders
- ☺ Spinners

Polymorphism via Interfaces

- Interfaces can be used to set up polymorphic references
- 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
  - `Philosopher` and `Dog` are not related by inheritance
- In the following, the first call to `speak` invokes one version and the second invokes another:

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

- A type cast can tell the compiler what type the reference points to

```
Speaker special = new Philosopher();
((Philosopher)special).pontificate();
```

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.

However,

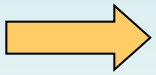
```
first.pontificate(); // would fail
```

Outline

Late Binding

Polymorphism via Inheritance

Polymorphism via Interfaces



Sorting

Searching

Properties

Sliders

Spinners

Algorithm

- An algorithm is a precise step-by-step description of a method used to perform a computation.
  - An algorithm precisely specifies the input conditions
  - If the input conditions are met, the algorithm is guaranteed to correctly perform the computation in a finite amount of time
- Algorithms and Data Structures are the building blocks of programs

Algorithm Efficiency

- There usually are many algorithms that solve a given problem
- Algorithms differ in efficiency (how long they take)
  - For example: searching for a library book by looking randomly on the shelves vs. using the call number
- Algorithms differ in complexity (how complicated the code)

Programs

- Efficient algorithms are hard to invent
- Successful professional computer scientists might invent one algorithm during their who career—if they are lucky
- Most of us never invent a novel algorithm. What we do instead is learn to reduce … problems to previously solved problems

  -- John V. Guttag, “Introduction to Computation”

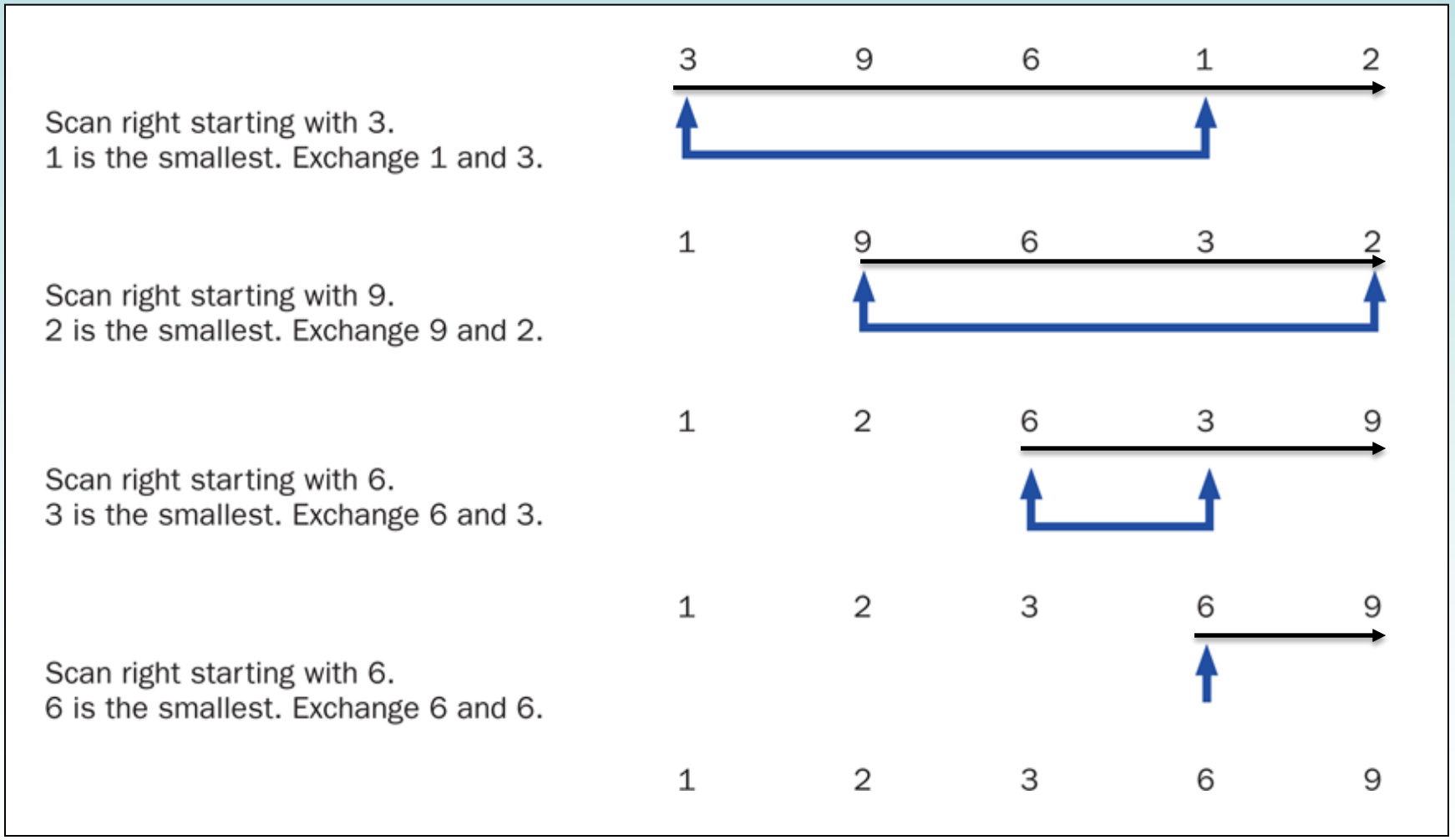
Sorting

- *Sorting* is the process of arranging a list of items into a particular order
  - sort test scores into 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 smallest value in the rest of 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

The Comparable Interface

- Any class can implement **`Comparable`** to provide a mechanism for comparing objects of similar type

```
if ( obj1.compareTo(obj2) < 0 )
   System.out.println ( "obj1 is less than obj2" );
```

- The value returned from `compareTo` should be
  - negative if `obj1` is less that `obj2`,
  - 0 if they are equal,
  - and positive if `obj1` is greater than `obj2`
- It's up to the programmer to determine what makes one object less than another

Using 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);
    }
}
```

continue

```
        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
```

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

public class Contact implements Comparable<Contact>
{
   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 Contact

```
//-----------------------------------------------------------------
//  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 Contact

```
//-----------------------------------------------------------------
//  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 Contact

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

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

end Contact

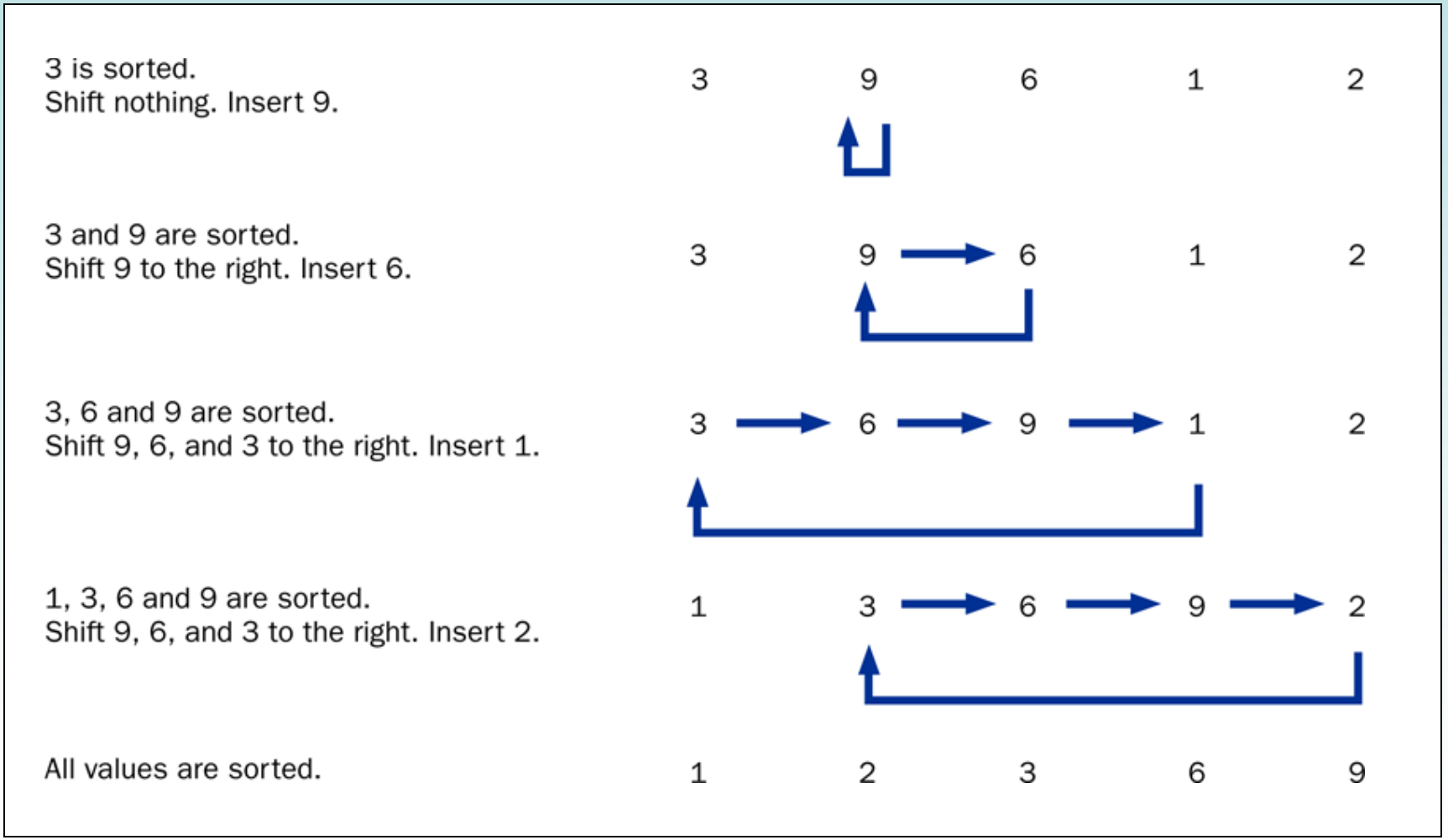
The `selectionSort` method in the `Sorting<T>` class:

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

   for (int index = 0; index < list.length - 1; index++)
   {
      // Scan the remaining list looking for its minimum
      min = index;
      for (int scan = index + 1; scan < list.length; scan++)
         if (list[scan].compareTo((T)list[min]) < 0)
            min = scan;

      // Swap values to put minimum at index
      temp = list[min];
      list[min] = list[index];
      list[index] = temp;
   }
}
```

Second sorting method: Insertion Sort

- The strategy of Insertion Sort:
  - pick an 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 `insertionSort` method in the `Sorting<T>` class:

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

      //  Shift larger values to the right
      while (position > 0 && key.compareTo((T)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
  - Often this is abbreviated O(n^2)
- Other sorts are more efficient: *order* $(n \log_2 n)$

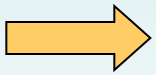
Outline

Late Binding

Polymorphism via Inheritance

Polymorphism via Interfaces

Sorting



Searching

Properties

Sliders

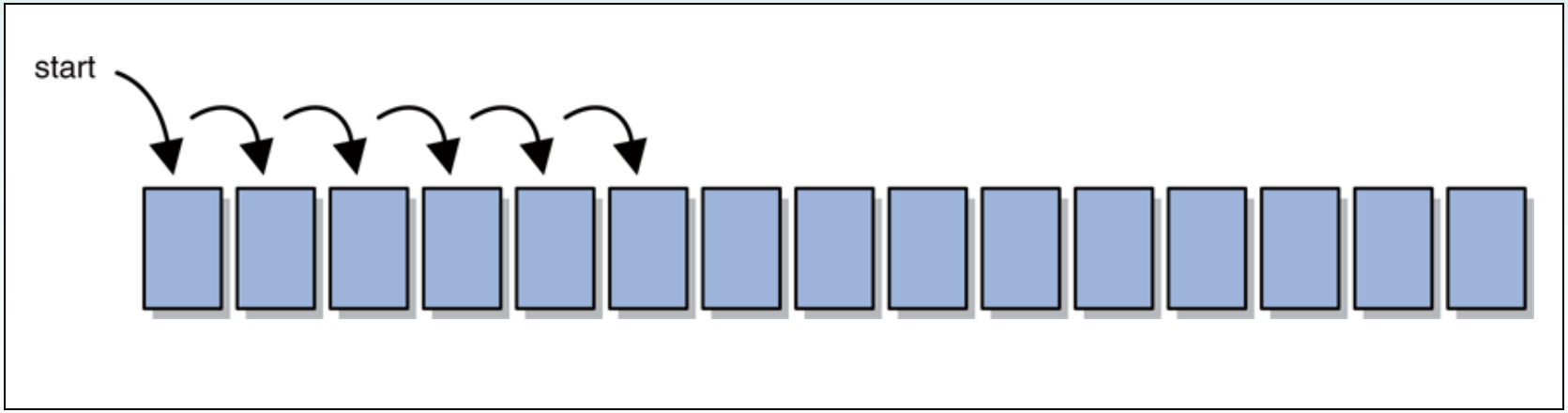
Spinners

Searching

- *Searching* is the process of finding a *target element* within a list of items.
- The target may or may not be in the list
- To perform the search efficiently, minimize the number of comparisons
- Two classic searching approaches: linear search and binary search
- We'll implement the searches with `Comparable` objects

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

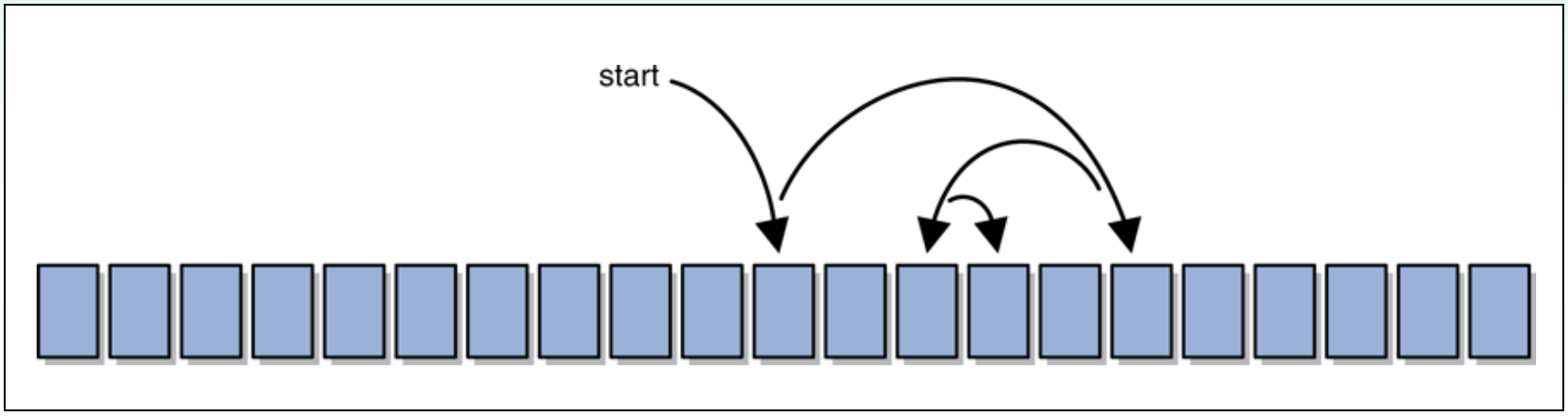


Binary Search

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

Binary Search

- Continue 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



Comparing Searching Methods

- Approximately n number of comparisons are made with linear seach with a list of size n (worst case).
- Approximately $log_2 n$ number of comparisons are made with binary search to search a list of size n (worst case).
  - With 1024 things to search:
    - linear search might take 1024 comparisons
    - binary search might take 10 comparisons

1024/2 == 512, 512/2 == 256, 256/2 == 128

128/2 == 64, 64/2 == 32, 32/2 == 16, 16/2 == 8

8/2 == 4, 4/2 == 2, 2/2 == 1

Searching

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

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

```
//********************************************************************
//  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 = n
        found =                                          test);
        if (foun
            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
```

The `linearSearch` method in the `Searching<T>` 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 T linearSearch(T[] list, T 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<T>` 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 Comparable<T> binarySearch(Comparable<T>[] list,
                                  Comparable<T> target)
{
   int min = 0, max = list.length - 1, mid = 0;
   boolean found = false;

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

continue

continue

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

Outline

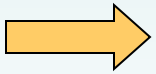
Late Binding

Polymorphism via Inheritance

Polymorphism via Interfaces

Sorting

Searching



Properties

Sliders

Spinners

Properties

- A JavaFX *property* is an object that holds a value
- A property is *observable* – it can be monitored and changed as needed
- So, instead of holding an `int` primitive or an `Integer` object, a JavaFX class might store an `IntegerProperty` object
- Most values of JavaFX classes, such as the width of a `Rectangle`, are stored as properties

Properties

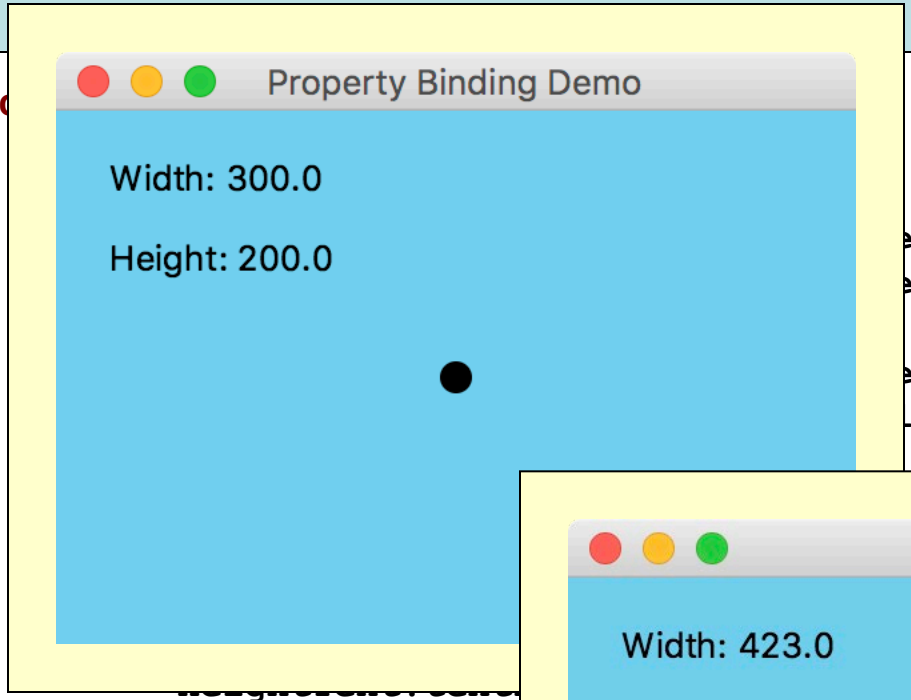
- A key benefit of properties is *property binding*
- A property can be bound to another property so that when the value of one changes, the other is automatically updated
- No explicit event handling is needed
- Let's look at an example that binds the center point of a circle, and the text of a `Text` object, to the size of the scene
- See `PropertyBindingDemo.java`

```
import javafx.application.Application;
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.text.Text;
import javafx.stage.Stage;

//*******************************************************************
//  PropertyBindingDemo.java       Author: Lewis/Loftus
//
//  Demonstrates the ability to bind one property to another.
//*******************************************************************

public class PropertyBindingDemo extends Application
{
    //-----------------------------------------------------------------
    //  Displays the width and height of the scene, as well as a circle
    //  in the center of the scene. The scene is updated using property
    //  bindings as the window is resized.
    //-----------------------------------------------------------------
    public void start(Stage primaryStage)
    {
        Group root = new Group();
        Scene scene = new Scene(root, 300, 200, Color.SKYBLUE);
```

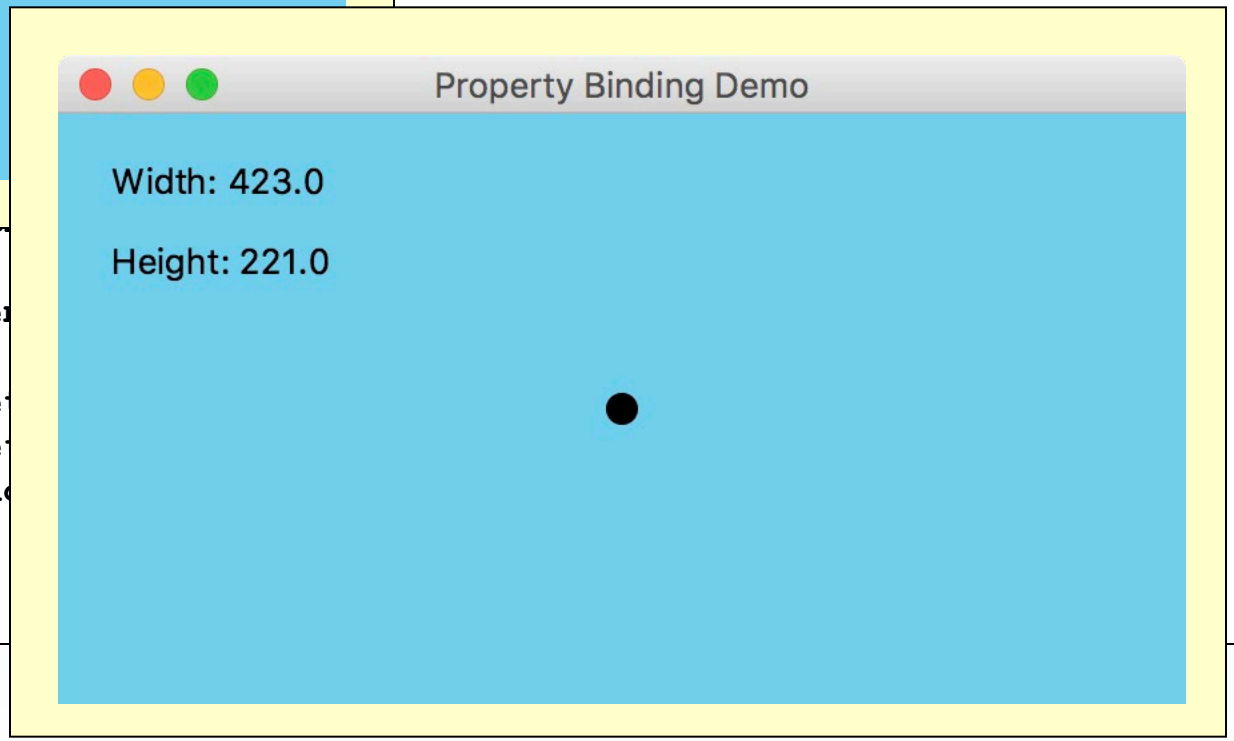
continue

continue

```
        Circle center = new Circle(6);
        center.centerXProperty().bind(scene.widthProperty().divide(2));
        center.centerYProperty().bind(scene.heightProperty().divide(2));

        StringProperty width = new SimpleStringProperty("Width: ");
        StringProperty height = new SimpleStringProperty("Height: ");

        Text widthText = new Text(20, 30, "");
        widthText.textProperty().bind(width.concat(scene.widthProperty()));

        Text heightText = new Text(20, 60, "");
        heightText.textProperty().bind(height.concat(scene.heightProperty()));

        root.getChildren().addAll(center, widthText, heightText);

        primaryStage.setTitle("Property Binding Demo");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```




```
ene.widthProperty().divide(2));
ene.heightProperty().divide(2));

eStringProperty("Width: ");
leStringProperty("Height: ");
```




```
        root.getChildre

        primaryStage.se
        primaryStage.se
        primaryStage.sh
    }
}
```

Change Listeners

- A property can have a *change listener*, similar to an event handler
- Properties have an `addListener` method:

```
myProperty.addListener(this::processChange);
```

- A change listener method accepts three parameters:
  - the `ObservableValue` (the property)
  - the old value
  - the new value

Change Listeners

- The types of the old and new values depend on the type of value the property holds

```
public void processChange(ObservableValue<? extends String> val,
    String oldValue, String newValue)
{
    // whatever
}
```

```
public void processChange(ObservableValue<? extends Integer> val,
    Integer oldValue, Integer newValue)
{
    // whatever
}
```

- See `ChangeListenerDemo.java`

```
import javafx.application.Application;
import javafx.beans.value.ObservableValue;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.text.Text;
import javafx.stage.Stage;

//*******************************************************************
//  ChangeListenerDemo.java       Author: Lewis/Loftus
//
//  Demonstrates the ability to respond to property changes using
//  change listeners. Functionally equivalent to PropertyBindingDemo.
//*******************************************************************

public class ChangeListenerDemo extends Application
{
    private Scene scene;
    private Circle center;
    private Text widthText, heightText;
```

continue

continue

```
    //-----------------------------------------------------------------
    //  Displays the width and height of the scene, as well as a circle
    //  in the center of the scene. The scene is updated using a change
    //  listener as the window is resized.
    //-----------------------------------------------------------------
    public void start(Stage primaryStage)
    {
        Group root = new Group();

        scene = new Scene(root, 300, 200, Color.SKYBLUE);
        scene.widthProperty().addListener(this::processResize);
        scene.heightProperty().addListener(this::processResize);

        center = new Circle(6);
        center.setCenterX(scene.getWidth() / 2);
        center.setCenterY(scene.getHeight() / 2);

        widthText = new Text(20, 30, "Width: " + scene.getWidth());
        heightText = new Text(20, 60, "Height: " + scene.getHeight());

        root.getChildren().addAll(center, widthText, heightText);

        primaryStage.setTitle("Change Listener Demo");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
```

continue

continue

```
    //-----------------------------------------------------------------
    //  Updates the position of the circle and the displayed width and
    //  height when the window is resized.
    //-----------------------------------------------------------------
    public void processResize(ObservableValue<? extends Number> property,
        Object oldValue, Object newValue)
    {
        center.setCenterX(scene.getWidth() / 2);
        center.setCenterY(scene.getHeight() / 2);
        widthText.setText("Width: " + scene.getWidth());
        heightText.setText("Height: " + scene.getHeight());
    }
}
```

Outline

Late Binding

Polymorphism via Inheritance

Polymorphism via Interfaces

Sorting

Searching


Properties



Sliders


Spinners

Sliders

- A *slider* is a GUI control that allows the user to specify a value within a numeric range
- The minimum and maximum values for the slider are set using the `Slider` constructor
- A slider can be oriented horizontally (default) or vertically
- It can also have optional tick marks and labels

Sliders

- You can use the `getValue` method of the slider to obtain its current value
- Or you can use property bindings or a change listener to react dynamically to the movement of a slider knob
- Let's look at an example that uses two sliders to change the shape of an ellipse
- See `EllipseSliders.java`

```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.geometry.Orientation;
import javafx.scene.Scene;
import javafx.scene.control.Slider;
import javafx.scene.layout.BorderPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Ellipse;
import javafx.stage.Stage;

//*******************************************************************
//  EllipseSliders.java       Author: Lewis/Loftus
//
//  Demonstrates the use of slider controls and property binding.
//*******************************************************************

public class EllipseSliders extends Application
{
    private Ellipse ellipse;
    private Slider xSlider, ySlider;

    //-----------------------------------------------------------------
    //  Displays an ellipse with sliders that control the width and
    //  height of the ellipse.
    //-----------------------------------------------------------------
    public void start(Stage primaryStage)
    {
```

continue

continue

```
        ellipse = new Ellipse(250, 150, 150, 75);
        ellipse.setFill(Color.SALMON);

        xSlider = new Slider(0, 200, 150);
        xSlider.setShowTickMarks(true);
        xSlider.setPadding(new Insets(0, 20, 20, 80));

        ellipse.radiusXProperty().bind(xSlider.valueProperty());

        ySlider = new Slider(0, 100, 75);
        ySlider.setOrientation(Orientation.VERTICAL);
        ySlider.setShowTickMarks(true);
        ySlider.setPadding(new Insets(20, 0, 0, 30));

        ellipse.radiusYProperty().bind(ySlider.valueProperty());

        BorderPane pane = new BorderPane();
        pane.setLeft(ySlider);
        pane.setBottom(xSlider);
        pane.setCenter(ellipse);
        pane.setStyle("-fx-background-color: grey");
```

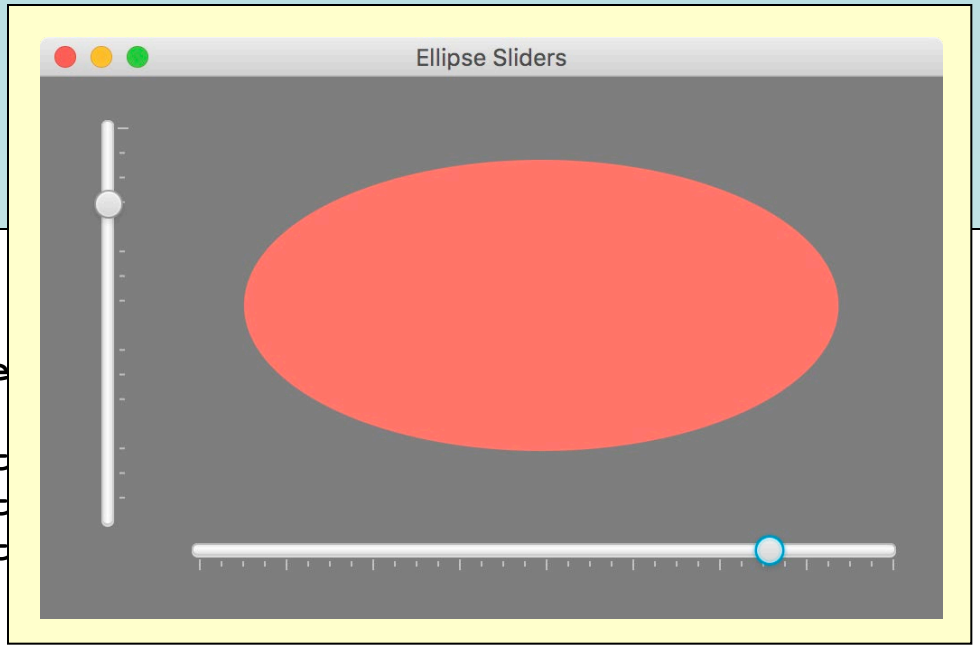
continue

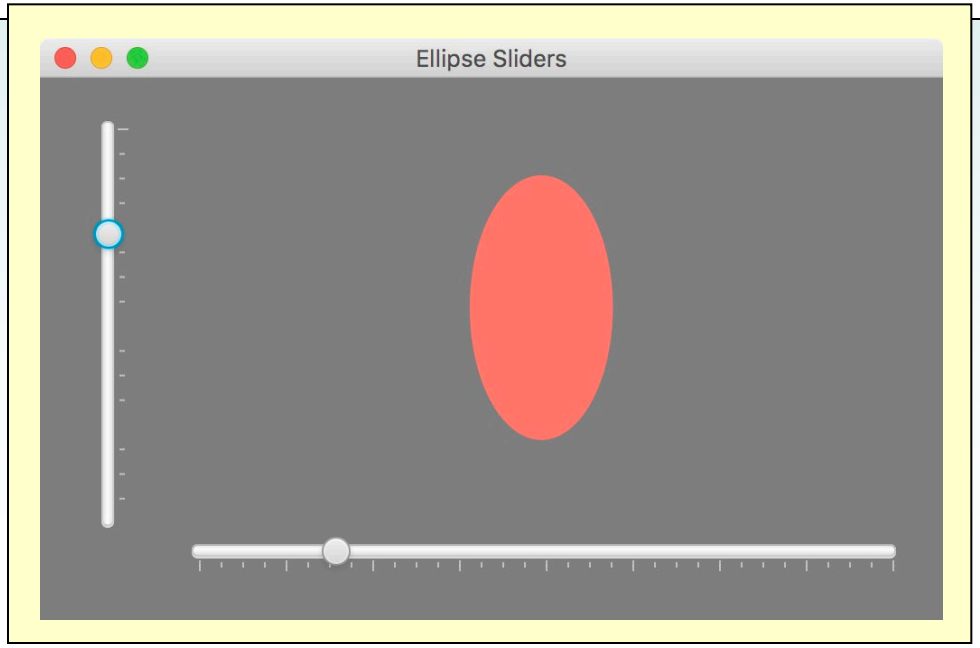
continue

```
        Scene scene = new Scene(pane, 500, 300);

        primaryStage.setTitle("Ellipse Sliders");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

continue

```
        Scene sce

        primarySt
        primarySt
        primarySt
    }
}
```





Outline

Late Binding

Polymorphism via Inheritance

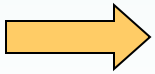
Polymorphism via Interfaces

Sorting

Searching


Properties


Sliders



Spinners

Spinner

- A *spinner* is a GUI control that allows the user to select a value from a list of predefined values
- The current value is shown in a text field with arrow buttons for changing the selection
- Only the current value is ever shown, so other GUI elements are never obscured by a drop-down list
- `Spinner` is a generic class – you specify the type of values the spinner presents

Spinner

- A `SpinnerValueFactory` is used to define the spinner options
- An integer spinner can be created using an `IntegerSpinnerValueFactory`
- A string spinner is created using an `ObservableList`
- See `SpinnerDemo.java`

```
import javafx.application.Application;
import javafx.beans.property.SimpleStringProperty;
import javafx.beans.property.StringProperty;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Spinner;
import javafx.scene.control.SpinnerValueFactory.IntegerSpinnerValueFactory;
import javafx.scene.layout.VBox;
import javafx.scene.text.Font;
import javafx.scene.text.Text;
import javafx.stage.Stage;

//*****************************************************************
//  SpinnerDemo.java       Author: Lewis/Loftus
//
//  Demonstrates the use of spinner controls and property binding.
//*****************************************************************

public class SpinnerDemo extends Application
{
    private Spinner<Integer> intSpinner;
    private Spinner<String> stringSpinner;
    private Text text;
```

continue

continue

```
    //-----------------------------------------------------------------
    //  Presents an integer spinner and a string spinner, updating some
    //  text when either value changes.
    //-----------------------------------------------------------------
    public void start(Stage primaryStage)
    {
        IntegerSpinnerValueFactory svf =
            new IntegerSpinnerValueFactory(1, 10, 5);
        intSpinner = new Spinner<Integer>(svf);

        ObservableList<String> list = FXCollections.observableArrayList();
        list.addAll("Grumpy", "Happy", "Sneezy", "Sleepy", "Dopey",
            "Bashful", "Doc");
        stringSpinner = new Spinner<String>(list);
        stringSpinner.getStyleClass().add(
            Spinner.STYLE_CLASS_SPLIT_ARROWS_VERTICAL);

        StringProperty textString = new SimpleStringProperty("");

        text = new Text();
        text.setFont(new Font("Helvetica", 24));
        text.textProperty().bind(textString.concat(
            intSpinner.valueProperty()).concat(" and ").concat(
            stringSpinner.valueProperty()));
```

continue

continue

```
        VBox pane = new VBox(intSpinner, stringSpinner, text);
        pane.setStyle("-fx-background-color: skyblue");
        pane.setAlignment(Pos.CENTER);
        pane.setSpacing(25);

        Scene scene = new Scene(pane, 300, 250);

        primaryStage.setTitle("Spinner Demo");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

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
  - property binding
  - additional GUI components