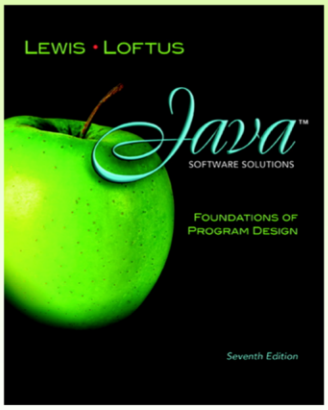
Chapter 10
Polymorphism



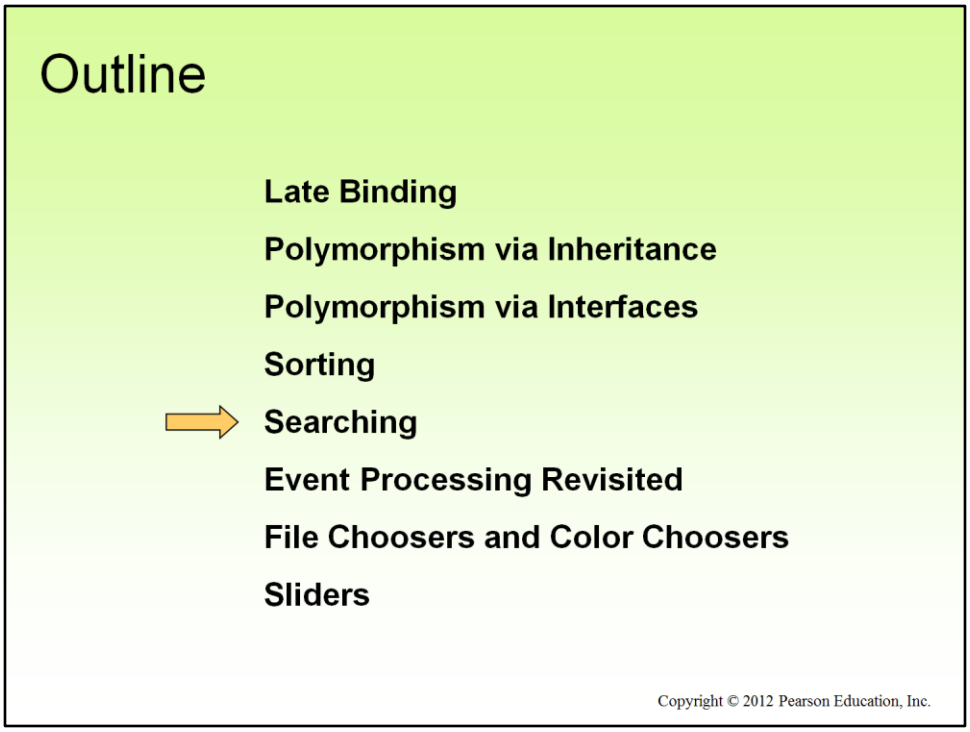
Java Software Solutions
Foundations of Program Design
Seventh Edition

John Lewis
William Loftus

Addison-Wesley
is an imprint of
PEARSON

Polymorphism

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



-Just as we saw with our sorting example, we'll see how we can use polymorphism with interfaces to implement searching

-As before, we'll use a class with static methods that take an array of **interface reference variables** to search

-As with sorting, the interface type we are using is the Comparable interface from the standard Java API

-If we have class objects we want to search, we must **implement** this interface and provide the definition for the compareTo method

-This is the same process we used for sorting and is another great example of using polymorphism with interfaces

-Because of polymorphism, we can write a **single** search method and use it for **any** class object we'd like!

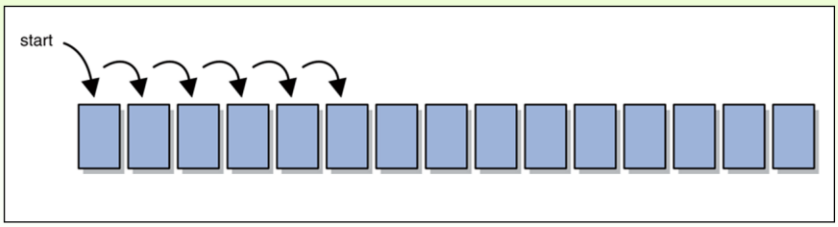
-Once again, polymorphism allows us to program in this more **general** way for any class type!

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



-On average, a linear search will look through **half the number** of elements in the list before a match is found

-This is one way we can classify the **efficiency** of this particular search algorithm

-Note also, that this does not require any pre-processing step such as sorting the list

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

-Note that a binary search requires a pre-processing step of **sorting** the list (ascending or descending) before a search is made

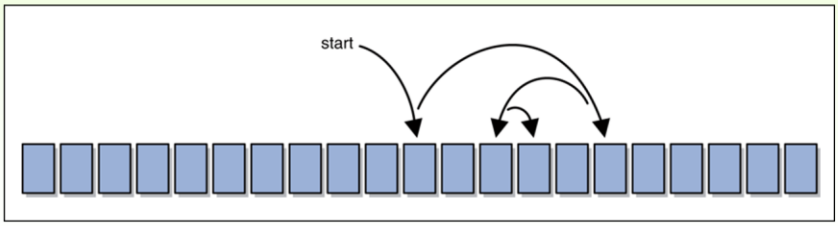
-We must consider this as part of its **efficiency** when comparing to the efficiency of other algorithms

-Note how sequentially entire **halves** of the list of elements are **eliminated** using a binary search

-This makes for a very efficient method of searching (assuming, however, the list is first sorted)

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`

```
//********************************************************************
//  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.");
   }
}
```


-Note we first use the static method, linearSearch, from the Searching class by sending the list (friends) and the searching target (test)

-Note how this method will return the object found and note that we need to type-cast this result

found = (Contact)Searching.linearSearch(friends,test);

-This is necessary because this method returns a Comparable interface reference variable

-To use the result as a Contact, we type-cast as shown above

-Note to use the Searching.binarySearch method, we first must sort the list (Sorting.selectionSort)

-As with the linear search, we then type-cast the result from our binarySearch method to use our Contact result

found = (Contact)Searching.binarySearch(friends,test);

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` 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;
}
```


-As with the Sorting class, we have a static method that accepts an array of Comparable interface reference variables (list)

-It also takes a second Comparable interface reference variable that represents the item we want to match (target)

-Finally, note the return type of this method, it is also a Comparable reference variable

-Note that it can also return **null** if a match is not found

-The value **null** is often stored in reference variables to indicate they are not "pointing" to anything

-In other words, there is no memory address and hence, no real object being referenced

-Anytime you have a reference variable that is not storing anything, it should be assigned this **null** value

-We can also use null in conditional statements such as:

```
Contact result = (Contact)Searching.linearSearch(friends,test);
if( result == null )
        System.out.println("No match found");
```

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

-As with the linear search method, this method accepts and returns Comparable interface reference variables

continue

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


-It also can return **null** if a match is not found