11/9/17
polymorphism
in class assignment

Polymorphism

Chapter 9

5TH EDITION

Lewis & Loftus

java

Software Solutions

Foundations of Program Design

Polymorphism

- Polymorphism is an object-oriented concept that allows us to create versatile software designs
- Chapter 9 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

11/14/17
-sorting/
-test 3 on 11/21
-hands on
on test - arrays

Outline

➡ Polymorphic References

Polymorphism via Inheritance

Polymorphism via Interfaces

Sorting

Searching

Event Processing Revisited

File Choosers and Color Choosers

Sliders

Binding

- **Consider the following method invocation:**

when compiling, it doesn't call this doIt, it just makes sure that they have this method defined.

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

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

- **If this binding occurred 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***

- **Late binding provides flexibility in program design**

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 invoked 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 doesn't have to point to new Occupation(), it can be any capatible type - which means anything that inherits from it

- **Java allows this reference to point to an `Occupation` object, or to any object of <u>any compatible type</u>**

- **This compatibility can be established using inheritance or using interfaces**

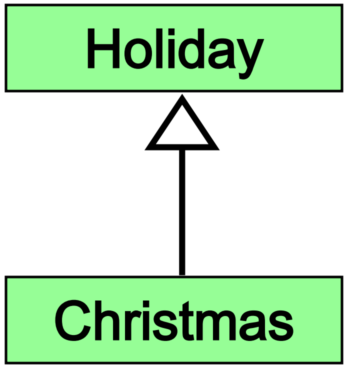
- **Careful use of polymorphic references can lead to elegant, robust software designs**

Outline

References and Inheritance

- **An object reference can refer to an object of its class, or to an object of any class related to it by inheritance**

- **For example, if the `Holiday` class is used to derive a class called `Christmas`, then a `Holiday` reference could be used to point to a `Christmas` object**



Christmas is the child

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

this is polymorphic reference - day to point to any of Holiday's children

say that holiday has the method celebrate. christmas extends holiday
holiday has celebrate() method - have fun
but for christmas, celebrate() method - open presents

References and Inheritance

- **Assigning a child object to a parent reference is considered to be a widening conversion, and can be performed by simple assignment**

parent can point to children - the children is considered 'more' cause you are adding additional methods and variables

- **Assigning an parent object to a child reference can be done also, but it is considered a narrowing conversion and must be done with a cast**

(Holiday)Christmas xmas = new Holiday();   this is narrowing - child points to parent object (must be cast)
child reference   parent object

- **The widening conversion is the most useful**

Polymorphism via Inheritance

reference - object

- It is the type of the object being referenced, not the reference type, that determines which method is invoked
- Suppose the `Holiday` class has a method called `celebrate`, and the `Christmas` class overrides it
- Now consider the following invocation:

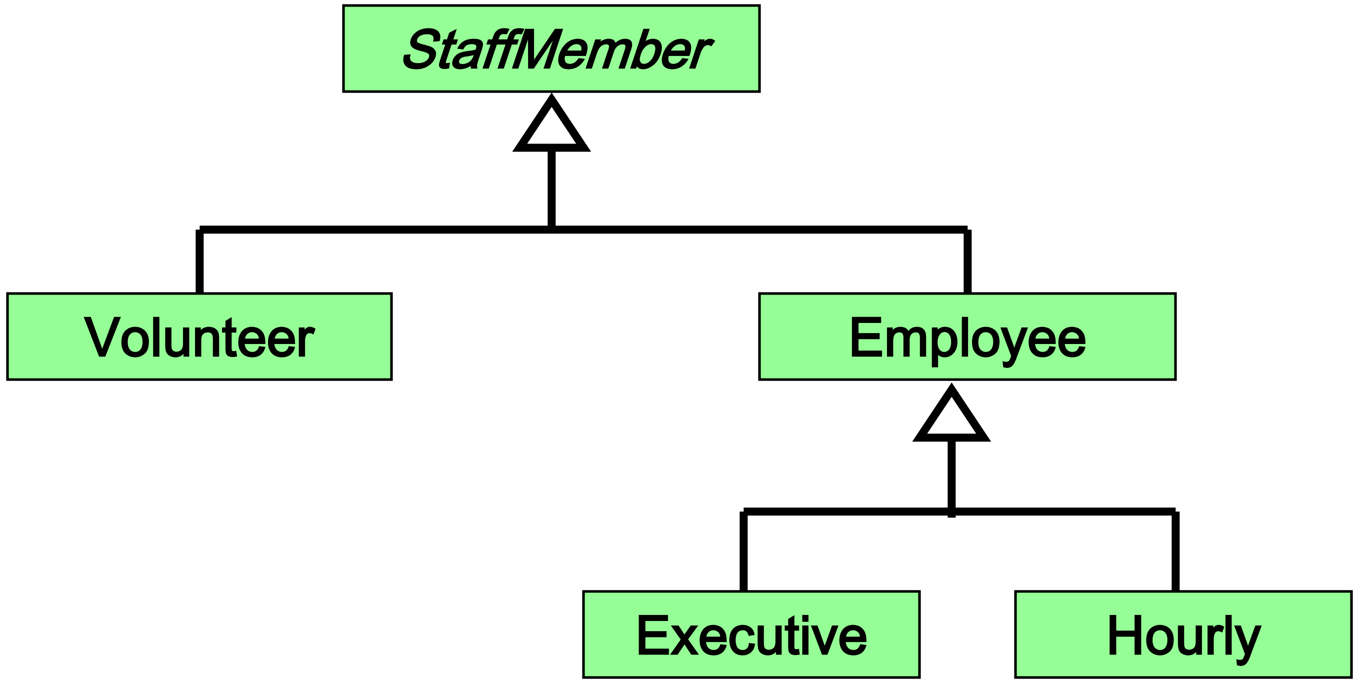
```
day.celebrate();
```

- If `day` refers to a `Holiday` object, it invokes the `Holiday` version of `celebrate`; if it refers to a `Christmas` object, it invokes the `Christmas` version

note: a parent reference will call child's overridden method
but if child has a new method: sing_carols the parent can't call that method because the parent does not have that method
the way to get the parent to sing_carols, will have to do a cast:
((Christmas)day).sing_carols

Polymorphism via Inheritance

- **Consider the following class hierarchy:**



Polymorphism via Inheritance

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

- See `Firm.java` (page 488)
- See `Staff.java` (page 489)
- See `StaffMember.java` (page 491)
- See `Volunteer.java` (page 493)
- See `Employee.java` (page 494)
- See `Executive.java` (page 495)
- See `Hourly.java` (page 496)

Outline

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` that the following line invokes depends on the type of object that `current` is referencing

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

Polymorphism via Interfaces

- **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();
```

11/9/17 end here

Outline

Sorting

- *Sorting* is the process of arranging a list of items in a particular order
- The sorting process is based on specific value(s)
  - sorting a list of test scores in ascending numeric order
  - sorting 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 approach of Selection Sort:
  - select a value and put it in its final place into 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

- **An example:**

```
original:         3   9   6   1   2
smallest is 1:    1   9   6   3   2
smallest is 2:    1   2   6   3   9
smallest is 3:    1   2   3   6   9
smallest is 6:    1   2   3   6   9
```

- **Each time, the smallest remaining value is found and exchanged with the element in the "next" position to be filled**

Swapping

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

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

Polymorphism in Sorting

- Recall that an 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 a group of `People`, or `Books`, or whatever

Selection Sort

there are 2 sorts that we are responsible for: selection and

- The sorting method doesn't "care" what it is sorting, it just needs to be able to call the `compareTo` method
- That is guaranteed by using `Comparable` as the parameter type
- Also, this way each class decides for itself what it means for one object to be less than another
- See `PhoneList.java` (page 502)
- See `Sorting.java` (page 503), specifically the `selectionSort` method
- See `Contact.java` (page 505)

Insertion Sort

- **The approach 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 addition**
  - **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**

selection sort: the lowest number gets swaped to the front,
then that number does not need to be checked again.
the next number will be the 'min' then check through the rest of the list, find the
min, and then then the min and the first checked number for this round is swapped.

Insertion Sort

- **An example:**

```
original:      3    9    6    1    2
insert 9:      3    9    6    1    2
insert 6:      3    6    9    1    2
insert 1:      1    3    6    9    2
insert 2:      1    2    3    6    9
```

- **See `Sorting.java` (page 503), specifically the `insertionSort` method**

inserttion/bubble sort: 3 is a sorted sublist, then check 9. its less than 3 so it doesn't move. then, check 6. since 6 is less than 9 but not less than 3, then it will insert itself between 3 and 9. then, 1 moves to the front. then, 2 moves in between 1 and 3

Comparing Sorts

big O notation

- **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 $n \log_2 n$***

algorithmic time
n linear search
n^2
log2 n

Outline

Polymorphic References

Polymorphism via Inheritance

Polymorphism via Interfaces

Sorting

11/14/17 ended here

→ Searching

Event Processing Revisited

File Choosers and Color Choosers

Sliders

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
- See `PhoneList2.java` (page 510)
- See `Searching.java` (page 511), specifically the `linearSearch` method

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**
- **See `PhoneList2.java` (page 510)**
- **See `Searching.java` (page 511), specifically the `binarySearch` method**

Outline

Polymorphic References

Polymorphism via Inheritance

Polymorphism via Interfaces

Sorting

Searching

➡ Event Processing Revisited

File Choosers and Color Choosers

Sliders

Event Processing

- **Polymorphism plays an important role in the development of a Java graphical user interface**
- **As we've seen, we establish a relationship between a component and a listener:**

```
JButton button = new JButton();
button.addActionListener(new MyListener());
```

- **Note that the `addActionListener` method is accepting a `MyListener` object as a parameter**
- **In fact, we can pass the `addActionListener` method any object that implements the `ActionListener` interface**

Event Processing

- **The source code for the `addActionListener` method accepts a parameter of type `ActionListener` (the interface)**
- **Because of polymorphism, any object that implements that interface is compatible with the parameter reference variable**
- **The component can call the `actionPerformed` method because of the relationship between the listener class and the interface**
- **Extending an adapter class to create a listener represents the same situation; the adapter class implements the appropriate interface already**

Outline

Polymorphic References

Polymorphism via Inheritance

Polymorphism via Interfaces

Sorting

Searching

Event Processing Revisited

➡ File Choosers and Color Choosers

Sliders

Dialog Boxes

- Recall that a dialog box is a small window that "pops up" to interact with the user for a brief, specific purpose
- The `JOptionPane` class makes it easy to create dialog boxes for presenting information, confirming an action, or accepting an input value
- Let's now look at two other classes that let us create specialized dialog boxes

File Choosers

- **Situations often arise where we want the user to select a file stored on a disk drive, usually so that its contents can be read and processed**
- **A *file chooser*, represented by the `JFileChooser` class, simplifies this process**
- **The user can browse the disk and filter the file types displayed**
- **See `DisplayFile.java` (page 518)**

Color Choosers

- **In many situations we want to allow the user to select a color**
- **A *color chooser* , represented by the `JColorChooser` class, simplifies this process**
- **The user can choose a color from a palette or specify the color using RGB values**
- **See `DisplayColor.java` (page 521)**

Outline

Polymorphic References

Polymorphism via Inheritance

Polymorphism via Interfaces

Sorting

Searching

Event Processing Revisited

File Choosers and Color Choosers

➡ Sliders

Sliders

- **A slider is a GUI component that allows the user to specify a value within a numeric range**
- **A slider can be oriented vertically or horizontally and can have optional tick marks and labels**
- **The minimum and maximum values for the slider are set using the `JSlider` constructor**
- **A slider produces a *change event* when the slider is moved, indicating that the slider and the value it represents has changed**

Sliders

- **The following example uses three sliders to change values representing the color components of an RGB value**
- **See `SlideColor.java` (page 524)**
- **See `SlideColorPanel.java` (page 525)**

Summary

- **Chapter 9 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**