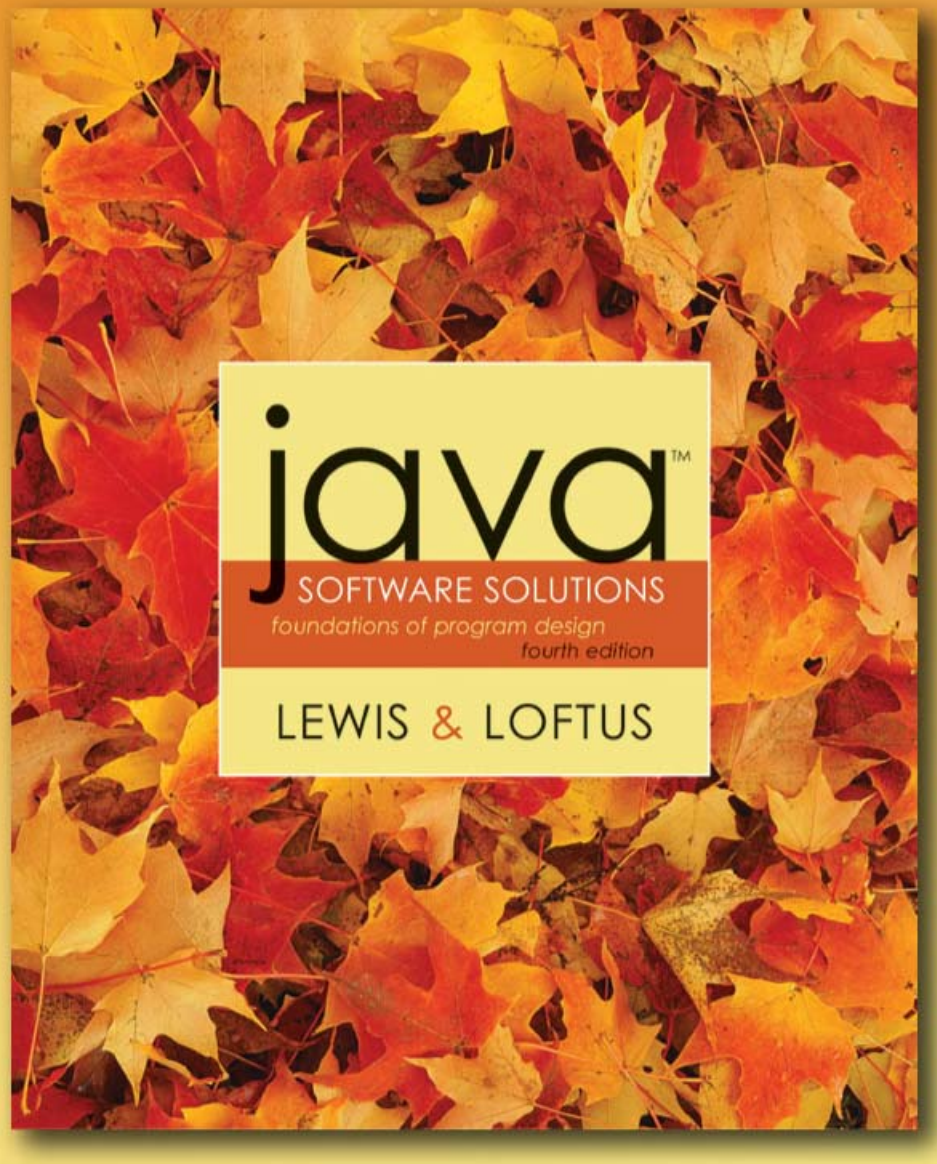
Lecture 9

Polymorphism

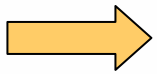


PEARSON Addison Wesley

Polymorphism

- **Polymorphism is an object-oriented concept that allows us to create versatile software designs**
- **Lecture 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**

Outline



Polymorphic References

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
- 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

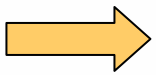
Polymorphism

- **Suppose we create the following reference variable:**

```
Occupation job;
```

- **Java allows this reference to 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



Polymorphic References

Polymorphism via Inheritance

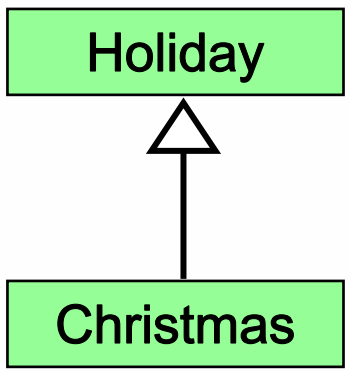
Polymorphism via Interfaces

Sorting

Searching

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




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

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

Polymorphism via Inheritance

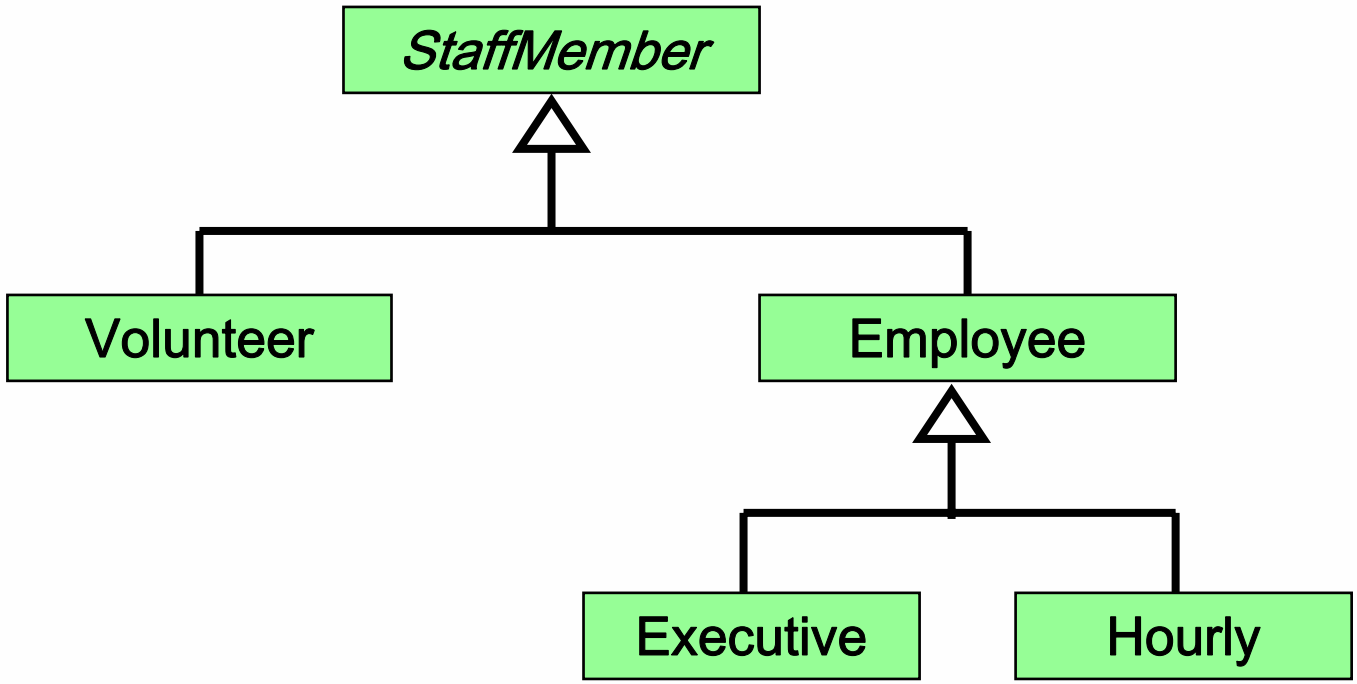
- It is the type of the object being referenced, not the reference type, that determines which method is invoked
- Suppose the `Holiday` class has a method called `celebrate`, and the `Christmas` class overrides it
- Now consider the following invocation:

```
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

Polymorphism via Inheritance

- **Consider the following class hierarchy:**

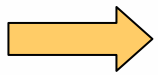


Polymorphism via Inheritance

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

- See `Firm.java` (page 486)
- See `Staff.java` (page 487)
- See `StaffMember.java` (page 489)
- See `Volunteer.java` (page 491)
- See `Employee.java` (page 492)
- See `Executive.java` (page 493)
- See `Hourly.java` (page 494)

Outline



Polymorphic References

Polymorphism via Inheritance

Polymorphism via Interfaces

Sorting

Searching

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();
```

Outline

- Polymorphic References
- 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 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

- 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 500)
- See `Sorting.java` (page 501), specifically the `selectionSort` method
- See `Contact.java` (page 503)

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**

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 501), specifically the `insertionSort` method**

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* $n \log_2 n$**

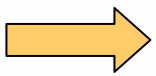
Outline

Polymorphic References

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
- See `PhoneList2.java` (page 508)
- See `Searching.java` (page 509), 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 508)
- See `Searching.java` (page 509), specifically the `binarySearch` method

Summary

- **Lecture 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**