Chapter 7: Sequential Data Structures

We have so far explored the basic features of the Java language, including an overview of object-oriented features, threads, exceptions, and we created a variety of programs and applets using the AWT as well as Swing. In this and the next chapter we will focus instead on traditional programming structures that can be implemented in any of the common programming languages and that are used in most medium to large programming project. Rather than focusing on language constructs specific to Java, we will learn how to *use* Java to implement constructions that are defined in a more abstract way and can be created using any other programming language as well. Most of these new concepts can be explained and implemented without resorting to programs with a graphical user interface, but time permitting we will provide several complete applications that will illustrate the usefulness of our new idea.

This chapter is organized XXX

Quick View

Here is a quick overview of the topics covered in this chapter.

(*) 7.1. Recursion
Single Recursion; Some Useful Recursive Methods; Multiple Recursion; The Cost of Recursion

(*) 7.2. The Tower of Hanoi Applet

(*) 7.3. Searching and Sorting
Simple Search; Binary Search; Selection Sort; Bubble Sort; Insertion Sort; Quick Sort; Sorting Algorithms Compared

7.4. Java's Build-in Searching and Sorting Algorithms
Searching and Sorting Basic Data Types; Searching and Sorting Objects

(*) These sections are optional but recommended

Here is a quick overview of the topics covered in this chapter.

(*) 8.1. Sequential Data Structures
Lists; Queues; Stacks

(**) 8.2. A Mathematical Expression Parser
The Tokenizer; Converting to Postfix; Postfix Evaluation

8.3. Java Build-in Sequential Structures
The Java List Framework; StringTokenizer Class; Iterators

(*) These sections are optional but recommended
(**) These sections are optional

7.1. Searching and Sorting

An area where recursive methods can provide seemingly easy solutions to fairly complex questions is the area of searching and sorting. Indeed, there are many algorithms to solve "searching and sorting" problems, and many research papers have been written on this subject. In this chapter we will by no means cover the breadth of available search and sort techniques. Instead, we will introduce some of the more frequently used algorithms only. This section will lay a foundation for understanding the build-in Java search and sort mechanisms that will be introduced in section 7.3.

Search Algorithms

The general setup for a "searching" problem is simple: Given a list of `N` items, determine if one particular item is part of the list. Occasionally you may also want to know the location of the searched item if it is contained in the list. For this chapter, a "list" will be represented by an array and the items we will be looking for will be doubles. With a few changes other data types, including general objects, can also be searched, which will be covered in the exercises.

The easiest search follows exactly our own intuition. Suppose we are given the numbers

1, 19, 8, 7, 15, and 21

and we want to know whether 15 is included in that list. We seem to be able to answer "yes" almost immediately, without thinking about it. But, consider the following 50 random numbers between 0 and 10:

```
9.22, 9.38, 9.12, 7.79, 7.37, 0.87, 9.23, 5.30, 5.46, 9.16
1.80, 4.22, 6.17, 3.90, 3.80, 2.01, 8.87, 4.84, 3.06, 5.00
9.80, 5.13, 1.58, 9.28, 3.74, 9.46, 3.04, 5.66, 4.81, 5.44
1.02, 4.57, 3.97, 2.78, 1.47, 7.06, 4.49, 5.83, 0.21, 4.16
3.74, 7.88, 3.50, 1.62, 2.04, 2.47, 0.08, 0.00, 9.20, 5.16
```

Is the number 3.54 part of that list? Now we can *not* immediately tell the answer. Instead, what we do (more or less automatically) is to start looking at the first number, compare it against the one we are looking for, continue with the next number and so forth, until we either find the number or we reach the end of the list. Our first search algorithm uses exactly the same mechanism:

Definition 7.3.1: Simple Search

> *If `A` is an array of `double`, then to determine whether a number `key` is part of the array we compare that `key` number against every member of the array, starting from the first, until we either find a match or we reach the end of the array.*

Example 7.3.2:

Create a method that takes as input an array of `double` and a single `double` and returns `true` if the second number is found in the array of doubles. Then create another method with the same input but it returns the index where the second `double` is located in the first array, or -1 if the array does not contain that number.

Based on our description of the algorithm we might want to use a `while` loop to go through the members of the array until we either find a mach or reach the end of the array. However, we can achieve the same result with a `for` loop, remembering that the `return` statement exits a method immediately:

```
public static boolean search(double A[], double key)
{  for (int i = 0; i < A.length; i++)
      if (A[i] == key)
         return true;
   return false;
}
```

The second method is similar, but instead of returning a `boolean` it returns the appropriate integer:

```
public static int searchIndex(double A[], double key)
{  for (int i = 0; i < A.length; i++)
      if (A[i] == key)
         return i;
   return -1;
}
```

■

These methods will certainly work nicely, but we can also come up with an interesting alternative solution.

Example 7.3.3:

Redo the above method search so that it returns `true` if the key is found, but throws an `ItemNotFound` exception if it does not find the key.

First, we need to create the `ItemNotFoundException`. Recall that creating your own exception is accomplished by extending the default `Exception` class and providing it with two appropriate constructors (see definition 5.1.11 creating your own exceptions):

```
public class ItemNotFoundException extends Exception
{  public ItemNotFoundException()
   {  super("Item not found"); }
   public ItemNotFoundException(String msg)
   {  super("Item not found: " + msg); }
}
```

Now we can modify either of the above methods so that they throws an instance of this exception if the item has not been found. For example:

```
public static boolean search(double A[], double key)
               throws ItemNotFoundException
{  for (int i = 0; i < A.length; i++)
      if (A[i] == key)
         return true;
```

```
      throw new ItemNotFoundException();
   }
```

This has the (possible) advantage that you can now use this modified `search` method without having to use `if` statements to test whether you found an item. You can simply assume that you have found it in the `try` part of the block, while handling the case where you do not find the item in the `catch` block. Here is an easy example on how to use this method (don't forget to create the `ItemNotFoundException` first):

```
public class TestSearch
{  public static boolean search(double A[], double key)
                  throws ItemNotFoundException
   {  /* as before, no changes */ }
   public static void main (String args[])
   {  double A[] = {1,2,3,4,5,6,7,8,9,10};
      try
      {  System.out.println(search(A, 3.0));
         System.out.println(search(A, -3.0));
      }
      catch(ItemNotFoundException ie)
      {   System.out.println(ie);
      }
   }
}
```

Of course, the disadvantage now is that you must use a `try-catch` block to use this new exception. Figure 7.3.1 shows the result of running this class.

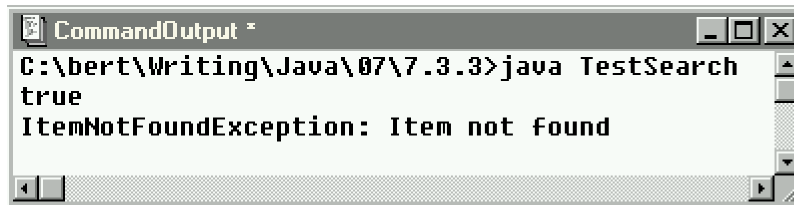


Figure 7.3.1: Result of `TestSearch` *program*

A more detailed discussion of the benefits of either approach is left as an exercise.

■

[xxx remove paragraph] While this approach certainly works, it is somewhat simple minded. There are other alternatives, and one in particular, which is reasonably simple to implement yet performs orders of magnitude faster when applicable.
[xxx end remove]

The worst case scenario for any search to be conducted is if you are searching for an item that is not present. For our current algorithm, this means that *every* item of the entire array has to be looked at before we can be certain that our item is not contained in the array. There is another type of search that is much quicker, i.e. you do not have to look at all elements in an array before being able to determine that your search item is not present.

Definition 7.3.4: Binary Search

Binary Search is a search algorithm that applies to arrays that are sorted. For simplicity, we will assume that the array is sorted in ascending order, i.e. starting with the smallest element.

> *To search for a `key` element, we start with the element in the middle of the sorted array, and determine if `key` is equal to, less than, or bigger than the middle element. If it is equal, the search is over. If `key` is less than that, you can discard every element above the current one. If the `key` is larger than that, you can discard everything below the current element. Therefore, you can split the array in half, discarding either the top or bottom half. Then you repeat the procedure on this new "half array" until you either find `key` in the array, or you can no longer determine a middle element.*

This definition probably sounds much more complicated that it really is. Let's consider a simple example:

Example 7.3.5:

Suppose you have the sorted array `A` = {-1, 1, 3, 5, 6, 7, 9, 10, 12, 20, 25} and you want to search for, say, 50, then for, say, -10, and finally for 6. Illustrate in each case how the binary search algorithm works.

Suppose we are searching the array `A` for 50. First, we start with the element in the middle, in our case the number 7. The key 50 is bigger than that number. Since our array is sorted that means that 50 can only be in the "upper half" of the array, i.e. between the number 7 and the last one. Therefore, we discard the lower half of the array and we continue our search with the (sub) array {9, 10, 12, 20, 25}. This time the middle number is 12. The key we are looking for (50) is again larger than 12, so we again discard the lower half of that array. We will therefore continue searching the array {20, 25}. This time, there is no middle number, so we pick either one of these numbers, say 20. The key 50 is again bigger, so we discard the lower half and continue searching the array {25}. This number is not equal to our key, but we can also not split this single number up again. Therefore, our search is over and we have determined that 50 is not part of the original array.

Now let's search the array `A` for -10. We pick the middle number 7, compare against our key -10. The key is less than the middle element, so we discard everything bigger than 7 and continue searching the array {-1, 1, 3, 5, 6 }. Picking the middle number 3 and comparing against our key -10 again results in our discarding the upper half of that array and continue our search in {-1, 1}. We pick either one of these numbers as the middle one, say -1. Comparing against our key results in discarding once again the upper half, to continue our search with {-1}. But that number is not equal to our key, and we can not split that "array" any further - therefore -10 is not part of our array.

Finally, let's search the array `A` for 6. Picking the middle number 7 and comparing against our key results in discarding the upper half of the array to continue the search in {-1, 1, 3, 5, 6}. Picking the middle number 3 and comparing against the key (6) will now result in our discarding the lower half of that array and continuing the search in {5, 6}. Picking either one, say 5, and comparing against 6 means to continue the search with the array {6}. That, indeed, is equal to our key, so that we have determined that 6 is part of the original array.

■

Example 7.3.6:

Assuming that the array `A` is an array of double numbers that is sorted in ascending order, create a method that will search this array using the binary search algorithm.

At first glance this might seem quite tricky. How can we "discard" parts of an array? Isn't it going to be tricky to determine which part of the array to discard?

First, we need to determine our method header. We obviously need an array and a double number as input. In addition, we do not always want to consider the whole array but a subarray of the original one. Therefore, we use two additional parameters indicating the starting and ending index of the subarray to be considered:

```
public static boolean binarySearch(double A[], double key,
                                   int start, int end)
{  /* applies binary search algorithm to search for key in A between
      start and end. */ }
```

Our algorithm asks for finding the middle element. Therefore, we need to compute

```
int mid = (start + end) / 2;
```

to see if we have found `key` at that index position, i.e. (`A[mid] == key`). If so we would be done. Otherwise, if (`A[mid] < key`) we need to search the subarray from index position `mid+1` to `end`, and if (`A[mid] > key`) we continue our search from `start` to `mid-1`. And in fact we already know how to "continue searching": we use the same method we are trying to define for that. The last condition to implement would be to decide when we can no longer "split" the interval in half (and the search would have failed). That is the case as soon as `(start > end)`, and would, in fact, be our "base case" for a recursion. Therefore, we will define our recursive method as follows:

```
public static boolean binarySearch(double A[], double key,
                                   int start, int end)
{  int mid = (start + end) / 2;
   if (start > end)
      return false;
   else if (A[mid] == key)
      return true;
   else if (A[mid] < key)
      return binarySearch(A, key, mid + 1, end);
   else
      return binarySearch(A, key, start, mid-1);
}
```

Our `binarySearch` method[1] requires two additional input parameters `start` and `end` that are necessary for the recursion but may be confusing to the user. That is no reason to discard our method because we can simply hide it making it private and setting up a convenient public "front-end" method:

```
public static boolean search(double A[], double key)
{  return binarySearch(A, key, 0, A.length-1); }
private static boolean binarySearch(double A[], double key,
                                    int start, int end)
{  /* as before */  }
```

■

As stated before, recursive methods may be expensive in terms of memory requirements, so we might want to create a non-recursive version of `binarySearch` (using the same method header as the `search` method above to make life simple for the user):

```
public static boolean search(double A[], double key)
{  int start = 0;
   int end   = A.length - 1;
```

[1] This is not a double recursive method even though there are two recursive calls in the method body. But in any given situation only *one* of those two calls will execute, hence this method is a single recursion.

```
      while (start <= end)
      {  int mid = (start + end) / 2;
         if (A[mid] == key)
            return true;
         else if (A[mid] < key)
           start = mid + 1;
         else
            end = mid-1;
      }
      return false;
   }
```

In the appendix we will analyze exactly how much faster the binary search algorithm is over the standard one. For now we will focus on extending our search algorithm(s) to include searching for other types. In our current discussion on searching we have only searched arrays of doubles. That was particularly convenient since `double` numbers readily compare via the standard comparison operators. That does not mean that *only* numbers can be searched.

In fact, in examples 4 and 6, section 3.6, we created `Sorter` classes to sort arrays of any type. We will create an analogous "general purpose" `Searcher` class that can search arrays of any type as long as the type implements a `Searchable` interface.

Example 7.3.7:

Create a general-purpose `Searcher` class that can search arrays of any type, using either the simple search or binary search algorithm. In addition, instead of simply returning true or false, the search algorithm should return a reference to the found object if available, or throw an `ItemNotFound` exception if it can not find the item.

To do this example, you should review the `Sorter` class we discussed in examples 4 and 6, section 3.6, as well as example 7.3.3 where we throw a custom made exception when an item is not found.

In any case, the simple search algorithm relies on the fact that we can check whether two entities are equal, while the binary search algorithm also requires us to be able to determine which of two entities is "bigger". Therefore, only items that can be compared in this way can be searched with these algorithms and our first task is to ensure that our items indeed can be compared properly.

Therefore, we create a `Searchable` interface that specifies three (abstract) methods as follows:

```
public interface Searchable[2]
{  public abstract boolean isEqualTo(Searchable obj);
   public abstract boolean isLessThan(Searchable obj);
   public abstract boolean isGreaterThan(Searchable obj);
}
```

Next, we create the `ItemNotFound` exception according to definition 5.1.11:

```
public class ItemNotFoundException extends Exception
{  public ItemNotFoundException()
   {  super("Item not found"); }
   public ItemNotFoundException(String msg)
   {  super("Item not found: " + msg); }
}
```

[2] As we will see in section 7.4, the build-in Java search methods use an interface `Comparator` that requires objects to implement `public int compare(Object o1, Object o2)`. That method returns –1, 0, or 1, depending on whether `o1` is less than, equal to, or bigger than `o2`.

Now we can create the `Searcher` class that searches an array of `Searchable` objects for a `key` that is also of type `Searchable`. The actual algorithm is much like it was before, but instead of returning `true` if an item is found it returns a reference to the object found and instead of returning `false` if the item is not found we throw an `ItemNotFound` exception. To make our class user friendly, we create one overloaded public `search method`, so that one version uses the simple search algorithm, and the second one uses binary search. It is left to the user to ensure that the array is properly sorted before using binary search.

```
public class Searcher
{  // constants
   public static final int SIMPLE = 1;
   public static final int BINARY = 2;
   // public methods
   public Object search(Searchable A[], Searchable key)
                   throws ItemNotFoundException
   {  for (int i = 0; i < A.length; i++)
         if (A[i].isEqualTo(key))
            return A[i];
      throw new ItemNotFoundException();
   }
   public Object search(Searchable A[], Searchable key, int method)
                   throws ItemNotFoundException
   {  if (method == SIMPLE)
         return search(A, key);
      else
         return binarySearch(A, key, 0, A.length-1);
   }
   // private methods
   private static Object binarySearch(Searchable A[], Searchable key,
                   int start, int end) throws ItemNotFoundException
   {  int mid = (start + end) / 2;
      if (start > end)
         throw new ItemNotFoundException();
      else if (A[mid].isEqualTo(key))
         return A[mid];
      else if (A[mid].isLessThan(key))
         return binarySearch(A, key, mid + 1, end);
      else
         return binarySearch(A, key, start, mid-1);
   }
}
```

Now, of course, we should check whether the above `Searcher` class really works.

Example 7.3.8:

Create an `Address` class that stores and displays the first and last name of an address as well as an email address. It should implement the `Searchable` interface where two addresses are considered equal if their first and last name matches. If they are not equal and have different last names, the first address is considered bigger if the last name of the first address is lexicographically bigger than that of the second. If the last names are equal but the first names differ, we consider the first one bigger if the first name of the first address is lexicographically bigger than the second. Create an array of these addresses and use the `Searcher` class to search for a particular address.

The Address class is completely straightforward, except that it needs to implement the Searchable interface. That forces the class to define the methods isEqualTo, isGreaterThan, and isLessThan.[3]

```
public class Address implements Searchable
{  private String first;
   private String last;
   private String email;
   public Address()
   {  first = last = email = new String(); }
   public Address(String _first, String _last, String _email)
   {  first = _first;
      last  = _last;
      email = _email;
   }
   public String toString()
   {  return last + ", " + first + " (" + email + ")"; }
   public boolean isEqualTo(Searchable other)
   {  if (!(other instanceof Address))
         return false;
      else
         return (last.equals(((Address)other).last) &&
                 first.equals(((Address)other).first));
   }
   public boolean isLessThan(Searchable other)
   {  if (!(other instanceof Adress))
         return false;
      else
      {  String otherFirst = ((Address)other).first;
         String otherLast  = ((Address)other).last;
         if (last.equals(otherLast))
            return (first.compareTo(otherFirst) < 0);
         else
            return (last.compareTo(otherLast) < 0);
      }
   }
   public boolean isGreaterThan(Searchable other)
   {  if (!(other instanceof Address))
         return false;
      {  String otherFirst = ((Address)other).first;
         String otherLast  = ((Address)other).last;
         if (last.equals(otherLast))
            return (first.compareTo(otherFirst) > 0);
         else
            return (last.compareTo(otherLast) > 0);
      }
   }
}
```

Please note that in order to access the private fields in the comparison methods we need to type-cast the objects from Searchable to Address (private fields of one Address can be accessed by other objects that are also of type Address). However, the input type to those methods must be Searchable because that is as dictated by the Searchable interface.

Having created all appropriate classes (exception class, Searchable interface, Searcher class, and Address class), we are ready to test if everything really works as advertised. The test class is simple: it instantiates the necessary objects such as a Searcher object and an array with various addresses.

[3] These methods should accept any object that implements Searchable, even if they are not of type Address. Therefore, we must make sure that the three methods return false if used with input types other than Address.

Then it defines two addresses, one present in the array and other not. Finally we search for either one, displaying the result of the search if possible:

```
public class SearchForAddress
{  public static void main(String args[])
   {  Searcher searcher = new Searcher();
      Address book[] = new Address[5];
      book[0] = new Address("Bert", "Wachsmuth", "wachsmut@shu.edu");
      book[1] = new Address("Nicole", "Wachsmuth", "");
      book[2]= new Address("Horst", "Wachsmuth", "howa@t-online.de");
      book[3] = new Address("Silke", "von der Emde","");
      book[4] = new Address("Leah", "Wachsmuth","");

      Address lookFor1 = new Address("Leah", "Wachsmuth","lives@home");
      Address lookFor2 = new Address("John", "Doe", "");

      try
      {  System.out.println(searcher.search(book, lookFor1));
         System.out.println(searcher.search(book, lookFor2));
      }
      catch(ItemNotFoundException ie)
      {  System.out.println(ie);
      }
   }
}
```

When we execute this class, the resulting output is as shown on the right. In particular, "Wachsmuth, Leah" was indeed found, even though the email address does not match the search key. Also, the displayed address has no email, which is again correct.[4]

```
CommandOutput *
C:\>java SearchForAddress
Wachsmuth, Leah ()
ItemNotFoundException: Item not found

C:\>
```

Figure 7.3.2: Using general-purpose Searcher class

■

Now that we have seen two different search algorithms, the obvious question is which one is better. The answer, of course, depends: if the array is not sorted, there's only one possible method – by default, it is best. If the array is sorted, there are two candidates, simple search and binary search. As we will analyze in the appendix, binary search is *a lot* more efficient than simple search.

[4] The address displayed is the one found not the one in the search `key`. Therefore it is important to return a reference to the object found. It is true that this object will have at least something in common with the known `key` (otherwise they would not be equal) but it might contain additional information different from `key`.

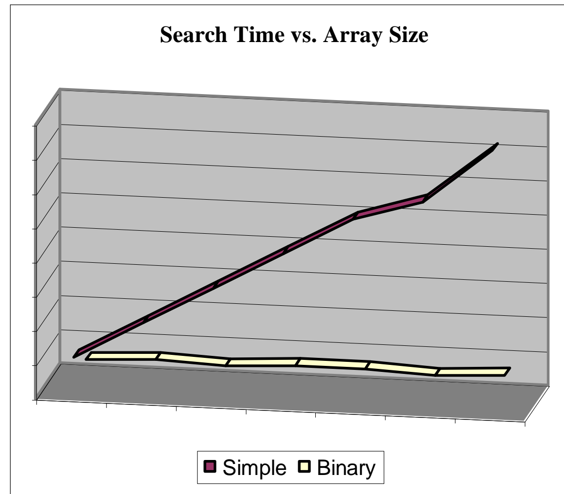


Figure 7.3.3: Search times versus array size for simple and binary search

Figure 7.3.3 shows an almost linear dependency of search time on array size, i.e. if the array size doubles, the search time doubles as well.[5] For binary search it seems as if the size of the array has no impact on the search time. While that is of course not true, binary search is so fast that arrays of a size that would matter to binary search can not be instantiated on a PC because of memory constraints.[6] A more detailed analysis of the performance of these algorithms can be found in the appendix.

Sort Algorithms

Just as for searching the general idea of sorting is easy: we start with an array of items in no particular order and we want to convert it into a sorted array in either ascending or descending order. We will restrict our discussion to arrays of double numbers because they can easily be compared using the standard comparison operators.[7]

Selection Sort

To determine our first sorting algorithm, consider the following 50 random numbers:

```
9.22, 9.38, 9.12, 7.79, 7.37, 0.87, 9.23, 5.30, 5.46, 9.16
1.80, 4.22, 6.17, 3.90, 3.80, 2.01, 8.87, 4.84, 3.06, 5.00
9.80, 5.13, 1.58, 9.28, 3.74, 9.46, 3.04, 5.66, 4.81, 5.44
1.02, 4.57, 3.97, 2.78, 1.47, 7.06, 4.49, 5.83, 0.21, 4.16
3.74, 7.88, 3.50, 1.62, 2.04, 2.47, 0.08, 0.00, 9.20, 5.16
```

[5] That is no surprise: since simple search looks at every element in the array, it will naturally take longer if there are more elements in the array.
[6] The search time should really be close to zero, but there is a certain overhead involved in calling a method to measuring the time it takes to finish. That time is shown in this chart, while the actual time to search the arrays is negligible for binary search. In this case the largest array was an array of 1,300,000 address objects.
[7] It is simple to extend all algorithms to arrays of `Object` by using a suitable interface. The details are left as exercises.

If you were to sort them starting with the smallest number, how would you do it? The strategy you are probably going to adopt will be similar to this:

- select the smallest of all 50 numbers, put that first and cross out that number in the original set
- select the smallest of the remaining 49 numbers, put that second and cross out that number in the original set
- select the smallest of the remaining 48 numbers, put that third and cross out that number in the original set
- continue in this fashion until all numbers are sorted

That method, indeed, is the basis of our first sorting algorithm (we have implemented that algorithm already in the `Sorter` class in example 4, section 3.6):

Definition 7.3.9: Selection Sort

Suppose `A` is an array of `double` numbers. To sort this array in ascending order using the selection sort algorithm:

- *Find the smallest element in the entire array `A` and swap it with the first element. Now the first element is in the correct position (being the smallest one in `A`).*
- *Find the smallest element in `A` minus the first element, and swap it with the second element. Now the first two elements are in the correct order.*
- *Find the smallest element in `A` minus the first two elements, and swap it with the third element. Now the first three elements are in the correct order.*

Continue in this fashion until the last element is reached. All elements in `A` are now in increasing order.

This algorithm is illustrated in figure 7.3.4:

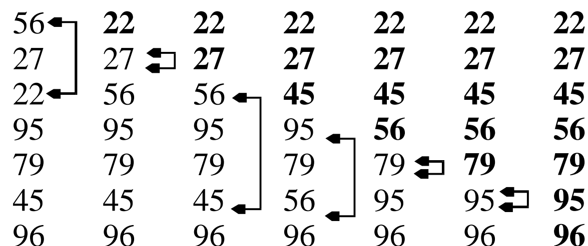


Figure 7.3.4: How Selection Sort works (bold numbers are in correct position)

- 22 is the smallest overall number and is swapped with the first number
- 27 is the smallest number starting at the second one, and swapped with itself
- 45 is the smallest number starting at the third, and swapped with the third
- 56 is the smallest number starting at the fourth, and swapped with the fourth
- 79 is the smallest number starting at the fifth, and swapped with itself
- 95 is the smallest number starting at the sixth, and swapped with itself

The last number must now be the largest, and the entire array is sorted.

Example 7.3.10:

Implement the Selection Sort algorithm and test it on a random array of size 10 to see if it works properly.

If we look at this algorithm we can isolate two distinct tasks:

- We need to be able to find the location of the smallest number in a part of an array.

We will use a method `selectMin` that takes as input an array and an integer indicating the starting index of where in the array we want to start looking for the minimum. It will return the index of the smallest element in the corresponding subarray.

- We need to be able to swap two elements in an array.

We will use a method `swap` that takes as input an array and two integers and swaps the corresponding entries in the array.

If we can solve these tasks the rest of the selection sort algorithm can be accomplished in a simple loop. Here is the implementation of the first method to find the location of the smallest number in an array `A`, starting at a particular index `start`:

```
public static int selectMin(double A[], int start)
{  int smallest = start;
   for (int i = start+1; i < A.length; i++)
      if (A[i] < A[smallest])
         smallest = i;
   return smallest;
}
```

Initially, we assume that the smallest number in the entire array (lacking an alternative) is located at `start`. Then we look at each number in the array from `start+1` to the end of the array. If we find a number that is smaller than `smallest`, then our initial idea was not correct and we reset `smallest` to that index. When the method is finished, `smallest` will indeed contain the index of the smallest number in the array `A` starting at `start`.

As for the method to swap two elements in an array, it is simple:

```
public static void swap(double A[], int pos1, int pos2)
{  double tmp = A[pos1];
   A[pos1] = A[pos2];
   A[pos2] = tmp;
}
```

Note, in particular, that this method returns void but does swap the respective elements of the array, because arrays are reference objects. With these methods in place, sorting the array is quite simple:

```
public static void sortSelect(double A[])
{  for (int i = 0; i < A.length-1; i++)
      swap(A, i, selectMin(A, i));
}
```

This method performs exactly like the algorithm prescribes: it starts with i = 0, finds the smallest element in the entire array, and swaps it with the 0th element in the array. Then it finds the smallest element in the array starting at index 1 and swaps it with the 1st element in the array, and so on.

What is left for us to do is to test these methods to see if the process really works:

```
public class SortTest
{  public static int selectMin(double A[], int start)
   { /* as above */ }
   public static void swap(double A[], int pos1, int pos2)
   { /* as above */ }
   public static void sortSelect(double A[])
   { /* as above */ }
   public static void printArray(double A[])
   {  for (int i = 0; i < A.length; i++)
         System.out.println(A[i]);
   }
   public static void main(String args[])
   {  double A[] = new double[10];
      for (int i = 0; i < A.length; i++)
         A[i] = Math.random()*10;
      System.out.println("Before sorting");
      printArray(A);
      sortSelect(A);
      System.out.println("After sorting");
      printArray(A);
   }
}
```

When you compile and run this class you will see that indeed our selection sort algorithm works as intended.

■

This sorting algorithm works nicely, but if we were sorting large arrays we would notice that the algorithm is slow. For example, in 3D game programming we might need an algorithm to draw geometric objects so that objects in the front really appear "before" those in the back. One such algorithm involves sorting the objects to draw by depth, then drawing from the back to the front. If the sorting algorithm is slow, the entire drawing of the scenery will suffer. Sorting is, in fact, a quite frequent task for a variety of data, in any kind of programming, so that we would very much like to improve our sort algorithm, if possible.

Bubble Sort

Therefore, we will investigate a few more sorting algorithms and compare them in terms of their speed in a later example. Our next algorithm has a nice name, but as it will turn out that's about all that can be said in its favor.

Definition 7.3.11: Bubble Sort (Exchange Sort)

Suppose `A` *is an array of* `double` *numbers. To sort this array in ascending order using the bubble sort algorithm:*

- *Process the full array by comparing consecutive elements: if the* `(i+1)`st *element is less than the* `i`th *element, swap them with each other, otherwise leave them alone. If no exchange is made, the array is sorted and we are done. At the end of this pass the overall largest number will have "bubbled" to the last position of the array.*
- *Repeat this "bubbling" procedure, but apply it to the current array except the last element. If no exchange was made, the array is sorted and we are done. At the end of this pass the second-largest number will have "bubbled" to the second-last position of the array.*

> - *Repeat this "bubbling" procedure, but apply it to the current array except the last two elements. If no exchange was made, the array is sorted and we are done. At the end of this pass the third-largest number will have "bubbled" to the third-last position of the array.*
>
> *Continue in this fashion until either no exchange was necessary during a particular pass, or there is nothing left to sort. This algorithm is also frequently called Exchange Sort.*

The first pass of this algorithm is illustrated in figure 7.3.5:

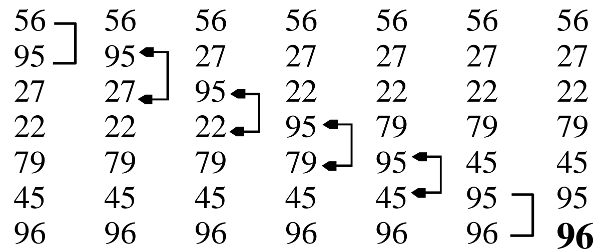


Figure 7.3.5: How Bubble Sort works - First Pass Only (bold number is in correct position)

- 95 is not less than 56, so no exchange is necessary
- 27 is less than 95, so we exchange them (which means the array was not sorted)
- 22 is less than 95, so we exchange them
- 79 is less than 95, so we exchange them
- 45 is less than 95, so we exchange them
- 96 is not less than 95, so no exchange is necessary

The largest number has bubbled to the end of the array. Now we repeat this procedure, but considering the array 56, 27, 22, 79, 45, and 95 (the number already in final position is shown in bold):

- 56 and 27 are exchanged, resulting in 27, 56, 22, 79, 45, 95, **96**
- 56 and 22 are exchanged, resulting in 27, 22, 56, 79, 45, 95, **96**
- 56 and 79 are not exchanged
- 79 and 45 are exchanged, resulting in 27, 22, 56, 45, 79, 95, **96**
- 79 and 95 are not exchanged

Now the array is 27, 22, 56, 45, 79, **95**, and **96**, and the "bubbling" process starts again, considering the array 27, 22, 56, 45, and 79 (the numbers already in final position are in bold):

- 27 and 22 are exchanged, resulting in 22, 27, 56, 45, 79, **95, 96**
- 27 and 56 are not exchanged
- 56 and 45 are exchanged, resulting in 22, 27, 45, 56, 79, **95, 96**
- 56 and 79 are not exchanged

Now the array is 22, 27, 45, 56, **79**, **95**, **96**, and the last three entries are guaranteed to be in their proper order. But, as it turns out, all other numbers have also bubbled into their correct position. One additional pass of the bubbling process would determine that no items need to be exchanged, indicating that the array is now sorted.

This algorithm is somewhat similar to the selection sort algorithm. However, instead of searching a "large" portion of the array for a small element, the algorithm searches locally only. The result of one pass is similar to one pass of selection sort: one element is guaranteed to be at the proper position.

However, the process of bubbling moves other elements closer to their final position as a side benefit and the hope is that fewer passes are necessary to sort the entire array.

We will later compare these algorithms in more detail. Right now, however, we should say that while the name of this sorting mechanism is certainly catchy, and the algorithm is somewhat "cute", it offers no significance performance improvement over selection sort for "average" arrays. To be fair, if the array was originally sorted already, the bubble sort algorithm would detect it and finish almost immediately, while a selection sort would stubbornly re-sort the array. For "average" arrays, however, this procedure offers no significant benefits over a selection sort, and may, in fact, perform worse.

But, it is of course time to implement that algorithm using the appropriate methods:

Example 7.3.12:

Implement the Bubble Sort algorithm and test it on a random array of size 10 to see if it works properly.

As with the selection sort algorithm, we can isolate two distinct tasks:

- We need to perform the "bubbling" procedure and apply it to the array from the first element to a given element.

Therefore, we define a method `bubble` that will look at all consecutive members of an array from index 0 to index `stop` and exchange consecutive items if necessary. As output, this method will return `false` if any exchange was performed or `true` otherwise, indicating whether the array is sorted or not.

- We again need to be able to swap two elements in an array.

We will use our previous method `swap` so we do not have to repeat it here (see example 7.3.11).

Here is a possible `bubble` method, using our previously defined `swap` method:

```
public static boolean bubble(double A[], int stop)
{  boolean sorted = true;
   for (int i = 0; i < stop; i++)
   {  if (A[i+1] < A[i])
      {   swap(A, i, i+1);
          sorted = false;
      }
   }
   return sorted;
}
```

With that method defined (and doing the majority of the work), the rest is simple: the `sortBubble` method needs to call `bubble` first on the full array, then on the array from 0 to the second-last entry, then on the array from 0 to the third-last entry, and so forth. The procedure is done either when a call to `bubble` returns true (indicating that the array is now sorted), or there is no piece left of the array to sort. We use a `while` loop as follows to implement this method:

```
public static void sortBubble(double A[])
{  int stop = A.length-1;
   boolean sorted = false;
   while ( (stop >= 0) && (!sorted))
   {  sorted = bubble(A, stop);
```

```
            stop--;
        }
    }
```

To test this method, simply add both methods to the previous `SortTest` class and modify its `main` method to call `sortBubble` instead of `sortSelect`. The details are left as an exercise (compare example 7.3.18).

■

Insertion Sort

As we mentioned, bubble sort is different from selection sort, but it is not necessarily an improvement. Next, we will introduce a sort algorithm that indeed is a better algorithm that search or bubble.

Definition 7.3.13: Insertion Sort

Suppose `A` *is an array of* `double` *numbers. To sort this array in ascending order using the insertion sort algorithm:*

- *Consider the second last element in the array. If the last element is less than this one, move it up by one and insert the second-last element in the last position.*
- *Consider the third-last element in the array, and the last two elements of the array. Move all entries in that subarray that are less than the third-last element up by one, and insert the third-last element into the resulting empty slot.*
- *Consider the forth-last element in the array, and the last three elements of the array. Move all entries in that subarray that are less than the fourth-last element up by one, and insert the fourth-last element into the resulting empty slot.*

Continue in this fashion until you reach the first element of the array, which is inserted into the appropriate spot of the subarray consisting of all elements but the first one. The array is now sorted in ascending order.

This algorithm is illustrated in figure 7.3.6:

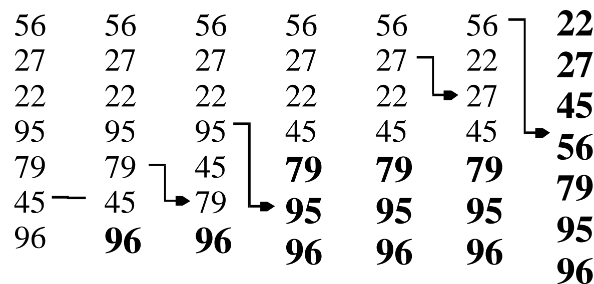


Figure 7.3.6: How Insertion Sort works

- The second-last number is 45, but the last number 96 is bigger, so no action is necessary
- The 3rd-last number is 79, the subarray to consider is [45, 96]. Since 45 is smaller than 79, it moves up one, and 45 is inserted there, resulting in the subarray [45, 79, 96].
- The 4th-last number is 95, the subarray to consider is [45, 79, 96]. Both 45 and 79 are smaller, so they move up one, and 95 is inserted, resulting in [45, 79, 95, 96].
- The 5th-last number is 22. It is smaller than the appropriate subarray [45, 79, 95, 96], so no action is necessary

- The 6th-last number is 27, the subarray to consider is [22, 45, 79, 95, 96]. Only 22 is smaller, so it moves up, 27 is inserted, and the result is [22, 27, 45, 79, 95, 96].
- The 7th-last number is 56, the subarray to consider is [22, 27, 45, 79, 95, 96]. The numbers 22, 27, and 45 are smaller, so they move up by one, 56 is inserted, and the resulting array is [22, 27, 45, 56, 79, 95, 96]

Since the 7th-last number is also the first, we are done, and indeed our array is now sorted.

At every step of this procedure the part of the array that is sorted is located at the tail end of the array, and grows by one at each step. However, only at the very last step are all numbers in their final position. That is substantially different from our previous algorithms:

- in selection sort the appropriate numbers were moved to the top of the array and were immediately in their proper place
- in bubble sort the appropriate numbers bubbled to the bottom of the array were and immediately in their proper place

Despite this apparent shortcoming the insertion algorithm will usually perform better than both of our previous algorithms. In particular, if an array is already sorted correctly, no insertion and no moves will be necessary and the algorithm will finish almost immediately. In other words, this algorithm "detects" sorted arrays, just as the bubble algorithm does. Before we elaborate further, here is the implementation of this algorithm:

Example 7.3.14:

Implement the Insertion Sort algorithm and test it on a random array of size 10 to see if it works properly.

Again, we will try to identify one or more subtasks so that we can implement the appropriate utility methods first. This time we need to be able to move elements of the array up by one and insert another element into the resulting free position. Specifically:

- The method needs, of course, the array as input, as well as an integer `start` indicating which element to look at. That integer also determines which subarray is to be used for the insertion (from `start+1` to the end of the array).
- To determine the spot where the current number is to be inserted, we use a `while` loop. That loop will begin at index position `j = start+1`, move the `j`th element up into the `(j-1)`st position, but only if the current number is larger than the `j`th element. Then `j` is incremented by one. When `j` has reached the end of the array, or when the `j`th element is no longer less than the current element, we have found the position to insert the current element
- As a by-product of the while loop described above, all necessary elements have been moved up by one position, so that this pass of our insertion sort algorithm is complete.

Here is the appropriate method:

```
public static void insert(double A[], int start)
{  double ins = A[start];
   int j = start+1;
   while ((j < A.length) && (ins > A[j]))
   {   A[j-1] = A[j];
       j++;
    }
    A[j-1] = ins;
}
```

Note that at the end the current number `ins` is inserted at `A[j-1]`. That is because the `while` loop will indeed find the correct position to insert our number, but increment `j` one additional time. Therefore, we need to inserted `ins` at `A[j-1]`, not at `A[j]` after the `while` loop.

As usual, after having implemented this utility method doing all the work, the main method to finish our implementation of the insertion sort algorithm is easy: it just calls the `insert` method in a loop that goes backwards through the array, starting at the second-last position of the array. Here are the details:

```
public static void sortInsert(double A[])
{  for (int i = A.length-2; i >= 0; i--)
      insert(A, i);
}
```

To test this implementation of our algorithm, simply add both of these methods to the previous `SortTest` class, and modify its `main` method to call `sortInsert` instead of any other sort method. The details are left as an exercise (compare example 7.3.18).

■

We now have three different sorting mechanisms, but none of them was recursive. Yet, we said earlier that using recursion one can create smart methods that in a few lines of code could accomplish close to magical results.

Quick Sort

This recursion-based new algorithm called Quick Sort will be significantly faster than our previous sorting methods in most cases (as we will see later). In fact, its performance for "average" arrays will be close to the theoretically best possible results. Here is this "quick" recursive sorting procedure:

Definition 7.3.15: Quick Sort

The basic idea of the Quick Sort algorithm is to partition, or rearrange, the array into three distinct pieces: one piece consisting of one element that is in its sorted position, and two subarrays that remain to be sorted. All elements of the first subarray are smaller than the single element, all elements of the other are larger than it. This idea is then applied again to the smaller and to the larger subarrays, until the entire array is sorted. Specifically:

Suppose the algorithm takes two integers as input, labeled `left` *and* `right`*. Then if* `left` *is less than* `right`*, rearrange* `A[left], … , A[right]` *so that there is a* `j` *such that:*

- `A[left], …, A[j-1]` *are less than* `A[j]`
- `A[j+1], …, A[right]` *are greater than* `A[j]`
- *Apply the algorithm again to the array* `A[left], …,A[j-1]`
- *Apply the algorithm again to the array* `A[j+1], …, A[right]`

That does not really specify how exactly the numbers should be rearranged to partition the array correctly, but that is not the point of the algorithm. In fact, different rearrangements are possible and the entire procedure would still be called "Quick Sort". However, to provide a concrete implementation of the Quick Sort algorithm, we do need to specify how this rearrangement can be accomplished:

Definition 7.3.16: The Rearrangement Step in Quick Sort

The Quick Sort algorithm asks for a particular rearrangement of the array. This step can be accomplished as follows:

- *ensure that the* `left` *element is less than or equal to the* `right` *element (swap if necessary)*
- *scan the array starting at* `left+1` *until you find an element bigger than or equal to the* `left` *one and call its index* `j`
- *scan the array starting at* `right-1` *until you find an element less than or equal to the* `left` *one and call its index* `k`
- *if* `j` *is less than* `k`*, swap* `A[j]` *and* `A[k]` *and continue scanning from* `j` *to the* `left` *and from* `k` *to the* `right` *until eventually* `j` *is bigger than* `k` *(after multiple swaps, if necessary)*
- *once* `j` *is bigger than* `k`*, swap* `A[left]` *with* `A[k]`

We have now completed the rearrangement: everything below `A[k]` *is less than or equal to* `A[k]` *and everything above* `A[k]` *is greater than or equal to* `A[k]`*. We can now repeat the process via recursion, as described in the Quick Sort algorithm.*

Here is an illustration of how this rearrangement algorithm works for the array 56, 27, 22, 95, 79, 45, 96:

- initially `left` is 0 (first element) and `right` is 6 (last element)
- 56 is less than 96, so no initial swapping necessary
- we scan up from 27 until we find 95 at index `j = 3`
- we scan down from 45 until we find 45 at index `k = 5` (same value)
- since `j < k` we swap 95 and 45, resulting in the array 56, 27, 22, 45, 79, 95, 96
- we scan up from 45 (`j = 3`) until we find 79 at index `j = 4`
- we scan down from 95 until we find 45 at index `k = 3`
- now `j > k`, so we swap 56 and 45, resulting in 45, 27, 22, 56, 79, 95, 96

This finishes the rearrangement part of the algorithm: we have rearranged the original array into the three pieces [45, 27, 22], [56], and [79, 95, 96], which meet our requirements.

According to the Quick Sort algorithm, we now apply this process recursively first to 45, 27, 22 and then to 79, 95, 96. Let's see what the algorithm will do for the first subarray 45, 27, 22: For this subarray, `left = 0` and `right = 2`.

- 45 is bigger than 22, so we swap them, resulting in 22, 27, 45
- scanning up from 27 we find 27 right there at index `j = 1`
- scanning down from 27 we find 22 at index `k = 0`
- since `j > k` we swap 27 with 22

Again, we are done and we make another recursive call with input arguments 0 and -1 (`= k-1`). But then `left` is bigger than or equal to `right`, so the (empty) base case of the recursion applies and the left subarray is sorted.

Similarly, the algorithm will determine that the right subarray 79, 95, 96 is also sorted, which means in total that the entire array is now sorted indeed.

As before, it's time for the implementation of this algorithm:

Example 7.3.17:

Implement the Insertion Sort algorithm and test it on a random array of size 10 to see if it works properly.

We had luck before with identifying subtasks, so let's try that again:

- We clearly need the ability to swap elements of the array but we have already implemented a `swap` method for that
- The main sorting method *should* take as input only the array of doubles, but the quick sort algorithm requires two integers left and right as input - therefore, we need a utility method called, say, `quickArrange`, that has as input an array of doubles as well as integers `left` and `right`.
- The `quickArrange` method should call itself twice with different parameters, therefore it will be a double recursive method.

So far, therefore, we need as our most important method something like this:

```
public static void quickArrange(double A[], int left, int right)
{  if (left < right)
   {  /* rearrangment code */ }
   quickArrange(A, left, k-1);
   quickArrange(A, k+1, right);
}
```

Next, according to the rearrangement algorithm we may need to swap the `left` and `right` elements of the array at the beginning of the algorithm, set up indices `j` and `k`, repeat some scanning process until the indices `j` and `k` are such that `j > k`, and then swap `A[left]` with `A[k]`. Therefore, our code looks like:

```
public static void quickArrange(double A[], int left, int right)
{  if (left < right)
   {  int j = left + 1;
      int k = right -1;
      if (A[left] > A[right])
         swap(A, left, right);
      while (j <= k)
      {  /* scanning up and down according to the algorithm */  }
      swap(A, left, k);
      quickArrange(A, left, k-1);
      quickArrange(A, k+1, right);
   }
}
```

Note that we are using the swap method we previously implemented. What's left is the actual scanning process:

```
while (A[j] < A[left])
   j++;
while (A[k] > A[left])
   k--;
```

Therefore, the complete method looks like this:

```
public static void quickArrange(double A[], int left, int right)[8]
{  if (left < right)
```

[8] This method is double recursive with an empty base case.

```
    {  int j = left+1;
       int k = right-1;
       if (A[left] > A[right])
          swap(A, left, right);
       while (j <= k)
       {  while (A[j] < A[left])
             j++;
          while (A[k] > A[left])
             k--;
          if (j < k)
             swap(A, j, k);
       }
       swap(A, left, k);
       quickArrange(A, left, k-1);
       quickArrange(A, k+1, right);
    }
 }
```

That leaves the main sorting method, which will simply start the recursive process with the right initial values:

```
public static void sortQuick(double A[])
{  quickArrange(A, 0, A.length-1); }
```

To test this implementation of our algorithm, simply add both of these methods to the previous `SortTest` class, and modify its `main` method to call `sortQuick` instead of any other sort method. The details are left as an exercise (compare example 7.3.18).

■

Now we have four different sorting methods at our disposal and we need to determine which, if any, is the best. If that is difficult, we should at least point out some advantages and disadvantages of either algorithm so that the user can make an informed decision as to which one to use for what job. A few points are obvious:

- Selection sort is the most intuitive one and is easiest to implement.
- Quick sort is a double-recursive method, and hence may be burdened by the overhead and memory needs of such a recursion. It is also the hardest to understand (most double-recursive methods are tricky to figure out)

Some other points are pretty easy to figure out, based on the way the algorithms work:

- The utility method `bubble` returns `true` to `sortBubble` if the array is sorted. Therefore, the bubble sort algorithm will stop after one pass through the array if the original array is already sorted in ascending order. Thus this algorithm "detects" sorted arrays and should handle them very quickly.
- Bubble sorting requires a lot of work at each pass, while it merely *hopes* that more than one element will bubble into its proper position during each pass. Such hopes may be disappointed, in which case the algorithm would expend a lot of work for little result.
- The insertion algorithm does not have to move any elements if the original array is already sorted in ascending order. Therefore, this algorithm also detects sorted arrays and handles them quickly.
- The selection sort will always work the same way, regardless of whether the original array is already sorted or not. It does therefore not detect sorted arrays.

The question of whether quick sort can detect arrays that are already sorted is left as an exercise.

To further compare these algorithms, we could run them on comparable data sets and measure the time the various algorithms take to sort the initial arrays. In fact, we can create a simple program to do exactly that:

Example 7.3.18:

Create a "sort algorithm test" class that executes the various sorting algorithms on data sets of various sizes and measure the time it takes for each algorithm to finish sorting. Create a table with these numbers and create appropriate `xy` graphs to view the results.

We have already implemented all methods necessary for the actual sorting algorithms. This time, we embed these methods into a `Sorter` class that can be useful as a stand-alone class for sorting arrays of doubles. We will add a few constants and methods to make this class a little more useful and relegate the remaining methods (except for `swap`) to private utility methods to hide the complexity of this class from outside view:

```
public class Sorter
{  public static final int SELECTION  = 1;
   public static final int BUBBLE     = 2;
   public static final int INSERTION  = 3;
   public static final int QUICK      = 4;

   public static void sort(double A[], int type)
   {  if (type == SELECTION)
         sortSelect(A);
      else if (type == BUBBLE)
         sortBubble(A);
      else if (type == INSERTION)
         sortInsert(A);
      else
         sortQuick(A);
   }
   public static long sortTimed(double A[], int type)
   {  long start = System.currentTimeMillis();
      sort(A, type);
      long stop  = System.currentTimeMillis();
      return (stop - start);
   }
   /* The remaining methods are exactly as before */
   public static void swap(double A[], int pos1, int pos2)
   private static void sortSelect(double A[])
   private static void sortBubble(double A[])
   private static void sortInsert(double A[])
   private static void sortQuick(double A[])
   private static boolean bubble(double A[], int stop)
   private static void insert(double A[], int start)
   private static void quickArrange(double A[], int left, int right)
   private static int selectMin(double A[], int start)
}
```

We have added a `sortTimed` method that times a sorting routine. It uses the method `currentTimeMillis()` from the `System` class, which returns the current time in milliseconds. That allows us to keep track of the time that sorting a particular array takes. This method is not the most exact, but it will allow us to compare the performances of the various sorting algorithms relative to each other.

The program that will generate a comparison table of execution times for the various algorithms is as follows. The only interesting method here is the `randomArray` method that produces an array of random numbers whose length is specified as input to the method:

```
public class SortPerformance
{  public static double[] randomArray(int n)
   {  double A[] = new double[n];
      for (int i = 0; i < A.length; i++)
         A[i] = Math.random() * 100;
      return A;
   }
   public static void main(String args[])
   {  System.out.println("Datapoints\tSelect\tBubble\tInsert\tQuick");
      for (int n = 1000; n <= 15000; n+=1000)
      {  System.out.print("N = " + n + ":\t");
         for (int type = 1; type <= 4; type++)
         {  double A[] = randomArray(n);
            System.out.print(Sorter.sortTimed(A, type) + "\t");
         }
         System.out.println();
      }
   }
}
```

Rather than showing the times produces by this program (which really depend on the particular computer system used), we have plugged the numbers into a chart-generating package (such as Microsoft Excel) to show a graphical comparison of execution times. The times relative to each other should be similar regardless of the particular system.

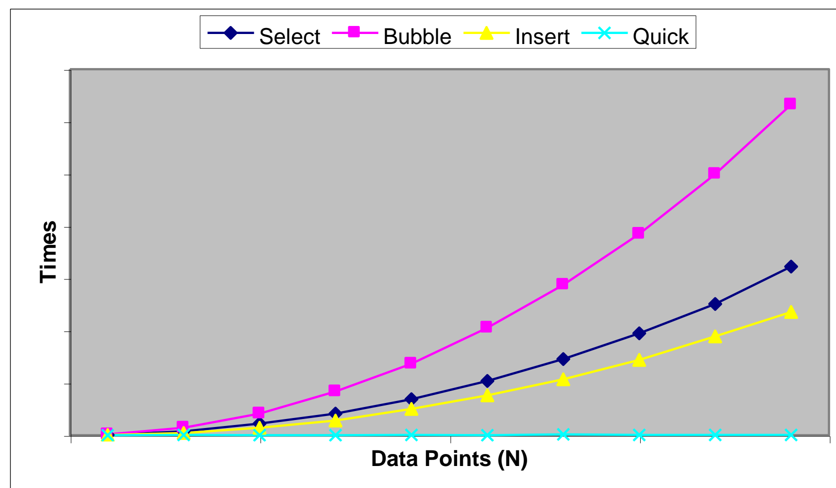


Figure 7.3.7: Sort Algorithm Performance, "random" data

This graph shows execution time to sort `N` random data points with each of our sorting algorithms. It is clear that bubble sort performed the worst, selection and insertion sort performed similarly, with insertion sort being slightly faster. Quick sort, however, is so much faster than the other algorithms that its search times are close to negligible as compared to the other algorithms.

Hence, for "random" data the quick sort algorithm beats the other algorithms hands-down.

■

But that does not mean that this algorithm is the best in every situation. Let's run the program again, but this time the array for whom the sorting time is measured is "almost sorted" instead of in random order. In other words, we modify the above `main` method as follows:

```
public static void main(String args[])
```

```
{   System.out.println("Datapoints\tSelect\tBubble\tInsert\tQuick");
    for (int n = 1000; n <= 15000; n+=1000)
    {   System.out.print("N = " + n + ":\t");
        for (int type = 1; type <= 4; type++)
        {   double A[] = randomArray(n);
            Sorter.sort(A, Sorter.QUICK);
            Sorter.swap(A, 0, A.length-1);
            System.out.print(Sorter.sortTimed(A, type) + "\t");
        }
        System.out.println();
    }
}
```

That means that an array of random numbers is generated, then sorted without measuring the sorting time. Then the first and last elements in the array are swapped, creating an "almost sorted" array. Only *then* is the array sorted again and the sorting time measured. Here are the results, comparing sorting times for such "almost sorted" arrays and the various sort algorithms:

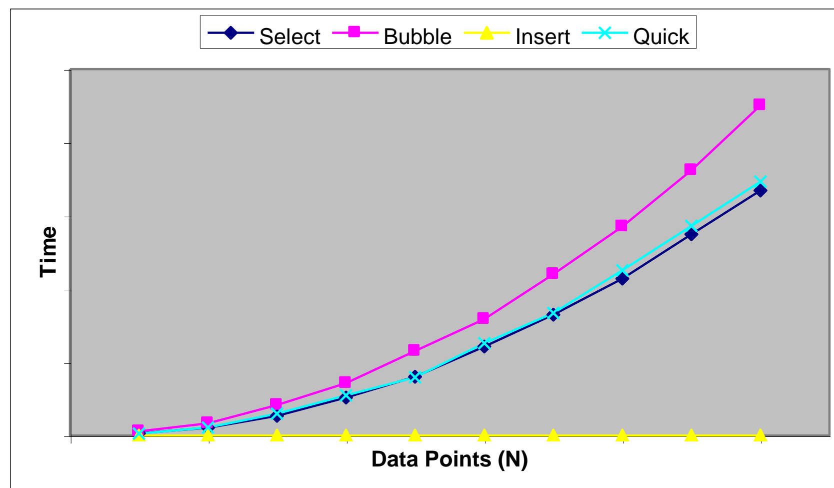


Figure 7.3.8: Sort Algorithm Performance, "almost sorted" data

It is surprising that bubble sort is still worst, because according to our previous discussion it is supposed to detect sorted arrays. But in this case the input array is not *completely* sorted and the bubble sort algorithm fails to take advantage of that. This time, the algorithm that clearly outperforms the others is insertion sort. Our previous champion, quick sort, is now just about comparable to selection sort.

Hence, the perfect sort algorithm really should combine the "detection ability" of insertion sort with the general speed of quick sort. Indeed, one can combine the insertion and quick sort algorithm to produce a "best possible general purpose sorting algorithm". While the details of such an algorithm are best discussed in a course on data structures, Java has - kindly - done the work of actually implementing that algorithm for us.

7.2. Java Build-in Searching and Sorting Algorithms

Since searching and sorting arrays is a frequent task that classes need to perform, Java includes - since version 1.2 - some build-in static methods to accomplish these tasks. These methods are very

convenient and flexible to use and we will see in example 7.4.4 how they compare against our "hand-made" methods and algorithms defined in section 7.3. If you do use these build-in methods, however, your code will *only* run in JVM's that support version 1.2 and better of Java.[9]

Searching and Sorting Basic Data Types

All build-in searching and sorting methods are part of the `java.util.Arrays` class, so we will start with that one:

Definition 7.4.1: Methods in the `Arrays` Class for Basic Data Types

The `Arrays` *class is a utility class for manipulating arrays (such as sorting and searching). It contains, among others, the following* `static` *methods for basic types:*

```
public static boolean equals(type[] a1, int[] a2)
public static void fill(type[] a, int val)
public static void fill(type[] a, int fromIndex, int toIndex, int val)
```

where `type` *can be* `boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, *or* `double`, *as well as the methods*

```
public static void sort(type[] a)
public static void sort(type [] a, int fromIndex, int toIndex)
public static int binarySearch(type[] a, int key)
```

where `type` *can be* `byte`, `char`, `short`, `int`, `long`, `float`, *or* `double`. *The* `binarySearch` *method will return the index where key can be found in the array or a negative number otherwise.*

The sort algorithm for arrays of basic data types is a modified quick sort algorithm. The `Arrays` *class is part of the* `java.util` *package and must be imported*[10].

Note that all methods are static so that you do not need to instantiate an object of type `Arrays` in order to search or sort.

Example 7.4.2:

Let's play a (hypothetical) game: the computer selects 100 random integers between 1 and 100 and stores them in an array. You are asked to pick a number between 1 and 100 as well. If your number is part of the array, you win $10, otherwise you have to pay $10. Would you play?

This question sounds like a probability theory question, which may or may not our strength. But we are good at programming so we will try to use a simulation to find out the chances of winning before deciding whether to play or not. After all, the array could be 100 times the number 39, in which case

[9] See definition 6.2.13 to find out how to ensure that your code is executed by the latest JVM available from Sun, regardless of which version of Java is implement in your favorite web browser.
[10] Note that this class only includes `binarySearch`, not simple search. Therefore, an array must be sorted before we can use the search method of this class.

our chances of winning that would be very small, or it could be the numbers from 1 to 100 so that our chances of winning would be perfect.

We will create a computer program that plays this game many times. Each time we record whether the number we picked is part of the random array. Then we compute the ratio of the number of times our guess was part of the array and the total number of games. That ratio will be our chance of winning. Specifically, we need to perform the following steps:

- we create a random array of size 100 where each element is an integer between 1 and 100
- we select a random number between 1 and 100 (simulating our random guess)
- we search the array for that number (before we can search the array using `binarySearch` we must make sure it is sorted)
- if found, we increment a "found" counter
- we repeat that process, keeping a "total" counter of tries
- the ratio of "found" counter and "total" counter will be our winning chance

Now that we have broken up our task into smaller subtasks it is easy to create the corresponding program:

```
import java.util.Arrays;

public class GuessingChance
{  final static int NUM_GAMES = 10000;
   final static int SIZE = 100;
   public static int[] sortedRandomArray(int size)
   {  int A[] = new int[size];
      for (int i = 0; i < A.length; i++)
         A[i] = (int)(A.length*Math.random()) + 1;
      Arrays.sort(A);[11]
      return A;
   }
   public static void main(String args[])
   {  int total = 0;
      int found = 0;
      while (total < NUM_GAMES)
      {  int A[] = sortedRandomArray(SIZE);
         int number = (int)(A.length*Math.random()) + 1;
         if (Arrays.binarySearch(A, number) >= 0)
            found++;
         total++;
      }
      System.out.print("Size:   " + SIZE + ", ");
      System.out.println("Chance: " + (double)found / total * 100);
   }
}
```

Now let's run this program several times to find the ratio we want.

[11] We need to sort the array so that we can use the `binarySearch` method. It would actually be more economical to not sort the array and use simple search instead of binary search but the `Arrays` class does not include a simple search method.

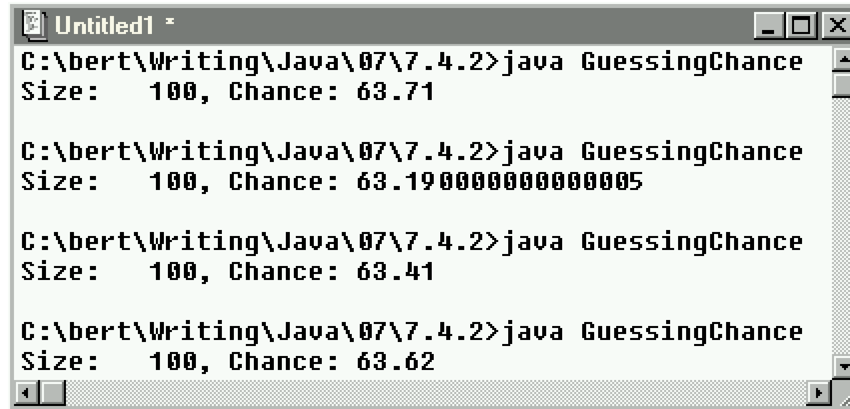


Figure: 7.4.1: Chances of winning $10 in "random array search" game

Based on figure 7.4.1 it seems that our chances of winning are about 63.5%. That's much better than fifty-fifty, so we will definitely play the game.

■

As it turns out, there's an interesting mathematical number hidden in the above example. To find it, let's generalize the game.

Example 7.4.3:

Suppose an array of N integers is generated where each element is a random integer between 1 and N. Now select a random number between 1 and N as well. What is the chance that this number is part of the array? In particular, determine the answer if N gets larger and larger (in math terminology we want to find the limit as N goes to infinity).

In example 7.4.3 we found that chance for N = 100 to be about 62%. Our attempt to find the answer for the more general question would simply be to run the above program for several array sizes and hope to detect a pattern. The code is simple, we just turn SIZE into a variable and adjust it in a loop as follows:

```
import java.util.Arrays;

public class GuessingChance
{  final static int NUM_GAMES = 10000;
   public static int[] sortedRandomArray(int size)
   {  int A[] = new int[size];
      for (int i = 0; i < A.length; i++)
         A[i] = (int)(A.length*Math.random()) + 1;
      Arrays.sort(A);
      return A;
   }
   public static void main(String args[])
   {  for (int size = 2; size < 9182; size *= 2)
      {  int total = 0;
         int found = 0;
         while (total < NUM_GAMES)
         {  int A[] = sortedRandomArray(size);
            int number = (int)(A.length*Math.random()) + 1;
            if (Arrays.binarySearch(A, number) >= 0)
               found++;
            total++;
         }
```

```
            System.out.print("Size:   " + size + ", ");
            System.out.println("Chance: " + (double)found / total * 100);
         }
      }
   }
```

When we execute this program, the chances of winning we get will look similar to those in table 7.4.2. It seems that for different sizes the probabilities are slightly different but they seem to settle on a number slightly above 63% as the array size gets larger.

Size N	Chance
2	75.069%
4	68.487%
8	65.743%
16	64.295%
32	63.663%
64	63.351%
128	63.163%
256	63.577%
512	62.805%
1024	63.051%
2048	63.363%
4096	63.023%
8192	62.985

Table 7.4.2: Winning chances

Some quick and reasonably easy mathematics will allow us to compute the exact probability of winning:

Let's say N = 2. All possible random arrays that could be created are [1, 1], [1, 2], [2, 1], and [2, 2]. The number to guess could be 1, which occurs in 3 out of 4 possible arrays, or it could be 2, which also occurs in 3 out of 4 possible arrays. Therefore, for N = 2 the chance of winning is 3/4 or 75%.

Now let's say N = 4. The array has four elements, each containing a number between 1 and 4. Therefore, there are 4 x 4 x 4 x 4 = 4^4 different array combinations. Take a number, say 2. There are 3 x 3 x 3 x 3 = 3^4 possible array combinations that do *not* include 2 (each element could be 1, 3, or 4), so the rest, namely $4^4 - 3^4$ *will* include 2. Therefore, the chance that 2 is part of the array is $(4^4 - 3^4)/4^4 = 1 - (3/4)^4$. That is the same for every other number so that for N = 5 the chance of winning is $1 - (3/4)^4 = 68.359\%$

In general, the above argument works to show that for an array of size N the chance of winning is $1-\left(\frac{N-1}{N}\right)^N$. You may remember from calculus that $\lim_{N\to\infty}\left(\frac{N-1}{N}\right)^N = e^{-1}$, where e = 2.718281828 is Euler's number (check your calculus textbook). Therefore, we have for larger and larger N that the probability will converge to $1-e^{-1} = .632120$ or 63.2%. Our simulation indeed showed this fairly well.[12]

■

In examples 7.4.2 and 7.4.3 we created a random array that was sorted and searched via the build-in `Arrays` methods sort and `binarySearch`. It would have been more economical to not sort the array and use a simple search, but the `Arrays` class does not include simple search. The program in example 7.4.3 is about as fast as the `Arrays.sort` algorithm, so we would like to know how good that algorithm is compared to our own created in section 7.3.

Example 7.4.4:

[12] This example also gives an interesting algorithm to approximate the value of Euler's number **e**, using only random numbers (and some programming). Run our method for some large N and use the probability you get to approximate **e**. In our case, N = 8192 gave a probability of 0.62985. Since $0.62985 \approx 1 - 1/e$ you can solve this equation for **e** to get $e \approx \frac{1}{1-0.62985} = \frac{1}{.37015} = 2.7016$. That's pretty close to the correct value of e = 2.718281828.

Modify the previous `SortPerformance` class from example 7.3.18 so that it includes the build-in `sort` method from the `Arrays` class to compare the performance of this sorting algorithm to our "hand-made" algorithms from the `Sorter` class. Make sure to run the performance test for random arrays as well as for "almost sorted" arrays.

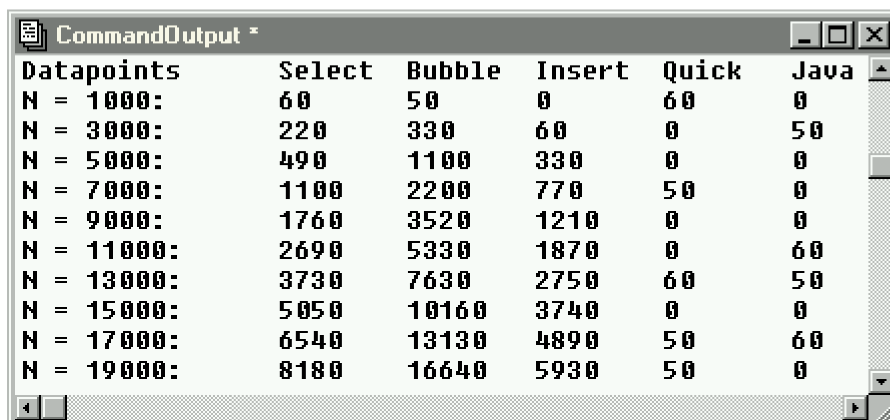
Instead of adding a new sort algorithm to the `Sorter` class, we simply use the build-in method directly and modify the `main` method of the `SortPerformance` class as follows (the new code is in bold and italics):

```
import java.util.Arrays;

/* first part of the class as before */

public static void main(String args[])
{  System.out.println("Datapoints\tSelect\tBubble\tInsert\tQuick\tJava");
   for (int n = 1000; n <= 15000; n+=2000)
   {  System.out.print("N = " + n + ":\t");
      for (int type = 1; type <= 4; type++)
      {  double A[] = randomArray(n);
         System.out.print(Sorter.sortTimed(A, type) + "\t");
      }
      double A[] = randomArray(n);
      long start = System.currentTimeMillis();
      java.util.Arrays.sort(A);
      long stop  = System.currentTimeMillis();
      System.out.println(stop - start);
   }
}
```

This program evaluates the performance for random arrays. The output looks similar to the following (the actual numbers will be different on different computer systems):



Datapoints	Select	Bubble	Insert	Quick	Java
N = 1000:	60	50	0	60	0
N = 3000:	220	330	60	0	50
N = 5000:	490	1100	330	0	0
N = 7000:	1100	2200	770	50	0
N = 9000:	1760	3520	1210	0	0
N = 11000:	2690	5330	1870	0	60
N = 13000:	3730	7630	2750	60	50
N = 15000:	5050	10160	3740	0	0
N = 17000:	6540	13130	4890	50	60
N = 19000:	8180	16640	5930	50	0

Figure 7.4.3: Comparing "hand-made" `sort` *with* `java.util.Arrays.sort` *for random arrays*

Thus, the build-in algorithm for `java.util.Arrays.sort` is comparable to our `QuickSort`. That is no surprise because both are using an algorithm based on `QuickSort`.

Now let's compare sort performance for almost sorted arrays. As before, we adjust the `main` method of the `SortPerformance` class so that it generates a random array, then we sort it without timing it. Then we swap the first and last elements to create an almost sorted array and we time our algorithms using that array:

```
public static void main(String args[])
```

```
{  System.out.println("Datapoints\tSelect\tBubble\tInsert\tQuick\tJava");
   for (int n = 1000; n <= 15000; n+=2000)
   {  System.out.print("N = " + n + ":\t");
      for (int type = 1; type <= 4; type++)
      {  double A[] = randomArray(n);
         java.util.Arrays.sort(A);
         swap(A, 0, A.length-1);
         System.out.print(Sorter.sortTimed(A, type) + "\t");
      }
      double A[] = randomArray(n);
      java.util.Arrays.sort(A);
      Sorter.swap(A, 0, A.length-1);
      long start = System.currentTimeMillis();
      java.util.Arrays.sort(A);
      long stop  = System.currentTimeMillis();
      System.out.println(stop - start);
   }
}
```

```
CommandOutput *
C:\>java SortPerformance
Datapoints      Select  Bubble  Insert  Quick   Java
N = 1000:       0       0       0       0       0
N = 3000:       170     270     0       170     0
N = 5000:       490     720     0       440     0
N = 7000:       990     1420    0       1260    60
N = 9000:       1650    2200    0       1540    0
N = 11000:      2470    3410    0       2310    0
N = 13000:      3460    4780    0       3190    0
N = 15000:      4560    6320    0       4340    0
```

Figure 7.4.4: Comparing "hand-made" `sort` *with* `java.util.Arrays.sort` *for almost sorted arrays*

This time, `java.util.Arrays.sort` is comparable to our insertion sort algorithm. But now the build-in algorithm has an advantage over the ones we created: it performs best for random arrays *as well as* for almost sorted arrays. Thus, this algorithm includes the advantages of both `InsertionSort` and `QuickSort` without inheriting their weaknesses.

■

A more thorough performance analysis of these algorithms would involve "big-O" notation, which measures the order of growth of an algorithm independent of machine-specific considerations such as processor speed and available memory. For details on "big-O" please refer to the appendix. For completeness, here is an overview of the performance of our algorithms using this mathematical notation.

Definition 7.4.5: Performance of Sort Algorithms in Big-O Notation

The sort algorithms introduced in this chapter have the following order of growth:

	Average Case	Best Case	Worst Case
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble Sort	$O(n^2)$	$O(n)$	$O(n^2)$
Insertion Sort	$O(n^2)$	$O(n)$	$O(n^2)$
Quick Sort	$O(n \log(n))$	$O(n \log(n))$	$O(n^2)$

Java Build-in	O(n log(n))	O(n log(n))	O(n log(n))

In addition:

- *Selection Sort performs lots of comparisons and few moves. It is best suited for data that compares quickly.*
- *Insertion Sort performs lots of moves and fewer comparisons. It is best suited for data that can be moved quickly.*
- *Bubble Sort does not have many advantages and is generally the least desirable algorithm.*
- *Quick Sort is recursive and general memory considerations associated with recursion apply. It can, however, be improved to yield the best possible general purpose sorting algorithm known today.*
- *Java's build-in algorithm is flexible and fast but requires Java version 1.2 or better.*

Note: The theoretically best possible sort algorithm can not perform better than O(n log(n)) on average.

Based on our performance analysis it seems that the `java.util.Arrays.sort` method is preferable over our own methods, unless there are specific reasons for creating a custom sorting algorithm. In fact, there's another reason why the build-in algorithm is quite handy: it can easily handle sorting (and searching) arrays of objects in addition to the basic data types.[13] To be sure, in the exercises you will be asked to modify "our" sort algorithms so that they, too, can handle arrays of type `Object`.

Searching and Sorting Objects

Before we can sort (or search for) objects we need a mechanism to compare objects, i.e. we need to be able to decide whether two objects are considered equal, or which one is considered larger than the other. After all, our algorithms all need to compare the elements in the array to be searched or sorted. For basic data types the standard comparison operators ==, <, and > work just fine, but for the type `Object` Java provides the `Comparator` interface for that purpose. That interface requires two methods to be implemented:

Definition 7.4.6: The `Comparator` Interface

The `Comparator` *interface is intended to impose an ordering on a collection of objects. It is frequently used to sort or search for objects in an array. The ordering imposed by a* `Comparator` *should satisfy the following conditions:*

- *Consistent:* `(compare(x,y) == 0)` *if and only if* `(x.equals(y))`
- *Anti-Reflexive:* `sgn(compare(x,y)) == -sgn(compare(y,x))` *for all* `x` *and* `y`
- *Transitive:* `(compare(x,y)>0) && (compare(y,z)>0)` *implies* `(compare(x,z)>0)`

The `Comparator` *interface is part of the* `java.util` *package and must be imported. The Java API defines it as follows:*

```
public abstract interface Comparator
```

[13] It may still be useful to implement your own searching and sorting methods if you want to ensure that your code executes in *every* JVM, not only in those supporting the build-in methods (version 1.2 and better).

```
{  public int compare(Object o1, Object o2)14
   public boolean equals(Object obj)15
}
```

Such a `Comparator` is similar to our `Searchable` interface from example 7.3.7 but uses only one method returning an integer instead of three methods returning a `boolean`. In addition, our `Searchable` interface needs to be implemented by objects that want to be searched or sorted. The `Comparator` interface, on the other hand, is used in a separate class and applied *to* objects rather then implemented *by* objects. The specific syntax for search and sort methods using a `Comparator` is given in the next definition:

Definition 7.4.7: Methods in the `Arrays` Class for Object Types

The `Arrays` *class mentioned in definition 7.4.1 also contains the following* `static` *methods to search and sort* `Object` *arrays:*

```
public static void fill(Object[] a, Object val)
public static void fill(Object [] a, int fromIndex, int toIndex,
                        Object val)
public static void sort(Object[] a, Comparator c)
public static void sort(Object [] a, int fromIndex, int toIndex,
                        Comparator c)
public static int binarySearch(Object [] a, Object key, Comparator c)
```

The sort algorithm for arrays of `Object` *types is a modified merge sort algorithm. The class is part of the* `java.util` *package and must be imported.*

Note that `sort` and `binarySearch` take a *separate* `Comparator` object as input. Another possibility would be to require the objects to be searched or sorted to implement the `Comparator` interface directly. Either approach would work fine but the approach chosen by Java does not require changing an existing class just to sort its objects.

Example 7.4.8:

Say we have an array consisting of elements of type `double`, `int`, or `String`. We want to sort that array so that all strings occur first, then all doubles, and finally all integers. Within their own types the elements should be sorted as usual. Create a sample program to accomplish that. Also illustrate how to search the array once it is sorted.

Obviously we can not have an array containing `double`, `int`, and `String` types so we need to use the wrapper types `Double` and `Integer` before storing `Double`, `Integer`, and `String` in the required array. The method to use for sorting is `Arrays.sort`, which requires an objects array as well as a `Comparator` as input. The rules for comparing the objects must be implemented in the `Comparator` class according to the following scheme:

- if 1st object is a `String` then:
    - if 2nd object is not a `String`, 1st object is smaller than 2nd

[14] If `compare(o1, o2) < 0`, then `o1` is less than `o2`. If `compare(o1, o2) == 0` then `o1.equals(o2)`. If `compare(o1, o2) > 0`, then `o1` is bigger than `o2`.

[15] Every Java class extends `Object` and `Object` already contains an `equals` method. By default it checks whether two objects are stored at the same memory location. You can override this method if your objects implement `Comparator` to have control over when exactly two objects are considered equal.

    - if 2nd object is also a `String`, compare them as `String`
- if 1st object is a `Double` then:
    - if 2nd object is a `String`, 1st one is bigger
    - if 2nd object is also a `Double`, compare them as `Double`
    - if 2nd object is an `Integer`, 1st one is smaller
- if 1st object is an `Integer` then:
    - if 2nd object is also an `Integer`, compare them as `Integer`
    - if 2nd object is not an `Integer`, 1st one is smaller

Therefore we create a class that implements `Comparator` as follows:

```
import java.util.Comparator;

public class SortRule implements Comparator
{  public boolean equals(Object obj1, Object obj2)
   {  return obj1.equals(obj2); }
   public int compare(Object obj1, Object obj2)
   {  if (obj1 instanceof String)
      {  if (obj2 instanceof String)
            return ((String)obj1).compareTo((String)obj2);
         else
            return -1;
      }
      else if (obj1 instanceof Double)
      {  if (obj2 instanceof String)
            return 1;
         else if (obj2 instanceof Double)
            return ((Double)obj1).compareTo((Double)obj2);
         else
            return -1;
      }
      else if (obj1 instanceof Integer)
      {  if (obj2 instanceof Integer)
            return ((Integer)obj1).compareTo((Integer)obj2);
         else
            return 1;
      }
      else
         return -1;
   }
}
```

The compare method follows closely the above outline, using the `instanceof` operator to check the type of `obj1` and `obj2`. Once the type of the input objects has been identified we use the `compareTo` method to compare like objects. While `String`, `Double`, and `Integer` all have a `compareTo` method, we can not use polymorphism because the common superclass of these classes, `Object`, does not have that method.

The `equals` method, on the other hand, is easy to implement: all three of our classes own that method, which they inherit from `Object`. Therefore we can let polymorphism decide which `equals` method is appropriate and return its result as the return value of the `Comparator` method `equals`.

Once the class implementing the particular rules for comparing objects is implemented it is exceedingly easy to sort and search arrays according to these rules:

```
import java.util.Arrays;

public class SortingObjects
```

```
{  public static Object[] createObjectArray()
   {  Object obj[] = {"Sorting is easy", new Double(-10),
                      new Integer(-10),
                      "Java by Definition", new Double(10),
                      new Integer(10),
                      new Double(-10), new Integer(-10) };
      return obj;
   }
   public static void print(Object obj[])
   {  for (int i = 0; i < obj.length-1; i++)
         System.out.print(obj[i] + ", ");
      System.out.println(obj[obj.length-1]);
   }
   public static void main(String args[])
   {  Object obj[] = createObjectArray();
      SortRule rule = new SortRule();
      System.out.println("BEFORE:");
      print(obj);
      Arrays.sort(obj, rule);
      System.out.println("SORTED:");
      print(obj);
      Double d = new Double(10);
      Float  f = new Float(10);
      System.out.print("Is 'Double(10)' part of the array? ");
      System.out.println((Arrays.binarySearch(obj, d, rule) >= 0));
      System.out.print("Is 'Float(10)' part of the array? ");
      System.out.println((Arrays.binarySearch(obj, f, rule) >= 0));
   }
}
```

Figure 7.4.5 shows the result of searching and sorting[16] this array.

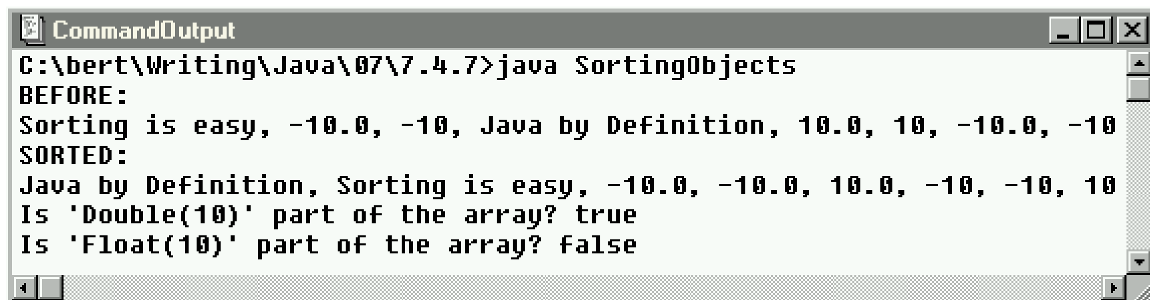


Figure 7.4.5: Results of executing SortingObjects

■

As our final example, let's use the `Arrays.sort` method and an appropriate `Comparator` to sort Address objects similar to example 7.3.8.

Example 7.4.9:

Create a simple `Address` class with fields for first and last name and email address (compare example 7.3.8). Then create an array of four addresses (or so) in some random order. Finally, sort the array so that the addresses appear in (a) increasing and (b) decreasing order.

[16] Of course we can only apply `binarySearch` because our array has been sorted. Note also that while the value of `f = new Float(10)` is part of the array, the object `f` is not.

By now we have seen the Address class plenty of time - perhaps it is time to pick on a different example - so it's just listed for easy reference (we have marked the fields as protected for easy access):

```
public class Address
{  protected String first, last, email;
   public Address(String _first, String _last, String _email)
   {  first = _first;
      last  = _last;
      email = _email;
   }
   public String toString()
   {  return last + ", " + first + " (Email: " + email + ")"; }
}
```

In order to sort arrays of addresses we need a Comparator class that will determine the ordering of addresses. In our case we decide that the order of addresses should be determined by the lexicographical order of last names, then by first names if two last names agree (compare to the Address class implementing the Searchable interface in example 7.3.8). Before creating the corresponding Comparator class, however, we need to override the default equals method that the Address class inherits from Object so that we can ensure consistency with the Comparator class. Thus, we add the following method to the Address class before proceeding:

```
public class Address
{  protected String first, last, email;
   public Address(String _first, String _last, String _email)
   {  /* as before */ }
   public String toString()
   {  /* as before */ }
   public boolean equals(Object obj)
   {  if (obj instanceof Address)
      {  Address a = (Address)obj;
         return ((last.equals(a.last)) && (first.equals(a.first)));
      }
      else
         return false;
   }
}
```

Now we can create a Comparator that is designed so that it designates two addresses as equal exactly when one address would declare another one as equal. Otherwise, we'll use lexicographical ordering of the last and first names to determine when one address is smaller or larger than another:

```
import java.util.*;

public class AddressCompare implements Comparator
{  public boolean equals(Object obj1, Object obj2)
   {  return obj1.equals(obj2); }
   }public int compare(Object obj1, Object obj2)
   {  if (!((obj1 instanceof Address) || (obj1 instanceof Address)))
         return false;
      else
      {  Address a1 = (Address)obj1;
         Address a2 = (Address)obj2;
         if (a1.last.compareTo(a2.last) < 0)
            return -1;
         else if (a1.last.equals(a2.last))
            return (a1.first.compareTo(a2.first));
         else
```

```
                return +1;
            }
        }
    }
```

In fact, we have used the `compareTo` method of the `String` class to impose an ordering on addresses so that addresses are ordered in increasing order. We could just as well design a `Comparator` that imposes an ordering in decreasing order. This is easy to do by just "reversing" the previous `Comparator`:

```
import java.util.*;

public class AddressCompareReverse extends AddressCompare
{  public int compare(Object obj1, Object obj2)
   {  return -super.compare(obj1, obj2); }
}
```

Now we can easily sort an array of addresses in either increasing or decreasing order as follows:

```
import java.util.*;

public class SortTest
{  public static void printAddresses(String msg, Address data[])
   {  System.out.println(msg);
      for (int i = 0; i < data.length; i++)
         System.out.println("\t" + data[i]);
   }
   public static void main(String args[])
   {  Address data[] = {
            new Address("Bert", "Wachsmuth", "wachsmut@shu.edu"),
            new Address("John", "Doe", "doejohn@somewhere.edu"),
            new Address("Jane", "Doe", "doejane@somewhere.edu"),
            new Address("Rip", "van Winkle", "rwinkle@somewher.edu")};
      printAddresses("Original order:", data);
      Arrays.sort(data, new AddressCompare());
      printAddresses("Sorted ascending:", data);
      Arrays.sort(data, new AddressCompareReverse());
      printAddresses("Sort descending:", data);
   }
}
```

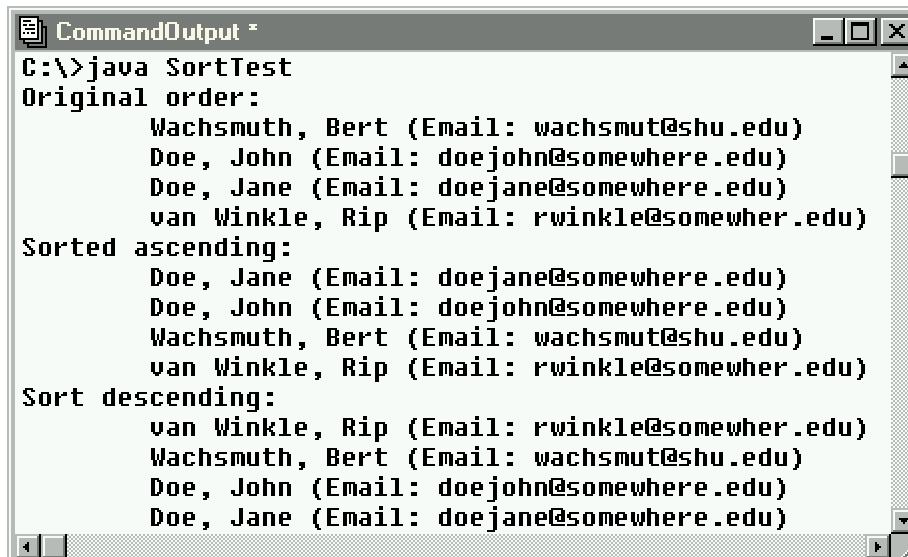


Figure 7.4.6: Results of sorting addresses

■

Chapter 8: Abstract Data Types

In chapter seven we focused on recursion, searching, and sorting. These techniques really do not take advantage of the fact that Java is an object-oriented programming language. That is particularly evident since the methods we introduced in that chapter were static methods and do not require instantiation of an object. In this chapter we also want to focus on general programming techniques that can be implemented in a variety of languages, but this time we will make extensive use of objects.

In section 8.1 we will introduce data storage structures such as lists, queues, and stacks that will be welcome additions to arrays, the only mass-storage structure we know so far.[17] The structures we will cover will be dynamic, i.e. they will only use as much memory as necessary at any given time, and flexible, i.e. they will form "reusable" classes that can be used without changes in many situations. Such dynamic structures form the basis for most commercial programs and a thorough understanding of them is essential for any successful programmer.

In section 8.2 we will explain how to use the structures created in 8.1 to implement a mathematical expression parser, i.e. a class that can translate and evaluate mathematical expressions such as "`8*(14.4-99.3)^2 + (9.3-1.1)*(9.3+1.1)`". While this section is not necessary to understand lists, stacks, and queues, it provides an interesting and useful application that serves to illustrate the usefulness of our structures as well as the power of object-oriented programming. The example is rather long and could be skipped at first reading of this chapter.

In section 8.3 we will introduce the sequential structures that are build-in to Java since version 1.2. Just as Java includes ready-made searching and sorting methods[18], it also includes ready-to-use sequential structures such as lists and stacks (which are, to be sure, quite extensive and complete). Section 8.1 will form the theoretical background to this section, but if you only want to learn about *using* dynamic structures you could jump ahead to section 8.3 right away and return to 8.1 and 8.2 at a later time. If you do use these build-in classes your code will only run in JVM's that support Java version 1.2 or better.[19]

The purpose of this chapter is *not* to be a complete discussion of abstract data types. That will be left to a "Data Structures" course. Instead it will illustrate how some common data structures such as List, Stack, and Queue can be implemented in the Java language, and give several examples of their use.

Please note: several of the examples and exercises will compare the performance of various classes. It would be a good idea to review the "order of growth" of a method as defined in the appendix.

---

[17] To be picky, in chapter 6 we have seen data storage structures other than arrays, namely the models associated with various views. In this section we will explain in detail how these models can store data dynamically, allocating only as much memory as necessary.
[18] See section 7.4
[19] One particular dynamic structure called `Vector` is part of Java since the beginning and is therefore supported by any JVM.

7.3. Sequential Data Structures

The structures we will introduce in this section are examples of a general concept called abstract data types. We already know that Java contains a variety of basic or atomic data types (`double`, `int`, `char`, etc) and new types can be created simply by designing a class. Abstract data types (ADT's) describe conglomerates of atomic, or basic, data types that are used to store data, endowing it with some type of internal structure and with operations to manipulate the data.

Definition 8.1.1: Abstract Data Type, or ADT

An Abstract Data Type, or ADT, is a specification of a data structure in abstract terms, without reference to programming language or machine dependent terminology. Abstract data type specifications contain two parts:

- *a description of the data together with an internal structure of the data*
- *a description of the valid (and invalid) operations that can be performed on this data*

Abstract data types, by nature of being abstract, can not be used directly in a program, but they can be implemented into concrete instances, which then can be used in programs.

Abstract Data Types are typically described in Java by an appropriate interface. That interface describes as comments the data and its structure and lists as abstract methods the valid operations on that data. A concrete implementation of an ADT and its corresponding interface can then be provided by one or more classes. In our case we will for simplicity use classes to realize abstract data types directly, skipping the step of first defining an interface for an ADT.

To ensure that the conditions that will lead to successful applications of the various operations described in an ADT are satisfied, a Java class implementing an ADT can - and should - make use of the exception mechanism.

The data structure imposed by an ADT can take many different forms. Data can be organized sequentially, hierarchically, in clusters, according to "size", etc. The structures that we will focus on in this chapter are sequential structures, i.e. data that is arranged in such a way that individual data items have one successor, one predecessor, or both.[20] Other structures (for example hierarchical structures such as "trees") are left to courses specifically focusing on data structures and would go beyond the scope of this text.

Lists

Until now the only data structure we had to store large amounts of data was an array, either one or multi-dimensional. Arrays, however, are not flexible enough in many situations because they are declared to have a fixed size. Of course the size could be at runtime, but once declared the size of an array is fixed. That leads to two problems:

[20] In section 6 we have already seen a non-linear data structure: `javax.swing.JTree` in some detail, but we have not provided a theoretical framework for such hierarchical structures.

- Waste of memory: an array can be declared to be of a large size, but only a small part of it is used. The unused part of the array occupies memory that is not available to be allocated by the JVM.
- Lack of memory: an array is declared to be of some size, but in the course of a program it turns out that it needs to be larger. One could create a new and larger array and copy the elements from the old array into the new one, but that takes precious computing time.

Example 8.1.2:

Outline the data structure you would use to create a simple text-editing program. Discuss the advantages and disadvantages of your choice(s).

Leaving aside the (non-trivial) details of what a text editing program should be able to do, it needs, in essence, to be able to store and manipulate characters. To store these character we have two choices:

- we could use an array of characters
- we could use a string

If we used an array of characters we would have to declare it at the beginning of our program. Thus, we would need to specify a maximum number of characters our text editor could handle. That is the approach that was chosen for simple text editors[21] such as Microsoft Windows Notepad, which can handle up to approximately 32,000 characters. This could be accomplished by introducing a field in the text editor class like:

```
char letters[] = char[32000];
```

Even if we did not mind this size limitation, we would occupy 32,000 bits of memory even if the actual text file we are working on contains only 10 characters. That, clearly, is not an optimal solution.

If we used a String, we would have the advantage that a String can grow (and shrink) dynamically, as needed. However, the particular methods that the String class offers may or may not be the most useful set of methods for our particular situation. In addition, we don't know whether there is a performance penalty in using a String, because we don't really know *how* a String object manages to grow and shrink.[22] In particular, if at one time our String was very large, but then needed to shrink to a small size (because, say, of a "cut" operation in our editor) we don't know whether any memory is really returned to the JVM at that time. Therefore, a String is may not be an optimal solution either.

■

Even if a String would work as a dynamic structure for characters, it could not be used as a general dynamic storage structure for other data types. Therefore, we are in need of a new structure with the following features:

- the structure should always occupy as much memory as necessary
- the structure should be able to grow at any time, and accomplish its growth quickly
- the structure should be able to shrink at any time and return unused memory to the JVM

---

[21] The `java.awt.TextField` actually has the same constraint: it can only hold up to about 32,000 characters. The `javax.swing.JTextField`, on the other hand, has no such limitations due to the model/view split it uses.

[22] As a matter of fact a `String` is not useful at all for dynamically storing characters. Another class called `StringBuffer` should be used for strings that vary in length, while `String` is reserved for strings of fixed size. In our programs most strings were more or less fixed so we did not have much use for `StringBuffer`.

- the structure should be ordered so that every element has a well-defined location

Before worrying about how to actually implement such a structure, we will formalize it in an abstract way and call such a structure a *list*.

Definition 8.1.3: ADT List

A list is an ordered structure of elements such that it is either empty or it has a unique first element, every element except for the last one has one immediate successor, and the last element is that one without a successor. One element in a non-empty list is always designated as the current *element. In addition, a list should provide the following operations:*

- *create a empty list*
- *designate the first element of a non-empty list as the current element*
- *move the current element to the successor of the current element in an non-empty list, unless the current element is the last*
- *insert an element into the list*
- *retrieve the element designated as current, unless the list is empty*
- *remove the current element from the list, unless the list is empty*
- *replace the current element in the list by another element, unless the list is empty*
- *check whether a list is empty*
- *check whether the current element is the last element*

If we had such a structure, we could easily see how to use it for a text editor program. The list would store characters, it could grow at any time via the *insert* operation, and it could shrink at any time via the *remove* operation (which would return any unused memory to the JVM). We could access any element in the list by first moving to the first one, then moving through as many successors as necessary to eventually retrieve the element we want. Based on those operations we could implement more advanced operations such as "replace a group of characters", or "save all characters to disk", and so forth.

Since the above list structure is defined without reference to any implementation-specific details it is an abstract data type. To be useful we need to provide an implementation of this ADT into a *concrete* data type and we will naturally use a class for that, bypassing an interface. To model our abstract data type list we will hide all necessary implementation details from the user and we will define the visible part of a List class as follows:

Definition 8.1.4: List class

A `List` *is either empty or a sequential structure of elements such that one element is always designated as the* current *one. The* `List` *class supports the following methods:*

- `constructor`*: creates an empty* `List` *containing no elements*
- `public void first()`*: makes the first element the current one. Exception: can only be used if the list is not empty*
- `public void next()`*: makes the element after the current one the (new) current one. Exception: can only be used if the* `List` *is not empty, and the current element is not the last*
- `public void insert(Object o)`*: inserts an element, or object, into the* `List`

- `public Object retrieve()`*: returns the element designated by current. Exception: can only be used if the* `List` *is not empty*
- `public void remove()`*: deletes the current element from the* `List`*. Exception: can only be used if the* `List` *is not empty*
- `public void replace(Object o)`*: replaces the current element by another one. Exception: can only be used if the* `List` *is not empty*
- `public boolean isEmpty()`*: returns* `true` *if the* `List` *is empty,* `false` *otherwise*
- `public boolean isLast()`*: returns* `true` *if the current element is the last one,* `false` *otherwise*

The next step would be to find a concrete realization of this class, i.e. to specify the code that makes the class work as specified. There are two possible standard solutions in Java:

- The first one uses an array that is suitably modified during an `insert` and `remove` operation. We will leave this solution (mostly) to the exercises.
- The second approach uses a construction that wraps the elements to be stored in a class called a `Node`

A list is supposed to store data such that each data element contains a successor. Therefore, elements in a list really consist of two pieces: the data and a reference to the successor. A `Node` is a utility class that models this situation. It should be comprised of two pieces: one piece to store a Java object, and another piece to store a reference to another `Node`.

Definition 8.1.5: The `Node` class

A `Node` *is a class that can store an arbitrary Java* `Object` *such that:*

- *it can return and replace the* `Object` *stored*
- *it can contain a reference to another* `Node` *called the successor*

In our implementation, a `Node` *contains two non-private fields, one called* `data` *that stores the actual object and another called* `next` *to store a reference to another* `Node` *object, if any, or* `null`*:*

```
class Node
{  protected Object data;
   protected Node   next;
   protected Node()
   {  data = null;  next = null; }
   protected Node(Object _data)
   {  data = _data; next = null; }
}
```

We have marked the fields `data` and `next` as `protected` so that they can be used by classes in the same package (or directory) as the `Node` class. We could have chosen to mark all fields as private and implement the appropriate `set` and `get` methods, but for simplicity we will use the `protected` modifier.

A `Node` can be pictured as follows, where we denote a reference to null by an electrical "grounding" symbol:

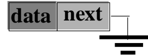


Figure 8.1.1: A properly grounded Node

Example 8.1.6:

Write a code segment that creates three nodes `A`, `B`, and `C`, that form a circular reference. Node `A` should point to `B`, `B` should point to `C`, and `C` should point back to `A`. In the data piece of nodes `A`, `B`, and `C` store the strings "`Java`", "`by`", "`Definition`", respectively. Then check the output produced by the statement:

```
System.out.println(A.next.next.next.data);
```

First, let's consider the corresponding picture: we want to create three nodes that look as follows:

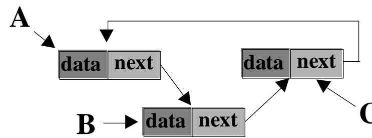


Figure 8.1.2: Three nodes arranged with circular references

Therefore, our code needs to instantiate three new nodes, then adjust the `next` references of each `one` to point to the correct node in turn. Without worrying about storing the `data`, the code would be:

```
Node A = new Node();
Node B = new Node();
Node C = new Node();
A.next = B;
B.next = C;
C.next = A;
```

This code will indeed create the nodes arranged with circular references. Next, we need to store the appropriate strings in the `data` piece of each node by using the second from of the constructor and call on the `System.out.println` method to see the desired output:

```
public class CircularNodes
{  public static void main(String args[])
   {  Node A = new Node(new String("Java"));
      Node B = new Node(new String("by"));
      Node C = new Node(new String("Definition"));
      A.next = B;
      B.next = C;
      C.next = A;
      System.out.println(A.next.next.next.data);
   }
}
```

Since `A.next` points to `B`, we have `A.next.next.next.data = B.next.next.data`, and since `B.next` equals `C`, we have `B.next.next.data = C.next.data`. Finally, `C.next` is the same as `A`, so that `C.next.data` = `A.data`. Therefore, the output will be the string "`Java`", which will be confirmed by running the above program (the `Node` class must be saved to the same directory before compiling and executing this class).

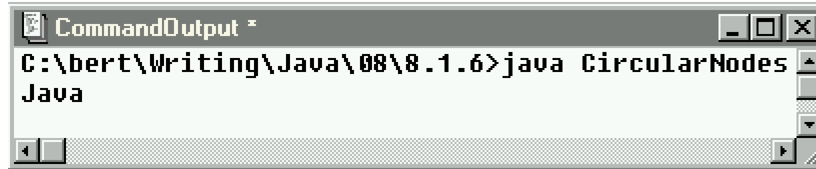


Figure 8.1.3: Output of `CircularNodes` *program*

■

Nodes can be tied together to form a sequential structure that could represent our List class we want to create:

Figure 8.1.4: A sequential structure consisting of several nodes tied together

However, for this list to be valid, one of the nodes must be designated as the `current` node, and we also need to have a reference to the `start` of the list in order to access its nodes. Hence, our new picture of a `List` using nodes looks as follows:

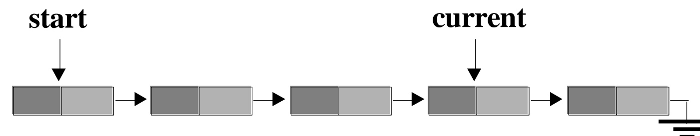


Figure 8.1.5: Image of a non-empty `List`

In other words, a `List` really consists of two reference variables `start` and `current`, and any number of nodes which on their own point to their successor, if any, or to `null` for the last node. The nodes, however, are not stored internally in the list class, because once we have access to the first node (via the `start` reference) we can reach all nodes by using the `next` references of each node.

Once we have a picture in mind we should be able to implement the appropriate class:

Example 8.1.7:

Create a concrete implementation of a `List` class. If an exception is necessary to define particular methods, create it as well.

First, let's reconsider the conditions that ensure success of the various operations:

- `first()` requires a non-empty list
- `next()` requires a non-empty list and the current element should not be the last
- `retrieve()` requires a non-empty list
- `remove()` requires a non-empty list
- `replace(Object o)` requires a non-empty list

Therefore, there are two different kinds of exceptions: one to indicate that a list is empty and another one to indicate that the `current` element is the last element. We could create a hierarchy of exceptions with, say, a `ListException` as the base class and two other exceptions as subclasses. For simplicity, however, we will create only one new exception class and use it to display the corresponding error messages for the two types of possible errors:

```
public class ListException extends Exception
{  public ListException()
```

```
    {  super("List Exception"); }
    public ListException(String msg)
    {  super(msg); }
 }
```

With this exception in place we can now implement our `List` class. After all, we know the methods it should contain and from the above picture we have determined that the class needs exactly two fields, `start` and `current`. Here is the basic structure of the class, without implementing the methods:

```
public class List
{  private Node start, current;

   public List()
   public void first() throws ListException           { }
   public void next() throws ListException            { }
   public void insert(Object o)                       { }
   public void remove() throws ListException          { }
   public Object retrieve() throws ListException      { }
   public void replace(Object o) throws ListException { }
   public boolean isEmpty()                           { }
   public boolean isLast()                            { }
}
```

Note that the fields `start` and `current` are private so that they are not visible to the user of the class. Also, no reference is made to any sequence of nodes internal to the list. First, let's consider the image of an empty list:

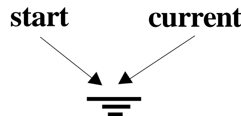


Figure 8.1.6: Picture of a valid empty List

That picture can be translated into the corresponding `List` constructor code:

```
public List()
{  start = current = null; }
```

Based on that picture and code we can easily implement the `isEmpty` method:

```
public boolean isEmpty()
{  return (start == null); }
```

Next, let's try the `isLast` method: it should return `true` if the `current` element of the list is the last element. We have defined the last element as that one without a successor, or that one whose `next` reference is `null`. Hence, our method needs to check whether the `next` reference of the node referenced by `current` is pointing to `null`. Therefore, our method looks like:

```
public boolean isLast()
{  if (current != null)
      return (current.next == null);
   else
      return false;
}
```

Note that this method returns `true` if the `current` node is the last node (i.e. `current.next` points to `null`), or if the list is empty (i.e. `current` points to `null`). It does not throw an exception.

The `retrieve` and `replace` methods are also easy to implement. They simply return or modify the `data` piece of the node pointed to by `current`:

```
public Object retrieve() throws ListException
{  if (start == null)
      throw new ListException("retrieving data from empty list");
   else
      return current.data;
}
public void replace(Object obj) throws ListException
{  if (start == null)
      throw new ListException("replacing data in empty list");
   else
      current.data = obj;
}
```

The "cursor movement" operations `first` and `next` are also simple to implement:

```
public void first() throws ListException
{  if (start == null)
      throw new ListException("finding first in empty list");
   else
      current = start;
}
public void next() throws ListException
{  if (start == null)
      throw new ListException("finding next in empty list");
   else if (current.next == null)
      throw new ListException("no next element available");
   else
      current = current.next;
}
```

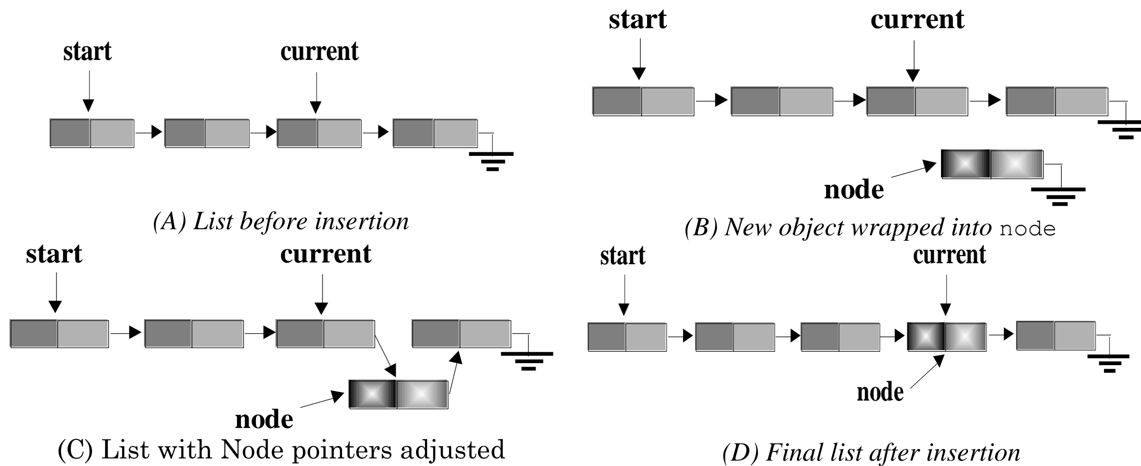
That leaves the operations `insert` and `remove`, which indeed are the difficult ones to implement. In fact, they are difficult to implement in part because they are not clearly defined:

- If we insert an element, where exactly should it be inserted in our list?
- After insertion, which element should be the current one?
- If we remove an element, which one should we actually remove?
- After removing, say, the current element, which one should become the new current element?

There are several possible answers to these questions (see exercises), and since our definition of a `List` does not mention these details, any possibility could be implemented. We will choose the following:

- A new element is always inserted right after the `current` one, and it should become the new `current` element after the insertion is complete.
- The element to be removed is always the `current` one, and its successor will become the new `current` element. If the element to be removed is the last one, we make the first element, if any, the `current` element. If the element to be removed is the only one in the list, our list should become a proper empty list.

Now that it is clear how insertion and deletion should work in theory, let's look at a picture of the insertion procedure in a generic situation:



(A) List before insertion *(B) New object wrapped into* `node`

(C) List with Node pointers adjusted *(D) Final list after insertion*

Figure 8.1.7: Inserting a new `Node` *into a generic* `List`

Thus, our insertion method consists of three stages, starting with the original list (A):

- (B): The input `Object` is wrapped into a new `Node` referenced by the variable `node`
- (C): The references of the `next` field of the `current` node as well as of the new `node` are modified to create the impression of the new `node` being seamlessly integrated into the list
- (D): The reference that `current` contains is reset to point to the newly arrived `node` in the list.

This translates immediately into the following programming steps:

```
public void insert(Object obj)
{  Node node = new Node (obj);          // step B
   node.next = current.next;            // step C
   current.next = node;
   current = node;                      // step D
}
```

However, this code results from the above picture, which depicted a "generic" situation. We should also check whether our code works in "extreme" cases. First, of course, we need to identify these extreme cases.

In the generic situation, the list contains some elements and current points to an "inside" element. The extreme cases, therefore, are:

a) `current` points to the first element
b) `current` points to the last element
c) the list is empty, i.e. `current` and `start` point to `null`

We need to verify - and if necessary adjust - our code in these situations before we feel confident that it will work as advertised.

Case (a): `start` and `current` both point to the first node:

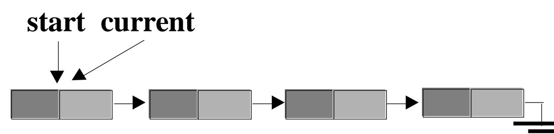


Figure 8.1.8: List insertion where start and current are the same

Our code creates the new `node` correctly, then sets the `next` reference of `node` to `current.next`, i.e. to the second node. The reference `current.next`, currently pointing from the first to the second node, is reset to point to the new `node`. Finally, the `current` reference is changed to point to `node`, so that the new `node` is indeed correctly inserted at the second position.

Case (b): `current` points to the last node in the list.

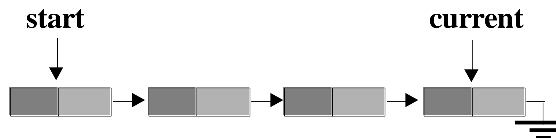


Figure 8.1.9: List insertion where start is first and current is last node

Again, the new `node` is created, then its `next` reference is adjusted to point to `null` (which is where `current.next` points to). Then `current.next` is adjusted to point to `node` and finally `current` is set to point to `node`. That will correctly insert the new `node` at the end of the list, and the list is properly terminated with a node whose next reference points to `null` as before.

Case (c): the list is empty, i.e. both `start` and `current` point to `null`.

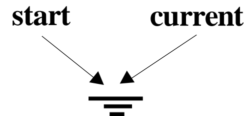


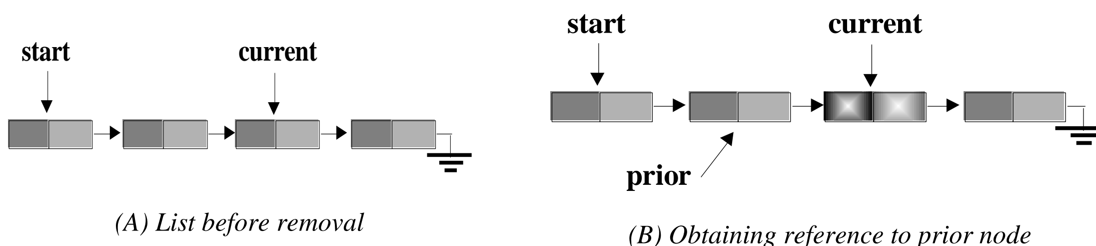
Figure 8.1.10: List insertion into an empty list

The new `node` is created and its `next` reference is set to point to `current.next`. However, since `current` points to `null` there is no `current.next` and the program will crash. Also, in this case the `start` reference will need to change, which does not happen in our current code. Therefore, we need to adjust our code to treat this particular situation separately:

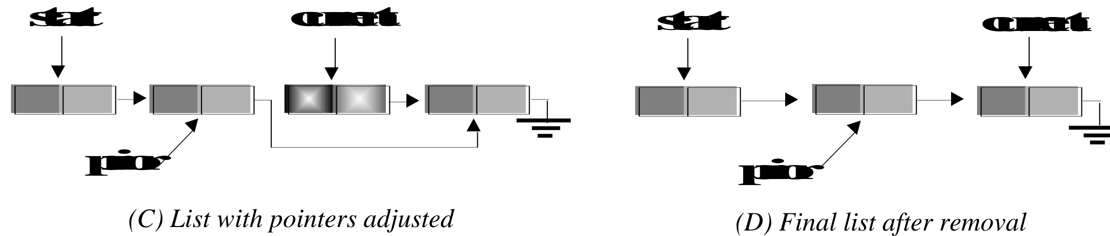
```
public void insert(Object obj)
{  Node node = new Node(obj);
   if (current != null)            // list is not empty
   {  node.next = current.next;
      current.next = node;
   }
   else                            // list is empty
      start = node;
   current = node;                // do this in both cases
}
```

This new `insert` method will perform correctly in every situation.

Finally, we have to take care of removing a node from the list. First, let's look at a picture of the generic situation. Rrecall that we want to remove the `current` element from the list and reset `current` to the node succeeding the removed one or to the first node if necessary:



(A) List before removal *(B) Obtaining reference to prior node*



(C) List with pointers adjusted *(D) Final list after removal*

Figure 8.1.11: Removal of the current node from a generic list

The first thing we notice is that if we want to remove the list element pointed to by `current` from the list, we must reset the reference of the *previous* element of the `current` one. However, we have no reference to that element ! Therefore, we first need to create a utility method that returns a reference to node prior to `current`, if any:

```
private Node findPrevious(Node now) throws ListException
{  if (now == start)
      throw new ListException("no previous element");
   else
   {  Node cursor = start;
      while (cursor.next != now)
         cursor = cursor.next;
      return cursor;
   }
}
```

Note that this method is marked private, so it can only be used inside the `List` class. It will throw an exception with appropriate error message if necessary, otherwise it will place a "`cursor`" at the first element and reset that `cursor` to its successor node until the successor node equals the input node. When it reaches that point, the loop is done and the method returns a reference to the node prior to the input node `now`. Armed with this method, our remove method is simple:

```
public void remove() throws ListException
{  if (start == null)
      throw new ListException("removing from empty list");
   else
   {  Node prior = findPrevious(current);   // step B
      prior.next = current.next;            // step C
      current = current.next;               // step D
   }
}
```

Note that the node that used to be referenced by `current` before the removal has only been removed from the list, not actually removed from memory. However, there is no reference to that node anymore, so it is no longer accessible. In practical terms, the node has indeed disappeared even though it is physically still in memory. But since no reference to that node exists the Java garbage collection thread can clean it up automatically when convenient. We could manually start the garbage collection process by adding a call to `System.gc();` but that would not be economical. One of the nice features of Java is this automatic garbage cleanup process and we should let it handle the necessary details by itself.

The above code, of course, may not cover every possible situation. After all, it is again modeled after the "generic" situation where the list contains some nodes and `current` points to a node in the middle of the list. Just as before we need to examine our code to check that it applies to any extreme situation and modify it if necessary. The extreme cases for this situation are:

a) `current` points to the first node, i.e. we want to remove the first node
b) `current` points to the last node, i.e. we want to remove the last node
c) there is only one node, and both current and start point to it

Case (a): `start` and `current` both point to the first node:

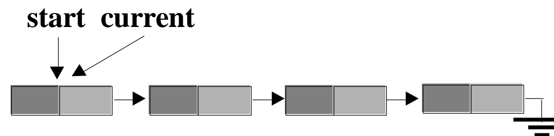


Figure 8.1.12: Node removal where start is equal to current

We notice immediately that our code will fail because the first thing it attempts to do is to find the node prior to `current`. Since `current` does not have a predecessor, a `ListException` would be thrown. We could modify our code to handle this particular situation as follows:

```
public void remove() throws ListException
{  if (start == null)
      throw new ListException("removing from empty list");
   else if (current == start)
      start = current = current.next;
   else
   {  Node prior = findPrevious(current);   // step B
      prior.next = current.next;            // step C
      current = current.next;               // step D
   }
}
```

This new code will cover the generic situation as well as case (a) in figure 8.1.12

Case (b): `current` points to the last node in the list.

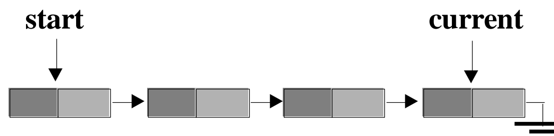


Figure 8.1.13: Node removal where start is first and current is last node

We will start with the above new code. Since `current` does not point to the first node, the last `else` part of the `if` statement will execute. The node prior to `current` will be found, then its `next` reference is adjusted to point to `null` since that's where `current.next` points to. Then `current` will be changed to also point to null. That last step, of course, is incorrect. In fact, during our clarification of the removal procedure we specified that if the last node is deleted from a list, `current` should point to `start` after the process. Hence, we modify our code once again by appending another `if` statement:

```
public void remove() throws ListException
{  if (start == null)
      throw new ListException("removing from empty list");
   else if (current == start)
      start = current = current.next;
   else
   {  Node prior = findPrevious(current);   // step B
      prior.next = current.next;            // step C
      current = current.next;               // step D
   }
   if (current == null)
      current = start;
}
```

Now the code will work in the generic situation, as well as in case (a), figure 8.1.12, and (b), figure 8.1.13.

Case (c): `current` and `start` both point to the only node in the list.

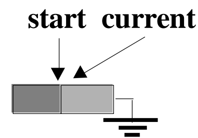


Figure 8.1.14: List removal where start and current point to only node

Since `start` and `current` both point to the same (and only) node, the first part of the `if` statement applies. But since `current.next` points to `null`, both `start` and `current` will be reset to also point to `null`. That results in a proper empty list, hence our code will cover this case just fine.

Our removal method, therefore, will now work in every possible situation. Since that was the last method we needed to implement, our `List` class is finished. For easy reference, here is the entire class once again[23]:

```
public class List
{  private Node current;
   private Node start;
   public List()
   {  start = current = null; }
   public boolean isEmpty()
   {  return (start == null); }
   public boolean isLast()
   {  if (current != null)
         return (current.next == null);
      else
         return false;
   }
   public void first() throws ListException
   {  if (start == null)
         throw new ListException("no first in empty list");
      else
         current = start;
   }
   public void next() throws ListException
   {  if (start == null)
         throw new ListException("finding next in empty list");
      else if (current.next == null)
         throw new ListException("no next element available");
      else
         current = current.next;
   }
   public Object retrieve() throws ListException
   {  if (start == null)
         throw new ListException("retrieving data from empty list");
      else
         return current.data;
   }
   public void replace(Object obj) throws ListException
   {  if (start == null)
         throw new ListException("replacing data in empty list");
      else
         current.data = obj;
   }
```

[23] The class will only compile if `ListException` and `Node` exist in the same directory.

```
    public void remove() throws ListException
    {  if (start == null)
          throw new ListException("removing from empty list");
       else if (current == start)
          start = current = current.next;
       else
       {  Node prior = findPrevious(current);
          prior.next = current.next;
          current = current.next;
       }
       if (current == null)
          current = start;
    }
    public void insert(Object obj)
    {  Node node = new Node(obj);
       if (current != null)
       {  node.next = current.next;
          current.next = node;
       }
       else
          start = node;
       current = node;
    }
    private Node findPrevious(Node now) throws ListException
    {  if (now == start)
          throw new ListException("no previous element");
       else
       {  Node cursor = start;
          while (cursor.next != now)
             cursor = cursor.next;
          return cursor;
       }
    }
 }
```

Before we discuss some benefits of and variations on this particular implementation, let's take a look at how this `List` class can be used.

Example 8.1.8:

Use a list to store two `double` numbers, then retrieve them again and print out the sum of these numbers.

This seems easy enough: create an empty list, insert two `double` numbers, retrieve them again and add them. So, we might try the following:

```
public class ListTest
{  public static void main(String args[])
   {  List list = new List();
      list.insert(10.0);
      list.insert(20.0);
      /* code to retrieve and add the numbers */
   }
}
```

But that code will not compile with error messages as in figure 8.1.15:

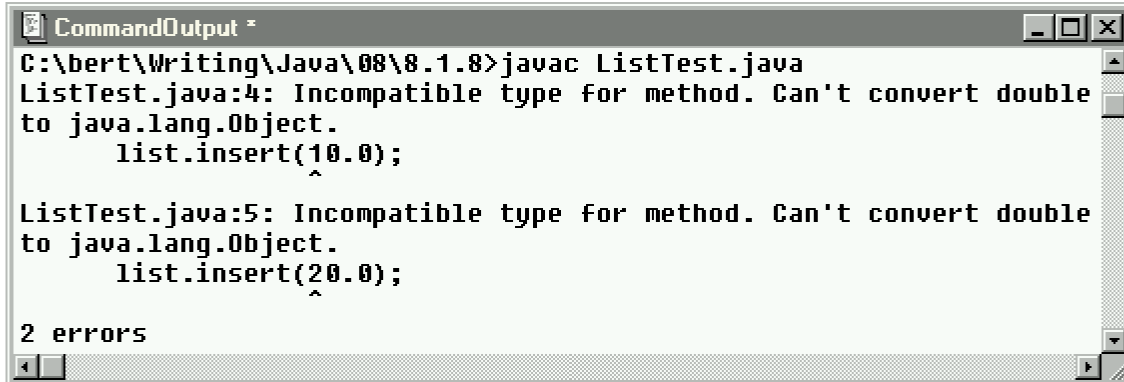


Figure 8.1.15: Error message when compiling original `ListTest` *class*

Indeed, our list can store the Java type `Object` or any type that derives from `Object`. Since every class extends `Object` *except* for the basic data types, these types can *not* be stored directly in our list. But the basic data types have corresponding wrapper classes that can wrap the value of a basic type into a proper Java `Object`. Hence, we modify our code as follows:

```
public class ListTest
{  public static void main(String args[])
   {  List list = new List();
      list.insert(new Double(10.0));
      list.insert(new Double(20.0));
      try
      {  list.first();
         double x = list.retrieve();
         list.next();
         double y = list.retrieve();
         System.out.println(x + y);
      }
      catch(ListException le)
      {  System.err.println(le);
      }
   }
}
```

However, our class again does not compile, this time with error messages similar figure 8.1.16:

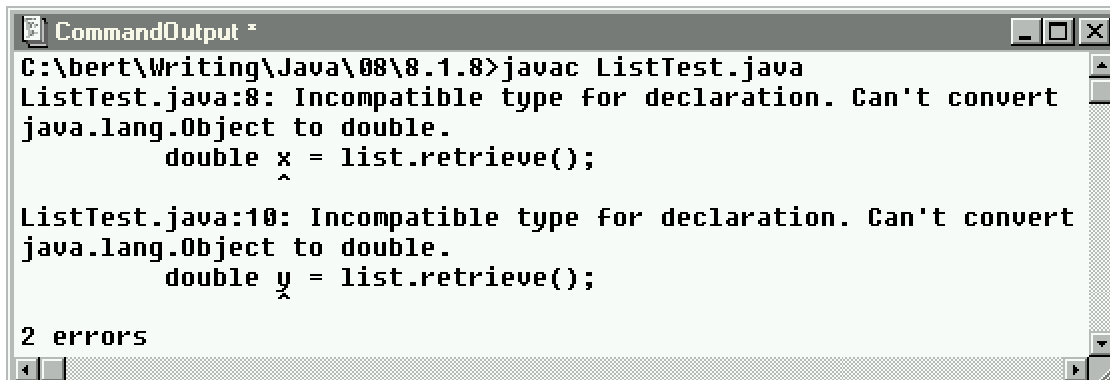


Figure 8.1.16: Error message when compiling second version of `ListTest`

Since we insert the type `Double` into the list we can not expect to retrieve a `double` type. But if we change our code once again by changing the types of `x` and `y` to `Double` instead of `double`:

```
public class ListTest
{  public static void main(String args[])
   {  List list = new List();
      list.insert(new Double(10.0));
      list.insert(new Double(20.0));
      try
      {  list.first();
         Double x = list.retrieve();
         list.next();
         Double y = list.retrieve();
         System.out.println(x + y);
      }
      catch(ListException le)
      {  System.err.println(le);
      }
   }
}
```

we get even more error messages than before:

```
CommandOutput
C:\bert\Writing\Java\08\8.1.8>javac ListTest.java
ListTest.java:8: Incompatible type for declaration. Explicit cast needed
to convert java.lang.Object to java.lang.Double.
         Double x = list.retrieve();
                ^
ListTest.java:10: Incompatible type for declaration. Explicit cast needed
to convert java.lang.Object to java.lang.Double.
         Double y = list.retrieve();
                ^
ListTest.java:11: Incompatible type for +. Can't convert java.lang.Double
to int.
         System.out.println(x + y);
                              ^
3 errors
```

Figure 8.1.17: Error message when compiling third version of `ListTest`

We insert the type `Double` into the list, and since `Double` extends `Object` the insertion works fine. However, the retrieval method *always* delivers an `Object`. If we know that the `Object` is really a `Double`, then we could use type-casting to change the types, as in:

```
Double x = (Double)(list.retrieve());
list.next();
Double y = (Double)(list.retrieve());
System.out.println(x + y);
```

Now, of course, we can not add `x` and `y`, because they are of type `Double`, not `double`, so we still have a problem (which we could overcome with the `doubleValue()` method of the `Double` class).

At this point, we should step back and think about the problem before attempting quick fixes. What we did create was a generic `List` class that can store any type of `Object`. However, in our case we want to store a particular data type, namely one or more `double`. Hence, we should create a new class that extends `List` and is tailored to our particular needs instead of trying other, non-OOP solutions.

In our case we want our list to store and retrieve `double` values, so we first create a new type of list, which will be very short yet completely functional:

```
public class DoubleList extends List
{  public void insert(double x)
   {  super.insert(new Double(x)); }
   public double retrieveDouble() throws ListException24
   {  return ((Double)(super.retrieve())).doubleValue(); }
}
```

With this new `List` type it is – finally – as easy as we thought to solve the original problem:

```
public class ListTest
{  public static void main(String args[])
   {  DoubleList list = new DoubleList ();
      list.insert(10.0);
      list.insert(20.0);
      try
      {  list.first();
         double x = list.retrieveDouble();
         list.next();
         double y = list.retrieveDouble();
         System.out.println(x + y);
      }
      catch(ListException le)
      {  System.err.println(le);
      }
   }
}
```

■

This solution is actually quite common when dealing with a `List`. We could use either type-casting when retrieving objects of a particular type or we could extend our list to adjust the `insert` and `retrieve` mechanism so that they use exactly the types we are interested in at that time. Note that in our `DoubleList` the `insert` method is overloaded, but we can not overload the `retrieve` message because of different return types.

Example 8.1.9:

Use the `DoubleList` created in example 8.1.8 to find the sum of the first 200 integers (which we know from example 29 to be equal to `20100`).

Assuming that we created the `DoubleList` class in example 8.1.8, the code should be easy: we insert the numbers as `double` in a loop, then retrieve them in a second loop to compute their sum. The first loop can be a `for` loop, but the retrieval part is more complicated. We first make sure that we start at the beginning of the list by calling on the `first` method. Then we use a `while` loop to cycle through the elements in the list until we have reached the last one. Each time we use `retrieveDouble` to get an element from the list and next to advance the current element by one. These calls must be embedded in a try-catch block to catch any list exceptions. Here is our first attempt, which requires the classes `Node`, `ListException`, `List`, and `DoubleList` to be present:

```
public class SumOfList
{  public static void main(String args[])
   {  DoubleList list = new DoubleList();
```

[24] We can not overload the `retrieve` method because the new method is supposed to return a `double`. Overloaded methods, however, must have the same return type. For the `insert` method overloading works fine.

```
            for (int i = 0; i < 100; i++)
               list.insert((double)(i+1));
            try
            {  double sum = 0.0;
               list.first();
               while (!list.isLast())
               {  sum += list.retrieveDouble();
                  list.next();
               }
               System.out.println("Sum of 1 to 200 is: " + sum);
            }
            catch(ListException le)
            {  System.err.println("List exception: " + le); }
         }
      }
```

The code will compile fine and produce an answer as shown in figure 8.1.18, but the answer is incorrect.

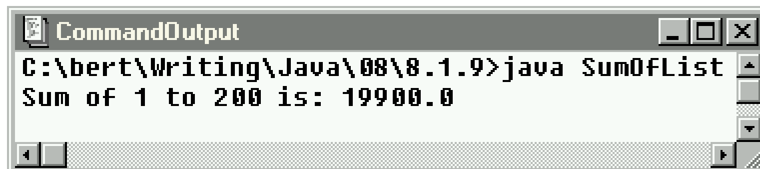


Figure 8.1.18: Results of incorrect `SumOfList`

A quick comparison against the correct answer of `20100` shows that the last number 200 has not been added. In fact, our `while` loop stops when the `current` element of the list is the last one, but it does not process the last element. Therefore, we must do that separately after the `while` loop. Adding a line to our `SumOfList` class will create the correct answer:

```
public class SumOfList
{  public static void main(String args[])
   {  DoubleList list = new DoubleList();
      for (int i = 0; i < 100; i++)
         list.insert((double)(i+1));
      try
      {  double sum = 0.0;
         list.first();
         while (!list.isLast())
         {  sum += list.retrieveDouble();
            list.next();
         }
         sum += list.retrieveDouble();
         System.out.println("Sum of 1 to 200 is: " + sum);
      }
      catch(ListException le)
      {  System.err.println("List exception: " + le); }
   }
}
```

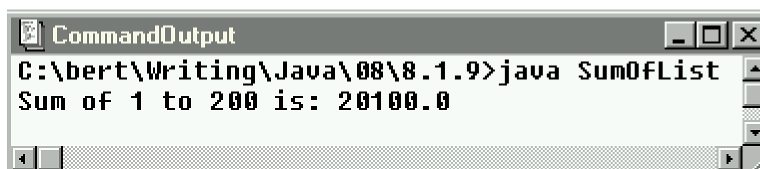


Figure 8.1.19: Results of correct SumOfList

■

Example 8.1.9 shows how to process all elements of a `List` in a while loop, but it was not very enlightening otherwise. The next example will illustrate how to use a `List` in a complete program that will remove the size restrictions that used to be imposed by an array. It will also show how to take advantage of polymorphism when working with lists.

Example 8.1.10:

XXX – Removed the MyDraw Applet in chapter 4 XXX In chapter four, example 4.6.7, we created a simple drawing applet called `MyDraw`. It could draw rectangles and circles as well as resize and move the last shape drawn. The applet used a hierarchy of shapes, stored the shapes in an array, and used polymorphism to select the methods appropriate for drawing a particular shape. At that time we simply restricted our applet so that it could draw up to `MAX_FIGS = 100` shapes. That is certainly sufficient for a simple program but won't do for a professional-strength application. Modify the `MyDraw` program so that it uses a `List` instead of an array for a – basically – unlimited number of shapes.[25]

First, let's recall the essential ingredients of that program. It used a hierarchy of `Shapes` as follows:

```
public abstract class Shape
{ /* various abstract method */ }

public class Rectangle extends Shape
{ /* implements the abstract methods specific to a rectangle */ }

public class Circle extends Shape
{ /* implements the abstract methods specific to a circle */ }
```

The highlights of the applet code were as follows:

```
public class MyDraw extends Applet implements ActionListener
{  private final int MAX_FIGS = 100;
   private Shape figures[] = new Shape[MAX_FIGS];
   private int   currentFig = -1;
   /* various other fields, see original code */
   public void init()
   { /* defines the layout for the applet, see original code  */ }
   public void actionPerformed(ActionEvent e)
   {  Object target = e.getSource();
      if (target == drawRectangle)
         handleNewFigure("rectangle");
      else if (target == drawCircle)
         handleNewFigure("circle");
      else if (currentFig >= 0)
      {  if (target == left)
            figures[currentFig].moveLeft(units);
         else if (target == right)
            figures[currentFig].moveRight(units);
         else if (target == up)
         /* more code calling on other methods of a Shape */
      }
      repaint();
   }
```

[25] The size of a `List` is not unlimited. It is rather limited by the available memory provided by the operating system. More sophisticated versions of a `List` class could also make use of available disk space to swap currently unused elements of the list to disk, giving that `List` an even greater but still limited possible size.

```
      public void paint(Graphics g)
      {  if (currentFig >= 0)
            for (int i = 0; i < currentFig+1; i++)
               figures[i].draw(g);
      }
      private void handleNewFigure(String figure)
      {  if (currentFig < MAX_FIGS-1)
         {  currentFig++;
            int x = getSize().width / 2;
            int y = getSize().height / 2;
            if (figure.equals("rectangle"))
               figures[currentFig] = new Rectangle(x,y,x+20,y+20,Color.red);
            else if (figure.equals("circle"))
               figures[currentFig] = new Circle(x, y, 10, Color.blue);
         }
      }
   }
```

If we want to convert this applet so that it uses a `List` instead of an array we should first create an appropriate `List` specific to dealing with shapes. Therefore, we will create `ShapeList` extending `List` with a `retrieve` routine specific to our shapes. The new `retrieve` method will return `Shape` but the list should contain `Rectangle` or `Circle` objects. After all, `Shape` is abstract and can not be used to instantiate any objects.

We could use the `instanceof` operator to check for the specific type of object in the list and type-cast the object appropriately before returning it:

```
public class ShapeList extends List
{  public Shape retrieveShape() throws ListException
   {  Object obj = super.retrieve();
      if (obj instanceof Rectangle)
         return (Rectangle)obj;
      else if (obj instanceof Circle)
         return (Circle)obj;
      else
         throw new ListException("unknown object in list");
   }
}
```

This only works because `Rectangle` and `Circle` are subclasses of `Shape`, so that if the method type-casts and returns a `Rectangle` it does return a `Shape` as well because every `Rectangle` is also a `Shape`. However, this is overkill: we can leave it to polymorphism to figure out the appropriate type of an object when necessary. In other words, our `retrieveShape` method can be simplified to:

```
public class ShapeList extends List
{  public Shape retrieveShape() throws ListException
   {  return  (Shape)super.retrieve(); }
}
```

The standard `insert` method can insert objects of type `Shape` just fine so we do not need to override it. When it inserts a `Rectangle` or `Circle` or any other object that extends `Shape`, the inserted object itself *is* of the correct type. As long as we type-cast it into a `Shape` when retrieving it, polymorphism can pick up on the true nature of the object and select the appropriate methods.

Armed with this new `List` we can easily modify the `MyDraw` applet to use a `ShapeList` instead of an array. In fact, our code will become somewhat easier since we can leave the actual details of keeping track of the current object etc. completely to the `ShapeList` class. Here is the new code, with the modified code displayed in bold and italics:

```
public class MyDraw extends Applet implements ActionListener
{  /* we don't need MAX_FIGS and currentFig */
   private ShapeList figures = new ShapeList();
   /* various other fields as before, see original code */
   public void init()
   { /* defines the layout for the applet, see original code  */ }
   public void actionPerformed(ActionEvent e)
   {  Object target = e.getSource();
      if (target == drawRectangle)
         figures.insert(new Rectangle(getSize().width/2-10,
                                      getSize().height/2-10,
                                      getSize().width/2+10,
                                      getSize().height/2+10, Color.red));
      else if (target == drawCircle)
         figures.insert(new Circle(getSize().width/2,
                                   getSize().height/2, 10, Color.blue));
      else
      {  try
         {  if (target == left)
               figures.retrieveShape().moveLeft(units);
            else if (target == right)
               figures.retrieveShape().moveRight(units);
            else if (target == up)
               /* more code as above calling on other methods of Shape */
         }
         catch(ListException le)
         {  System.err.println(le); }
      }
      repaint();
   }
   public void paint(Graphics g)
   {  if (!figures.isEmpty())
      {  try
         {  figures.first();
            while (!figures.isLast())
            {  figures.retrieveShape().draw(g);
               figures.next();
            }
            figures.retrieveShape().draw(g);
         }
         catch(ListException le)
         {  System.err.println(le + " nothing to draw"); }
      }
   }
}
```

The `paint` method uses a while loop similar to the one in example 8.1.9, processing the last item in the list separately. The `handleNewFigure` method is no longer necessary.

To summarize, in our original `MyDraw` applet we had to keep track of the shapes "manually" and we were restricted to a fixed number of shapes. Using our new data structure we can leave these details to the `ShapeList` class and we do not need to impose an artificial limit on the number of shapes. Polymorphic selection of the drawing methods appropriate to the particular shapes works just as before. Thus, this applet is "better" than before, and in the process the code seems to have got easier, too. Of course, we have simply "moved" some parts of the old code into the new `List` class, which is actually much more complex than an array. But the `List` class is entirely reusable, so if we spend some time *once*, we can use that to our benefit every time we make use of `List`.

■

Our list implementation using nodes to store objects is only one possible implementation of the abstract data type `List`. Another commonly used variation is:

Definition 8.1.11: Double-Linked List

A double-linked list is a list implementation that uses "double" Nodes instead of "single" Nodes. A double Node has two reference variables in addition to storing data, one to point to a successor node and a second one to point to a predecessor. A double-linked list offers performance benefits over a standard linked list, paid for by additional memory requirements per node and more complex implementations of the list methods. The details are sketched in figures 8.1.20 and 8.1.21.

"Double-linked" list: this implementation uses a `Node` with *two* reference variables, one to point to the successor node and the second to point to the predecessor node:

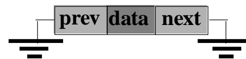


Figure 8.1.20: Image of a double-linked Node

With this node one can implement a list that looks similarly to figure 8.1.21:

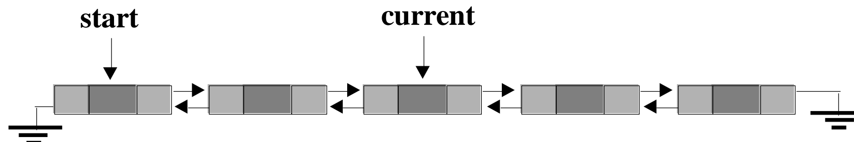


Figure 8.1.21: Image of a double-linked List

To implement this type of list, the insert and remove methods will be most complicated, as before, while the remaining methods will remain almost the same. The details are left as an exercise, but to gain experience with double-linked nodes here is a simple example:

Example 8.1.12:

Write some code that will change the state of a double-linked list from the one depicted in figure 8.1.22 on the left to the situation on the right. You do not need to create an actual `insert` (or other) method, the code segment to adjust the various pointers will be sufficient. You may, however, need to create double-linked `Node` class before you can use such nodes.

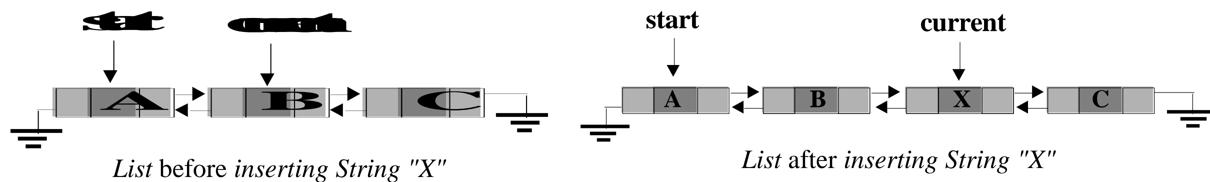


Figure 8.1.22: Inserting into a double-linked List

First, we need to create the `DoubleNode` class that contains a field for an object and two fields for reference variables to the predecessor and successor node. As before, we will make those fields protected for simplicity:

```
public class DoubleNode
```

```
{  protected Object data;
   protected DoubleNode next;
   protected DoubleNode prior;
   public DoubleNode()
   {  data = null;
      next = prior = null;
   }
   public DoubleNode(Object _data)
   {  data = _data;
      next = prior = null;
   }
}
```

Next, our code needs to create a new node containing the string "X" and adjust all necessary pointers so that the new data will fit into the list as depicted in figure 8.1.22. That is simple as long as we can give names to the reference variables we want to adjust:

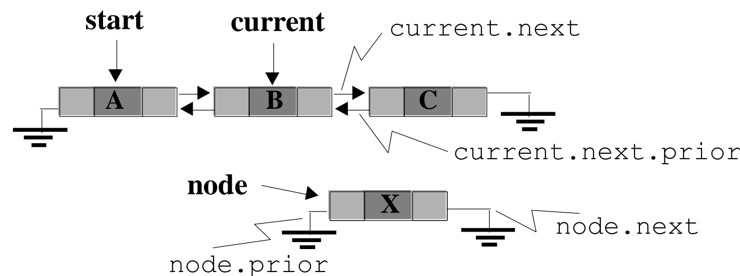


Figure 8.1.23: Names for nodes in double-linked List

Once everything has a name, the code is - hopefully - easy:

1. Create new node with appropriate data section:

```
Node node = new DoubleNode("X");
```

2. Adjust the various pointers to insert `node` right before `current`:

```
node.prior = current;
node.next = current.next;
current.next.prior = node;
current.next = node;
```

3. Resetting current to point to the new node to finish the exercise:

```
current = node;
```

Without creating the actual `insert` method for a double-linked list it would be difficult test this code. In fact, in the exercises you will be asked to create the entire double-linked implementation for a `List`, so you can verify this code at that time.

■

Another frequent variation of a `List` implementation is based on arrays, yet maintains the dynamic nature of a list:

Definition 8.1.13: Array-Based List

An array-based list is a list implementation that uses an array instead of nodes to keep track of the data. The class is constructed so that the array-based list still can grow and shrink dynamically in size. The details are sketched below.

"Array-based" List: this implementation does not use nodes but an array to store objects.

size	4
current	2

0	data
1	data
2	data - current
3	data
4	data - empty
5	data - empty

Figure 8.1.24: array-based List

The data is stored in an array of a fixed length (six in the picture on the left). The field `size` indicates how many data objects are actually stored at this time (four in the picture, leaving two empty slots) and the field `current` determines the object that is considered to be the current one at the given time.

There are two basic versions to consider: one version uses an array of a fixed size, the other uses an array that can grow and shrink dynamically. The "fixed size" version clearly is not interesting to us, so we will briefly explain the "dynamic" version. In this implementation the `List` could contain four fields: three integers (or longs) labeled `current`, `growBy`, and `size`, and one array of objects of some initial length. It will contain all standard public method of the `List` class – differently implemented – as well as two private methods `grow` and `shrink`:

```
public class List
{  private long current = -1;   // points to current data object
   private long growBy = 5;     // how much array should grow or shrink
   private long size = 0;       // how many objects are currently stored
   private Object data[] = new Object[10];
   // standard List methods
   private void grow()   { }
   private void shrink() { }
}
```

Initially this class can contain at most 10 objects. As soon as the 10th object is added (via the `insert` method) the method `grow` is called by `insert`. That method will create a new array of size `data.length + growBy`, copy all old `data` objects into that array and reset `data` to point to the new and longer array. In this way, the array `data` can grow as needed and the `List` can accommodate as many objects as available memory permits.

Similarly, if an object is removed from the list via the `remove` method, it will call `shrink` whenever the actual number of objects indicated by the value of `size` falls below some number (say one third of the current size of the array). The `shrink` method will create a new array of, say, half the size of the current array, copy all objects from `data` into that array, and reset `data` to point to this new and shorter array. In this way, the array will shrink as necessary, and the automatic garbage collection thread can reclaim any unused memory.

If the `List` class is implemented this way, it would mean that occasionally a call to `insert` or `remove` would take much more time than usually because the `grow` or `shrink` methods need to be called to adjust the size of the array. If many elements need to be added in succession, it would be better to make the array grow by more than 5 slots so that the `grow` method is not called too frequently. To that end, one could add a utility method such as `setIncrement` that could change the value of `growBy`, and perhaps a second constructor that would specify the initial size of the `data` array.

We will not implement this class but leave the details as an exercise. However, the Java language does provide a special class called `Vector` that works somewhat similar and we will briefly discuss that class in definition 8.3.2.

The biggest disadvantage of this array-based implementation is that the data is stored in an array and an array requires one contiguous block of memory. That limits the maximum possible size of the array to the largest contiguous block of memory (at the most), not to all available memory. The node-based implementation, on the other hand, can store its data in nodes spread around the available memory, but it pays for this flexibility by adding an extra variable requiring memory to each data element stored.

The advantages and disadvantages of these different implementations are summarized in the following table:

	Advantage	**Disadvantage**
standard List	relatively easy to implement, non-contiguous memory needs; best used if list size is small or if there are more insertions than deletions	some extra memory needed per node, slowness of `findPrevious` and `remove` methods; not useful if "random" items in list need to be accessed frequently
double-linked List	speed of `findPrevious` and `remove` methods; best used if speed is important	difficult to implement, extra memory needs per node; not useful if memory is "tight"
array-based List	easy to implement, no extra memory needs; best used if there are frequent traversals of the list relative to the number of insertion and deletions; quick access to all items	contiguous memory requirement, slowness of growing and shrinking; not useful if there are frequent insertion and deletions

Table 8.1.25: Comparing implementation versions for List

Any class using one of these types of lists can, in principle, also use any other type without changing any of the functional code of the class. Thus, a programmer can pick whatever representation of a `List` seems to be the best solution for the particular situation.

At this point we should be convinced of the usefulness of a `List`. Other ADT's with more structure can also be created that will be specialized but offer advantages in certain situations.

Queues

Queues are a special type of sequential structure where insertion of new objects takes place at one end and retrieval takes place on the other end. Queues are used frequently behind the scenes in operating systems, and in particular in networking systems. For example, many offices use printers (especially high-speed, high-quality ones) that are networked so that more than one person can send print requests to that printer. That poses a slight problem: one person might send a multi-page print job to the printer and while the printer is about to print page 10, another person sends another print job to the same printer. If both people had direct access to this printer, the first print job would now be interrupted and the second job would start printing. Therefore, access to a printer is usually mediated by what is called a "print queue". Instead of sending a print job directly to a printer across the network, the print job is instead sent to a special holding area (usually a program running on a

dedicated computer), and only that special program can access the printer directly. As print requests are coming into that holding area, the data is stored at the end of a queue. The dedicated program, on the other hand is the only one with access to the printer. It removes print jobs from the top of its queue, sends them to the printer, waits until that job is finished, and sends the next job, if available. The printer will not get confused as to which job to finish, because only *one* computer has access to the printer. To the users it looks as if the printer prints their requests in an orderly "first-come, first-serve" fashion.

Definition 8.1.14: ADT Queue

A queue is a sequential data structure where data is inserted at one end and removed from the other end. Direct access to data residing in the middle of the queue is prohibited. The operations that a queue should provide are as follows:

- *to create an empty queue*
- `enqueue`*: to enter data into a queue on one end*
- `dequeue`*: to remove data from the queue on the other end and return the data*
- `peek`*: to retrieve the data on the removal end without removing it from the queue*
- `isEmpty`*: to check whether a queue is empty*

A queue implements a "first-in, first-out", or FIFO access structure, i.e. the data that was entered first into a queue is also retrieved first.

If we wanted to implement a `Queue` class, an easy solution would be to use our list as the superclass and implement `Queue` as extending `List`. Then we would modify the `insert` and `retrieve` methods to ensure that data is always inserted at the beginning of the list, say, and retrieved at the end. However, that would not be in accordance with definition 8.1.14. If a class extends `List`, all public methods of the `List` class are available. In particular, the methods `first` and `next` are available to position the cursor anywhere in the list and data could be retrieved from anywhere in the queue. Therefore, we will implement a `Queue` class entirely from scratch instead of extending `List`.

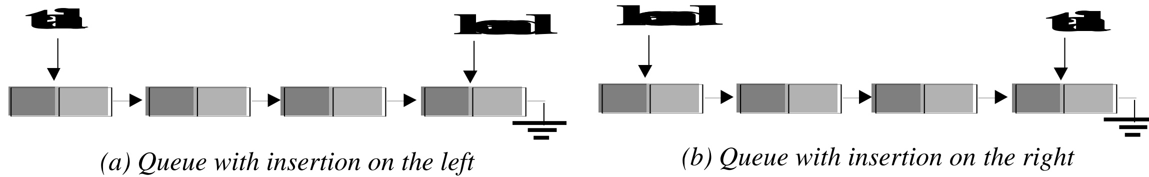
Two implementations are possible: one using `Nodes` and the second one using an array. But for a `Queue`, there are only two essential operations (`enqueue` and `dequeue`) and traversal or search is not allowed. Therefore, the node-based implementation (with single nodes) is the most advantageous one:

Example 8.1.15:

Create a node-based implementation of a `Queue` that implements the methods `enqueue`, `dequeue`, `peek`, and `isEmpty` as defined in definition 8.1.14.

You can use the `Node` class from definition 8.1.5 "as is", without any modifications.

Before we write any code, let's draw a picture of how our `Queue` would look like. We have two choices: we could insert elements at the left and remove them from the right, or we can insert elements on the right and remove them from the left. Correspondingly, let's look at two representations of a queue (where `tail` denotes the side of the queue for insertion, `head` the side from removal):



(a) Queue with insertion on the left *(b) Queue with insertion on the right*

Figure 8.1.26: Two choices of representing a node-based queue

To insert (`enqueue`) a new node as in figure 8.1.26, case (a), we need to adjust the `tail` variable to point to the new node, as well as the `next` reference of the new node to point to the old `tail` of the queue. To remove (`dequeue`) a node we need to reset the `next` reference of the node prior to `head` to point to `null`. That operation therefore requires access to the second-last node and in our picture we have no convenient access to that node. Several remedies are possible, but let's first consider the second case.

To insert (`enqueue`) a node as in figure 8.1.26, case (b), the `next` reference of the `tail` node needs to be adjusted to point to the new node. To remove (`dequeue`) a node in case (b), we need to reset `head` to point to the second node in the queue. But now we *do* have convenient access to all necessary reference variables, so that case (b) is preferred over case (a) and we will use that representation for our `Queue` implementation.

Of course, a `dequeue` operation can only succeed if the queue is not empty. To ensure that this condition is satisfied, we will employ a `QueueException` and throw it in case the `dequeue` method is used on an empty queue. Therefore, we will first create a `QueueException` class, similar to `ListException` above[26]:

```
public class QueueException extends Exception
{  public QueueException()
   {   super("Queue Exception"); }
   public QueueException(String msg)
   {  super(msg); }
}
```

Our `Queue` class so far looks as follows:

```
public class Queue
{  private Node head;
   private Node tail;
   public Queue()                                  { }
   public void enqueue(Object obj)                 { }
   public Object dequeue() throws QueueException   { }
   public Object peek() throws QueueException      { }
   public boolean isEmpty()                        { }
}
```

To focus on the `enqueue` (insert) operation, let's first consider a picture of the generic situation:

[26] `List`, `Queue`, and `Stack` (defined later) are frequently used together, and it would probably be advantageous to define `QueueException` and `StackException` as subclasses of `ListException`.

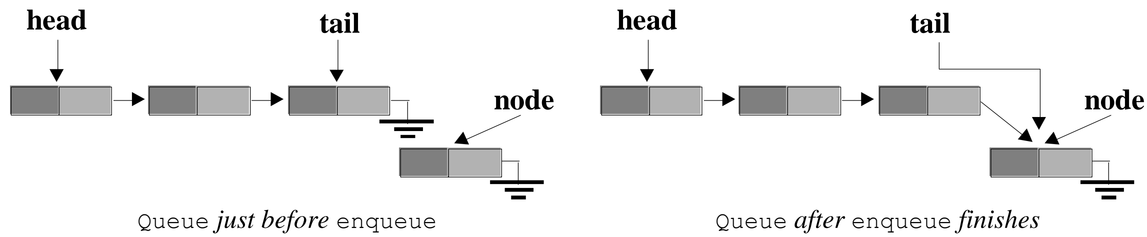


Figure 8.1.27: Inserting a new node into a generic queue

Based on that picture, the `enqueue` method looks as follows:

```
public void enqueue(Object obj)
{  Node node = new Node(obj);
   tail.next = node;
   tail = node;
}
```

However, that does not take into account the extreme situations of an empty queue (`head` and `tail` point to `null`) or of a queue where `head` and `tail` point to the same node:

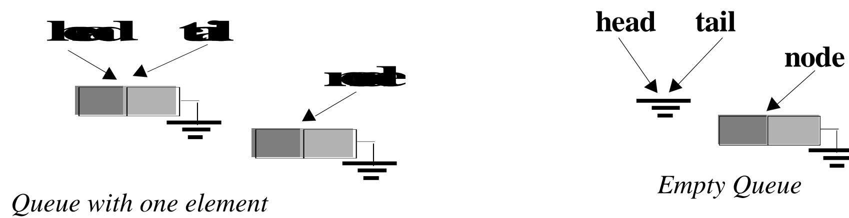


Figure 8.1.28: Inserting a new node into a queue in extreme cases

In the first case, our code will work just fine. In the second case we also need to adjust `head` so that it, too, will point to the new and only node that was inserted. Hence, our final version of the `enqueue` method is:

```
public void enqueue(Object obj)
{  Node node = new Node(obj);
   if (head == null)
      head = node;
   else
      tail.next = node;
   tail = node;
}
```

As for the `dequeue` method, we will again start with a generic picture:

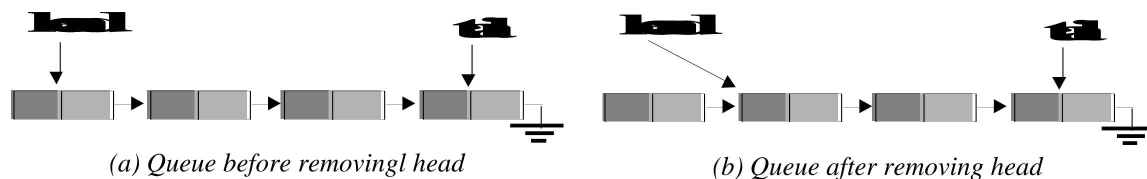


Figure 8.1.29: Removing a node from a generic queue

Based on that picture, our `dequeue` operation looks like this:

```
public Object dequeue() throws QueueException
{  if (head == null)
      throw new QueueException("removing from empty queue");
   else
      head = head.next;
}
```

While that code will indeed remove the head node from the queue, it does not *return* its data value as it should. In fact, as soon as we reset the head node, we have no possibility to access the value, so we need to *first* return the value, *then* reset the head node. But that will not work either. As soon as we execute a return statement the code after that statement will no longer execute, hence we can not reset the head reference after return.

The solution will be to introduce a temporarily store the data at the head of the queue, then reset the head reference, and finally return the appropriate data value. Therefore:

```
public Object dequeue() throws QueueException
{  if (head == null)
      throw new QueueException("removing from empty queue");
   else
   {  Object data = head.data;
      head = head.next;
      return data;
   }
}
```

But again, this does not take into account the extreme situation of having a queue with a single node. In that case tail needs to be set to null as well so we adjust our code as follows:

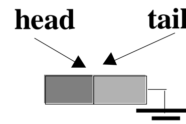


Figure 8.1.30: Queue with one element

```
public Object dequeue() throws QueueException
{  if (head == null)
      throw new QueueException("removing from empty queue");
   else
   {  Object data = head.data;
      head = head.next;
      if (head == null)
         tail = null;
      return data;
   }
}
```

The remaining methods are easy so here is the complete Queue class:

```
public class Queue
{  private Node head;
   private Node tail;
   public Queue()
   { head = tail = null; }
   public void enqueue(Object obj)
   {  Node node = new Node(obj);
      if (head == null)
         head = node;
      else
         tail.next = node;
      tail = node;
```

```
   }
   public Object dequeue() throws QueueException
   {  if (head == null)
         throw new QueueException("removing from empty queue");
      else
      {  Object data = head.data;
         head = head.next;
         if (head == null)
            tail = null;
         return data;
      }
   }
   public Object peek() throws QueueException
   {  if (head == null)
         throw new QueueException("peeking into empty queue");
      else
         return head.data;
   }
   public boolean isEmpty()
   {  return (head == null); }
}
```

■

We will provide some more elaborate examples in the next section after introducing stacks. For now, here is a simple example:

Example 8.1.16:

Create a (fake) `PrintJob` class that stores the name of the person submitting the job and the date the job was submitted. Make sure the class includes a `toString` method. Then insert a few print jobs into a queue, wait a few seconds, then retrieve the print jobs and show their values on standard output, noting the time the job was retrieved from the queue. Try to remove one more print job from the queue than is actually present to test the exception handling of our class.

The first part of this example is the creation of the `PrintJob` class. That class contains no surprises: it uses a field of type `String` for the name of the person submitting the job, and a second field of type `Date` for the date and time the job was created. Both values can be set in the constructor, and can be displayed with the `toString` method:

```
import java.util.Date;

public class PrintJob
{  private String name;
   private Date date;
   public PrintJob(String _name, Date _date)
   {  name = _name;
      date = _date;
   }
   public String toString()
   {  return (name + " (" + date + ")"); }
}
```

Recall that the `toString` method of the `Date` class is used automatically to convert a `date` to an appropriate `String` representation in the `toString` method.

The main program is also straightforward, with the exception, perhaps, of how to make our program "wait for a few seconds". But all programs in the JVM run in a thread and threads, as we know from

chapter 5, can be put to sleep. In fact, there is a static method of the `Thread` class called `sleep` that will put the currently executing thread to sleep for the indicated period of time.[27] Now we have all ingredients for our program:

```
import java.util.Date;

public class PrintQueueTest
{  public static void main(String args[])
   {  Queue printer = new Queue();
      printer.enqueue(new PrintJob("Bert Wachsmuth", new Date()));
      printer.enqueue(new PrintJob("Jane Doe", new Date()));
      try
      {  Thread.sleep(2000);
         while (!printer.isEmpty())
         {  System.out.println("Printing starts: " + (new Date()));
            System.out.println("\tJob: " + printer.dequeue());
         }
         System.out.println("Printing starts: " + (new Date()));
         System.out.println("\tJob: " + printer.dequeue());
      }
      catch(InterruptedException ie)
      {  System.err.println(ie); }
      catch(QueueException qe)
      {  System.err.println(qe); }
   }
}
```

The insertion of the print jobs is straightforward. To remove and display the information, we have embedded the call to `dequeue` in a `try-catch` block catching an `InterruptedException` as well as a `QueueException`. To print out a `PrintQueue` as a `String` we do *not* have to type-cast the `Object` returned by `dequeue` into `PrintJob`. After all, `PrintJob`, like every class, extends `Object`, and `Object` contains a `toString` method. Hence, polymorphism will select the `toString` method appropriate for a `PrintJob` automatically. We do not have to call on the `toString` method explicitly since that happens automatically in `System.out.println`. Figure 8.1.31 shows the output resulting from this program:

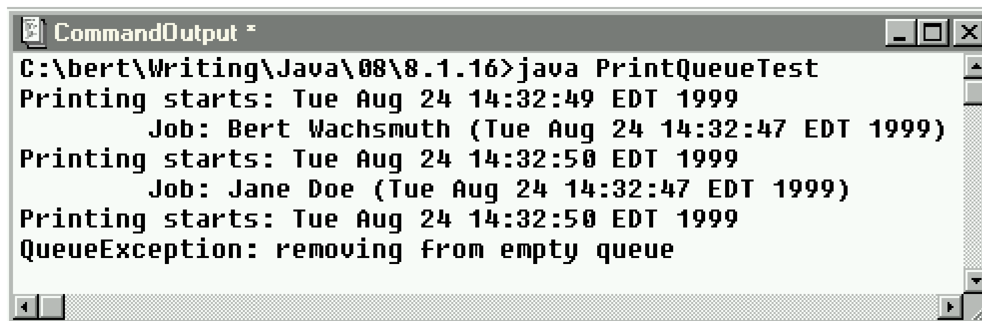


Figure 8.1.31: Output of `PrintQueueTest` *program*

■

There are several other useful variations of a queue: first of all, a `Queue` could be implemented using an array, but as we mentioned before there would be little or no performance gain in that. Another variation of a `Queue`, however, is quite useful: a priority queue. A priority queue is a queue that does not follow a FIFO retrieval mechanism but instead orders the data according to some priority such that the item to `dequeue` always has equal or higher priority than every other item in the list. In case of equal priority, the item that was inserted first will be `dequeued` first.

[27] See chapter 5, table 5.2.3.

Definition 8.1.17: ADT Priority Queue

> *A priority queue is a sequential structure containing items sorted according to some priority. Items are inserted via the* `enqueue` *method and removed via the* `dequeue` *method. Every item* `dequeued` *from the priority queue has a priority at least as high as every item still remaining in the priority queue. Among items of equal priority, they are* `dequeued` *in "first-in, first-out" order.*

To implement a priority queue we could extend `Queue`. We would only need to override the `enqueue` method so that it does not add object at the `tail` end of the queue but instead according to the priority of the new item compared to those already in the queue. We need to be able to compare items to be inserted against those already in the list, so we need to make sure that all items in the priority queue are comparable. The most flexible way to achieve that is via an interface. In fact, in chapter 7 we created the `Searchable` interface which will do perfectly fine in this case. Recall that interface from example 7.3.7:

```
public interface Searchable
{  public abstract boolean isEqualTo(Searchable obj);
   public abstract boolean isLessThan(Searchable obj);
   public abstract boolean isGreaterThan(Searchable obj);
}
```

So, our class would start out like this:

```
public class PriorityQueue extends Queue
{  public void enqueue(Searchable obj)
   { /* code to insert according to priority */ }
}
```

to `enqueue` objects that implement the `Searchable` interface. However, to implement the new `enqueue` method we need to be able to access at least the head reference to start searching the priority queue from the beginning for the correct position for the insertion. But head is marked as private in `Queue` so `PriorityQueue` does not have access to it. Therefore, we either modify the `Queue` class to mark `head` (and `tail`) as protected or we copy the code from the `Queue` class to the `PriorityQueue` class. For simplicity, we will designate both `head` and `tail` fields of the `Queue` class as protected:

```
public class Queue
{  protected Node head;
   protected Node tail;
   /* remaining code as before */
}
```

Now we have access to these fields in our `PriorityQueue` and the first order of business is to find the position where the new node should fit, according to its priority. The following code segment will accomplish this:

```
Node cursor = head;
while ((cursor != null) &&
       (!((Searchable)cursor.data).isLessThan(obj)))
   cursor = cursor.next;
```

It places a reference variable `cursor` at the `head` of the queue, then moves it from the `current` node to its successor as long as the cursor object is not less than the new object, or the end of the queue

has been reached. At the end of that loop, `cursor` will point either to `null` or to the first node with priority less than then priority of `obj`. Next, we have to consider four cases:

Case 1: the queue is empty (`head` points to `null`)
We can insert the node by pointing `head` and `tail` to the new (and only) node.
Case 2: `obj` has bigger priority than any other object in the queue (`cursor` points to `head`)
We need to insert the new node at the beginning of the queue (by changing `head`)
Case 3: `obj` has smaller priority than any other object in the queue (`cursor` points to `null`)
We need to insert the new node at the end of the queue (by changing `tail`)
Case 4: `obj` has priority less than the first and bigger than the last node
We need to insert the new object at the appropriate spot in the middle of the queue

Cases 1 to 3 are relatively easy. However, in the fourth case we need to insert the new node *before* the one pointed to by `cursor`. Thus, we need to modify the above loop to keep a reference variable that always points to the predecessor of `cursor`:

```
Node cursor = head, prior = head;
while ((cursor != null) &&
       (!((Searchable)cursor.data).isLessThan(obj)))
{  prior  = cursor;
   cursor = cursor.next;
}
```

In other words, the variable `prior` always points to the predecessor of `cursor` as soon as the loop starts.

With the help of these explanations and the above code segment, it should not be hard to finish the `enqueue` method for the `PriorityClass`. We will leave the details as an exercise.

■

We will introduce further examples in the next section when we know about another abstract data type called a `Stack`.

Stacks

While a Queue is a FIFO (first in, first out) structure, our next data type will represent a LIFO (last in, first out) structure, i.e. a structure where the node last inserted is the first to be retrieved. One such structure is called a Stack:

Definition 8.1.18: ADT Stack

A stack is a sequential data structure where data is inserted on one end and removed from the same end. Direct access to data residing in the middle of the stack is prohibited. The operations that a stack should provide are as follows:

- *to create an empty stack*
- `push`*: to enter data into a stack*
- `pop`*: to remove data from the stack*
- `peek`*: to retrieve the data without removing it from the stack*
- `isEmpty`*: to check whether a stack is empty*

> *A stack implements a "last-in, first-out", or LIFO access structure, i.e. the data that was entered last into the stack is retrieved first.*

In some sense, a stack is the opposite of a queue: in a queue, what goes in first also comes out first, whereas in a stack what is inserted *last* comes out first. At first glance it might seem that a queue is a "sensible" or "fair" structure with many uses but a stack is an "unfair" structure whose usefulness is not clear. Of course, the question of fairness is not relevant to data structures. As it will turn out a stack has indeed many uses and it can be quite helpful in complicated situations such as understanding how recursion works. And just as queues, stacks are used frequently in computer operating systems but they are usually much less obvious than queues.

One frequent use of a real-world stack-like structure can be found in many cafeterias. The mechanism that is used to dispense plates and trays allows you to take a plate from the top of a pile, then a spring pushes the remaining plates up a little so that the top plate is available for the next person to take. Indeed, that mechanism is LIFO: the last plate that was put on the pile is the first one to be taken out. Hence, such a mechanism is a good visual aide to imagine a stack.

Another use of a stack is implemented in most popular web browsers: when you visit pages on the web, a web browser usually allows you to go "back" to the previously visited pages. And in fact you always go back to the last page visited. Thus, a stack is used to keep track of the pages so that the page last inserted is removed first.

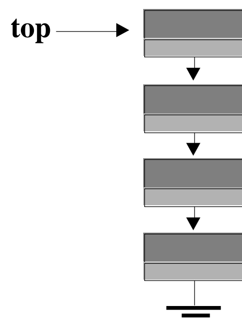
However, before determining other possible uses of stacks, we need to implement the appropriate class. Just as we did not create `Queue` as an extension of `List`, we will also design our `Stack` class from scratch.[28] Internally, a `Stack` will look somewhat similar to a `Queue`, but because of its LIFO mechanism the particular implementation will be different and its uses will be entirely different.

Example 8.1.19:

Create a node-based implementation of a `Stack` that implements the methods `push`, `pop`, `peek`, and `isEmpty` as defined in definition 8.1.18.

You can use the `Node` class from definition 8.1.5 "as is", without any modifications.

As usual, before attempting the actual implementation of the `Stack` class we will try to create an appropriate visual representation of a stack. Taking the above image of the plate-dispensing mechanism as a guide, we will visualize a stack as follows:



Instead of visualizing the nodes horizontally as we did for a `List` and `Queue`, we picture the nodes of a `Stack` vertically, with the top of the stack being appropriately at the top. Each node consists, as usual, of two fields, one for `data` and another for a reference to the `next` node. The `next` node, in this representation, is the one immediately underneath the current one. The bottom of the stack is designated by a `next` reference pointing to `null`. The only field we need inside a Stack class is a reference variable `top` pointing to the top of the stack.

[28] In section 8.3 we will introduce the Java build-in `Stack` class, which does extend a list-like structure called `Vector`. Technically speaking that should not be legal since that build-in stack will inherit operations that a stack should not possess.

Figure 8.1.32: Image of a Stack

Based on this picture and on the definition, we start our `Stack` class as follows:

```
public class Stack
{  private Node top;

   public Stack()
   public void push(Object obj)
   public Object pop() throws StackException
   public Object peek() throws StackException { }
   public boolean isEmpty()
}
```

Note that since the method `pop` and `peek` require the `Stack` to be not empty, they throw a `StackException` if that condition is not met. The `StackException` class will be as usual:

```
public class StackException extends Exception
{  public StackException()
   {  super("Stack Exception"); }
   public StackException(String msg)
   {  super(msg); }
}
```

Next, we will implement the three easy methods `peek`, `isEmpty`, and the `constructor`:

```
public Stack()
{  top = null; }
public Object peek() throws StackException
{  if (top == null)
      throw new StackException("peeking into empty stack");
   else
      return top.data;
}
public boolean isEmpty()
{  return (top == null); }
```

As far as the `push` method goes, it will instantiate a new node containing `obj`, set its `next` reference to the current `top` of the stack, then reset the `top` to point to the new node:

```
public void push(Object obj)
{  Node node = new Node(obj);
   node.next = top;
   top = node;
}
```

Note that this code will also work in the "extreme" case of an empty stack. The `pop` method is a little trickier: It needs to return the data stored in the `top` node of the stack, as well as reset the `top` field. However, if it first returns the data value, the method will finish and it can not reset the `top` of the stack. On the other hand, if we first reset the `top` we can not return the correct data object because nothing points to it any more. The solution, similar to the `dequeue` method of the `Queue`, is to temporarily store the correct data object elsewhere, adjust the `top` pointer, then return the object via its temporary reference:

```
public Object pop() throws StackException
{  if (top == null)
      throw new StackException("popping from an empty stack");
   else
```

```
      {  Object obj = top.data;
         top = top.next;
         return obj;
      }
   }
```

Note that again this code works for a "generic" stack with several nodes as well as for the "extreme" situation of a stack containing only one node.

For easy reference, here is the complete `Stack` class:

```
public class Stack
{  private Node top;
   public Stack()
   {  top = null; }
   public void push(Object obj)
   {  Node node = new Node(obj);
      node.next = top;
      top = node;
   }
   public Object pop() throws StackException
   {  if (top == null)
         throw new StackException("popping from an empty stack");
      else
      {  Object obj = top.data;
         top = top.next;
         return obj;
      }
   }
   public Object peek() throws StackException
   {  if (top == null)
         throw new StackException("peeking into empty stack");
      else
         return top.data;
   }
   public boolean isEmpty()
   {  return (top == null); }
}
```

■

Now let's test our new `Stack` class with an easy example:

Example 8.1.20:

Push the strings "`Java`", "`by`", "`Definition`" into a `Stack`, then `pop` them and print the results. What will you see?

The code is simple, so to make it more interesting we will use a loop terminated by an exception being thrown to pop all elements from a stack (even though that is not the best programming practice)[29]:

```
public class StackTest
{  public static void main(String args[])
   {  Stack s = new Stack();
      s.push("Java");
      s.push("by");
      s.push("Definition");
      try
```

[29] Compare with example 8.1.9.

```
        {  while (true)
               System.out.println(s.pop());
        }
        catch(StackException se)
        {  System.out.println("Done."); }
    }
}
```

When you execute the program, you will see the words "`Definition`", "`by`", "`Java`" (and, of course, "`Done.`") as in figure 8.1.33.

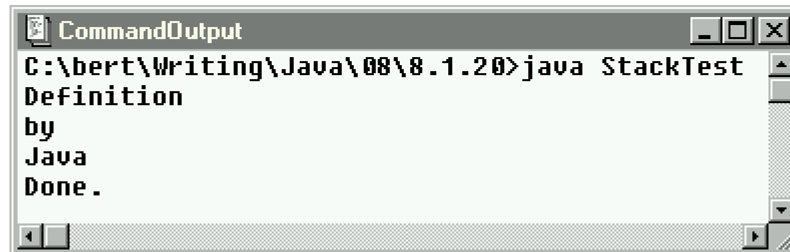


Figure 8.1.33: Output of `StackTest` *class*

That clearly shows the LIFO mechanism: the last string pushed onto the stack was "`Definition`", and therefore it is the first one that is popped from the stack.

■

The above example illustrates one simple use of a stack: pushing elements, say, from a list to a stack, then popping them from the stack will reverse the order of the elements. We can make use of that fact for our next program:

Example 8.1.21:

Use a `Stack` and a `Queue` to test whether an input string is a palindrome. Recall that a palindrome is a string that reads the same forwards and backwards. For example, "`racecar`" is a palindrome, while "`Java`" is not.

Use the `Console` class from section 2.4, example 5, to obtain the string from the user (or, for simplicity, hard-code the string into the class).

Our idea here is simple: we obtain the string from the user via the static `readString` method of the `Console` class, then we `enqueue` the characters into a queue, and also `push` the characters into a stack. Then we empty the queue and the stack one character at a time and compare them. If the characters match every time, the string was a palindrome because the stack will reverse the order of characters while the queue will keep the order.

To make matters a little simpler we will use the standard `main` method to fill the stack and queue with characters, while using a separate method

```
boolean isPalindrome(Stack s, Queue q)
```

that checks whether a stack and a queue are the same when elements are removed in their respective order.

A slight problem is that we can not enter single characters into a stack or queue because the basic data type `char` is not an object. But as we mentioned before, we can use the wrapper class `Character`

to wrap a `char` into an appropriate object.[30] Here is the class so far, without implementing the `isPalindrome` method:

```
public class CheckPalindrome
{  public static boolean isPalindrome (Stack s, Queue q)
   { /* code to be added later */ }
   public static void main(String args[])
   {  Stack stack = new Stack();
      Queue queue = new Queue();
      System.out.print("Enter string: ");
      String string = Console.readString();
      for (int i = 0; i < string.length(); i++)
      {  queue.enqueue(new Character(string.charAt(i)));
         stack.push(new Character(string.charAt(i)));
      }
      System.out.println("Palindrome: " + isPalindrome(stack, queue));
   }
}
```

The hard part is the method `isPalindrome`: we will use a loop that removes one element from the stack and one from the queue, then type-cast them into `Character`. If the two characters do *not* match, then we know we can not have a palindrome and the method returns `false`. Otherwise, the loop continues until the stack is empty (the queue will be empty as well since we inserted the same number of characters as into the stack). If the loop finishes without returning `false`, all characters matched and we must have a palindrome. Therefore our method should now return `true`:

```
public static boolean isPalindrome(Stack s, Queue q)
{  while (!s.isEmpty())
   {  try
      {  Character c1 = (Character)s.pop();
         Character c2 = (Character)q.dequeue();
         if (!(c1.equals(c2)))
            return false;
      }
      catch(StackException se)
      {  System.err.println(se); }
      catch(QueueException qe)
      {  System.err.println(qe); }
   }
   return true;
}
```

Note that we can not compare the two `Character` objects `c1` and `c2` with the double-equal operator since it would only check whether `c1` and `c2` are stored at the same memory location. Instead we must use the `equal` method that is part of the `Character` class.

■

Here is another useful application using a stack: a program that checks whether opening and closing parenthesis match. As you have probably noticed many times, it is not completely easy to properly close all parentheses in mathematical (or Java) expressions. We want to create a little program that checks whether each parenthesis that was opened is also closed, and no extra closing brackets are present. In fact, such a program is part of every Java compiler: remove any of the brackets '{', '(', ')', or '}' in the above class and compile to see the error message the Java compiler produces. That is, in fact, no trivial undertaking. Decide quickly, for example, whether all parentheses match properly in the expression below:

[30] Check the Java API for details on `java.lang.Character`.

```
{[(a + (5 - c) + [d * (9 - 2)]}
```

Of course not: the third opening bracket '(' is unmatched, as well as the second bracket '['. We want to create a method to decide exactly that.

Example 8.1.22:

Create a method that can decide whether all parentheses in an expression match correctly. Our program should recognize square, round, and curly brackets, i.e. '[', '(', '{', '}', ')', and ']'. Note: the actual meaning of the expression is irrelevant.

The basic problem is that while every opening bracket must have a corresponding closing bracket, we don't know in which order the brackets are appearing. What we need to do is place all opening brackets in a "holding area" until we find a closing bracket. That closing bracket must match the *last* opening bracket, so we need to check the first closing bracket against the last bracket entered into our holding area. In other words, our holding area must store opening brackets in a LIFO pattern, hence a stack is the perfect structure. Our algorithm will be as follows:

- Go through the input string character by character.
  - If we see an opening bracket, push it on a stack
  - If we see a closing bracket, pop a bracket from the stack and check if they match.
    - If they do not match, we are done - the parentheses do not match: a closing bracket does not have a matching opening bracket.
    - If the stack is currently empty, again we are done - the parentheses do not match: there are not enough opening brackets for all closing ones.
  - Otherwise, continue the process.
  - If at the end of the process the stack is *not* empty, there was an opening bracket that did not have a matching closing bracket, and again the parentheses do not match.
- Otherwise, if the stack is empty, and all characters were checked properly, all parentheses did match properly.

Before we implement that method, let's create a utility class Parenthesis: that class should offer the following capabilities:

- it should store a particular opening parenthesis
  we use a field `parens` of type `char` and set it via the constructor
- it should be able to check whether a character is an opening parenthesis
  we use a `static` method `boolean isOpening(char parens)`
- it should be able to check whether a character is a closing parenthesis
  we use a `static` method `boolean isClosing(char parens)`
- it should check whether the input parenthesis closes the current opening one
  we use a method `boolean matches(char otherParens)`

Thus, our `Parenthesis` class looks as follows:

```
public class Parenthesis
{  private char parens;
   public Parenthesis(char _parens)
   {  parens = _parens; }
   public static boolean isOpening(char parens)
   {  return ((parens == '(') || (parens == '[') || (parens == '{')); }
   public static boolean isClosing(char parens)
   {  return ((parens == ')') || (parens == ']') || (parens == '}')); }
   public boolean matches(char otherParens)
```

```
      {  if ((parens == '(') && (otherParens == ')'))
            return true;
         else if ((parens == '{') && (otherParens == '}'))
            return true;
         else if ((parens == '[') && (otherParens == ']'))
            return true;
         else
            return false;
      }
   }
```

With the help of that class we can follow the above algorithm closely:

```
public static boolean checkParens(String exp)
{  Stack s = new Stack();
   for (int i = 0; i < exp.length(); i++)
   {  char cc = exp.charAt(i);
      if (Parenthesis.isOpening(cc))
         s.push(new Parenthesis(cc));
      if (Parenthesis.isClosing(cc))
      {  try
         {  Parenthesis parens = (Parenthesis)s.pop();
            if (!parens.matches(cc))
               return false;
         }
         catch(StackException se)
         {  return false; }
      }
   }
   return s.isEmpty();
}
```

Note that we are making good use of the exception that is thrown if the stack was empty during a call to `pop`: it means that there is no opening parenthesis for the current closing one, so the parenthesis do not match. Also, at the end we simply return the result of a call to the `isEmpty` method of the stack, because the stack must be empty in order for all opening parentheses to have closing ones. We leave it as an exercise to check whether this method really works.

■

7.4. Java Build-in Sequential Structures

Because of their dynamic structure and flexibility, lists, queues, and stacks are used quite frequently. Since version 1.2 of the JDK, Java therefore provides a complete and extensive coverage of lists, offering an abstract framework to model the abstract data type as well as several concrete implementations of list-like classes. In this section we will discuss this framework and show how the concrete classes can be used. If you do use these build-in classes, your code will *only* run in JVM's that support version 1.2 and better of Java.[31]

[31] See definition 6.2.13 to find out how to ensure that your code is executed by the latest JVM available from Sun, regardless of which version of Java is implement in your favorite web browser.

The Java List Framework

Instead of simply providing a `List` class, Java provides an abstract framework that allows programmers to implement their own version of a concrete `List` class while adhering to the ADT `List` specifications. Programmers could, therefore, optimize an implementation of a `List` that works particularly well for a certain purpose, while being able to replace that optimized structure with another one should they later decide on a better optimization strategy. The code using either type of List will not have to change as long as it uses only the methods specified in the general framework.

The Java list framework is based on two interfaces, the `Collection` and the `List` interface, where `List` extends `Collection`. Java then provides several abstract classes and three concrete ready-to-use classes that implement the `List` interface. All classes and interfaces are part of the `java.util` package. The classes and their inheritance structure are listed in figure 8.3.1.

```
AbstractCollection extends Object implements Collection
|
+--AbstractList extends AbstractCollection implements List
   |
   +---AbstractSequentialList extends AbstractList implements List
   |   |
   |   +--LinkedList extends AbstractSequentialList implements List
   |
   +--ArrayList extends AbstractList implements List
   |
   +--Vector extends AbstractList implements List
```

Figure 8.3.1: The Java `List` *classes and their inheritance structure*

The basic interface that is interesting to us is the `List` interface, which characterizes the Java ADT List, i.e. it specifies the methods that a class needs to implement in order to be classified as a list. The Java ADT List is slightly different from the one chosen by us in definition 8.1.3. There we tried to introduce the very basic idea of what a list is, whereas Java tries to provide a complete and widely useful definition.

Definition 8.3.1: The Java ADT `List` Interface

The `List` *interface forms the specifications for the Java ADT* `List`. *A Java* `List` *is an ordered sequence of elements that supports the following operations:*

```
void add(int index, Object element)        boolean add(Object o)
void clear()                               boolean contains(Object o)
Object get(int index)                      int indexOf(Object o)
int lastIndexOf(Object o)                  boolean isEmpty()
Object remove(int index)                   boolean remove(Object o)
Object set(int index, Object element)      int size()
List subList(int fromIndex, int toIndex)   Iterator iterator()
ListIterator listIterator()                Object[] toArray()
ListIterator listIterator(int start)
```

If a class wants to be considered a `List`, it must implement these methods.[32]

Java provides three ready-to-use classes that implement this `List` interface. Their internal implementations are different from each other and they each have advantages and disadvantages, but they all contain the above methods and are valid lists according to definition 8.3.1.

Definition 8.3.2: The `ArrayList`, `LinkedList`, and `Vector` Classes

Java provides three concrete classes that implement the list interface and adhere to the Java ADT `List`.

- `ArrayList`*: an array-based list similar to the one described in definition 8.1.13*
- `LinkedList`*: a double-linked list similar to the one described in definition 8.1.11*
- `Vector`*: a list similar to `ArrayList` that is synchronized and therefore thread-safe*

All three classes implement the `List` interface and therefore contain the methods listed in definition 8.3.1. Each class has a constructor requiring no input, as well as additional methods as described in table 8.3.2. The classes are part of the `java.util` package and must be imported.

In addition to the methods specified in definition 8.3.1 these classes contain several additional methods as outlined in table 8.3.2. As long as these methods are not used, the three classes are interchangeable, except that `Vector` provides a thread-safe implementation of `List` and should be used if multiple threads access and modify list elements (see definition 8.3.7).

ArrayList	Vector	LinkedList
`ArrayList(int startSize)`	`Vector(int startSize)`	
`void trimToSize()`	`void trimToSize()`	`void addFirst(Object o)`
`void ensureCapacity(int minCapacity)`	`void ensureCapacity(int minCapacity)`	`void addLast(Object o)`
	`Object firstElement()`	`Object getFirst()`
	`Object lastElement()`	`Object getLast()`
	`int indexOf(Object o, int index)`	`Object removeFirst()`
	`int lastIndexOf(Object o, int index)`	`Object removeLast()`
	`void setSize(int newSize)`	

Table 8.3.2: Additional methods for the three concrete list classes

Example 8.3.3:

Create a program that stores several names in an `ArrayList` using, then asks for input from the user to search the list for the specified name. If the name is present, it should return it; otherwise it should print an appropriate message. A `Name` should consist of a first and a last name. For simplicity, you should use the `Console` class to obtain input from the user. Also, determine where the `add` method inserts a new element by default.

Since a "name" should consist of two parts, we first need to implement a `Name` class. It is straightforward:

---

[32] This specification does not seem to provide the methods `first`, `next`, and `isLast` that we required for our list in definition 8.1.4. Instead the Java `List` interface uses an `Iterator` that will provide that functionality as described in definition 8.3.16.

```
public class Name
{  private String first, last;
   public Name(String _first, String _last)
   {  first = _first;
      last = _last;
   }
   public String toString()
   {  return last + ", " + first; }
}
```

Now we can create our program: we instantiate an `ArrayList`, use the `add` method to insert several new names, then ask the user for a first and last name to search for. Finally, we use the `indexOf` method to try to locate a corresponding name (this will, in fact, turn out not to work, but we'll give it a try anyway):

```
import java.util.*;

public class NameList
{  public static void main(String args[])
   {  ArrayList list = new ArrayList();
      list.add(new Name("Bert", "Wachsmuth"));
      list.add(new Name("John", "Doe"));
      list.add(new Name("Jane", "Doe"));
      System.out.print("Enter first name: ");
      String first = Console.readString();
      System.out.print("Enter last name: ");
      String last = Console.readString();
      if (list.indexOf(new Name(first, last)) >= 0)
         System.out.println("found it");
      else
         System.out.println("not found");
   }
}
```

When we execute this program and search for a name we know should exist in the list, figure 8.3.3 shows that our program can not find it.

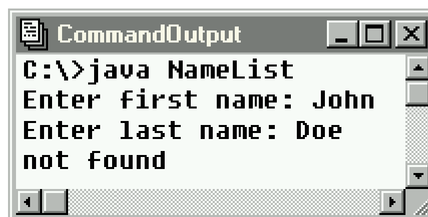


Figure 8.3.3: Output of incorrect `NameList` *program*

Our program can not find the name in the list because it uses the `equals` method that `Name` inherits from `Object`. That default method only checks whether two objects are stored at the same memory location, not whether they are structurally identical. Therefore, the fault in our program is not in its `main` method, but rather in the implementation of the `Name` class. To work properly, `Name` must implement the `equals` method in a way that is appropriate for this class.

Therefore, we add the method `public boolean equals(Object obj)` to the Name class. It type-casts the input object `obj` as a `Name` and then compare the fields `first` and `last` for equality:

```
public boolean equals(Object obj)
{  if (obj instanceof Name)
```

```
    {  Name name = (Name)obj;
       return ((first.equals(name.first)) && (last.equals(name.last)));
    }
    else
       return false;
 }
```

After adding this method to `Name` our `NameList` program will correctly find the name in the list if it is available, as seen in figure 8.3.4.

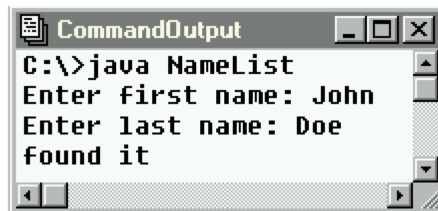


Figure 8.3.4: Output of corrected NameList program

To find out where exactly the `add` method inserts new elements, we simply print out all elements in order:

```
import java.util.*;

public class NameListAndPrint
{  public static void main(String args[])
   {  ArrayList list = new ArrayList();
      list.add(new Name("Bert", "Wachsmuth"));
      list.add(new Name("John", "Doe"));
      list.add(new Name("Jane", "Doe"));
      for (int i = 0; i < list.size; i++)
         System.out.println(list.get(i));[33]
   }
}
```

Figure 8.3.5 shows the output of this program:

Figure 8.3.5: Determining the order of the `add` *method*

The names are printed in the order they were inserted. That implies that the `add` method must insert a new element at the *end* of the list. We do not need to type-cast the objects returned by the get method because polymorphism can select the appropriate `toString` method automatically (compare example 8.1.8).

■

[33] This particular way of looping through the elements of a list works fine for small lists but may be inefficient for longer lists. Instead an `Iterator` should be used for more effective list traversals. See definition 8.3.16 and examples 8.3.15 and 8.3.17 for details.

Since the three concrete list classes contain the same methods, they can be exchanged without changing the basic code.

Example 8.3.4:

Convert the above program `NameListAndPrint` so that it uses first a `LinkedList` and then a `Vector` instead of an `ArrayList`.

Since we did not use any of the special methods of `ArrayList` described in table 8.3.2 we can simply switch the data type for our list from `ArrayList` to `LinkedList`. No further changes are necessary (the changed line is in bold and italics):

```
import java.util.*;

public class NameListAndPrint
{  public static void main(String args[])
   {  LinkedList list = new LinkedList();
      list.add(new Name("Bert", "Wachsmuth"));
      list.add(new Name("John", "Doe"));
      list.add(new Name("Jane", "Doe"));
      for (int i = 0; i < list.size; i++)
         System.out.println(list.get(i));
   }
}
```

Similar, to use a `Vector` instead of a `LinkedList`, we would simply change the line

```
LinkedList list = new LinkedList();
```

to

```
Vector list = new Vector();
```

The output in each case is identical to figure 8.3.5. In particular, new elements are always inserted at the end of the list.

■

Now that we have three versions of a list class that seemingly perform the same, a natural question is when to use which, and what advantages and disadvantages these classes have.

Definition 8.3.5: The Java `List` Classes Compared

Java provides three list classes `ArrayList`, `LinkedList`, *and* `Vector`. *While all implement the* `List` *interface, they have advantages and disadvantages based on their internal implementation*[34]*:*

- `ArrayList` *is an array-based implementation. It should be used if traversal of the list is more frequent relative to insertion and deletion. It is not thread-safe. The time for positioning is independent of the list size. Insertion and deletion are proportional to the list size.*
- `LinkedList` *is a double-linked list implementation. It should be used if insertions and deletions are more frequent relative to the number of traversals. It is not thread-safe. The time for positioning is proportional to the list size. Time for insertion and deletion is independent of list size.*

[34] See the discussion in section 8.1, in particular table 8.1.20

- *`Vector` is a synchronized array-based implementation. It should be used if the structure of the list is concurrently modified by more than one thread.*

The `List` interface has two "search" methods, `indexOf` and `contains`. All three lists implement simple (linear) searching algorithms and may be slow. Both also use the `equals` method of the stored object, which may be the default one inherited from `Object`.

This definition refers to performance issues using the various types of list. It is best understood by creating an example that highlights the various pros and cons for our lists. Example 8.3.3 already illustrated the problem with searching a list when no appropriate `equals` method exists. The next example will clarify which type of list performs fast in what situation.

Example 8.3.6:

Create a program that exercises our three list types as follows:

- add N elements to a list, then remove them again, where N = 1000, 2000, 4000, … 32,000
- create a list of N elements, then perform N retrievals of elements at random positions in the list, where N = 1000, 2000, 4000, …, 32,000

Time each operation for each list type and compare the performance.

We create two methods, one that adds N elements to a list, then removes them again, and another that performs N random retrievals from a list. Both methods will return the time they took to complete their task. Then we embed both methods in appropriate loops, where N = 1000, 2000, 4000, …, 32,000 and display the times in a table. Here is the code:

```
import java.util.*;

public class ListPerformance
{  public static long addRemove(List list, int n)
   {  long start = System.currentTimeMillis();
      for (int i = 0; i < n; i++)
         list.add(new Double(i));
      for (int i = 0; i < n; i++)
         list.remove(0);
      return (System.currentTimeMillis() - start);
   }
   public static long jumpAround(List list, int n)
   {  long start = System.currentTimeMillis();
      for (int i = 0; i < n; i++)
      {  Object obj = list.get((int)(n*Math.random())); }
      return (System.currentTimeMillis() - start);
   }
   public static void addRemovePerformance()
   {  System.out.println("N\tArray\tLinked\tVector");
      for (int n = 1000; n <= 32000; n *= 2)
      {  System.out.print(n + "\t" + addRemove(new ArrayList(), n));
         System.out.print("\t" + addRemove(new LinkedList(), n));
         System.out.println("\t" + addRemove(new Vector(), n));
      }
   }
   public static void accessPerformance()
   {  System.out.println("N\tArray\tLinked\tVector");
      for (int n = 1000; n <= 32000; n *= 2)
      {  ArrayList arrayList = new ArrayList();
```

```
            LinkedList linkedList = new LinkedList();
            Vector vector = new Vector();
            for (int i = 0; i < n; i++)
            {  arrayList.add(new Double(i));
               linkedList.add(new Double(i));
               vector.add(new Double(i));
            }
            System.out.print(n + "\t" + jumpAround(arrayList, n));
            System.out.print("\t" + jumpAround(linkedList, n));
            System.out.println("\t" + jumpAround(vector, n));
         }
      }
      public static void main(String args[])
      {  System.out.println("Insert/Delete Times");
         addRemovePerformance();
         System.out.println("Random Access Times");
         accessPerformance();
      }
   }
```

The output of this program is shown in figure 8.3.6 and explains very clearly what definition 8.3.5 is all about.[35]

```
CommandOutput *
C:\Java\08\8.3.6>java ListPerformance
Insert/Delete Times
N       Array   Linked  Vector
1000    0       0       0
2000    60      0       0
4000    110     0       110
8000    660     50      660
16000   3080    50      3130
32000   12250   170     11540
Random Access Times
N       Array   Linked  Vector
1000    0       50      0
2000    0       170     50
4000    0       770     0
8000    0       3960    0
16000   50      24500   0
32000   60      121160  60
```

Figure 8.3.6: Performance of the three list types

Insertion and deletion works fastest for a `LinkedList`. Increasing N seems to have little impact on its time. For `ArrayList` and `Vector`, larger values of N result in much larger computational times.

Random access to list elements results in the opposite situation. Now `ArrayList` and `Vector` perform fast, with different values of N having no impact on the time. The `LinkedList`, on the other hand, takes much longer as the value of N increases.

This shows that it is worthwhile to think carefully about how a list will be used before choosing one of the available types.

■

The final performance issue that needs clarification is the synchronized nature of a `Vector`. Definition 8.3.5 mentions that `ArrayList` and `LinkedList` are not thread-safe, whereas a `Vector` is.

Definition 8.3.7: Rule of Thumb for Synchronized and Unsynchronized Lists

If a list class is not synchronized and multiple threads add or remove elements from it at the same time, the structure of the list may be compromised. For synchronized lists, such additions or

[35] The particular code we chose highlights the performance differences of our lists. For example, accessing one element in a `LinkedList` is proportional to the list size (it is O(n)), but we perform `n` retrievals, which makes the time $O(n^2)$.

> *deletions are safe. On the other hand, synchronized lists may perform slower than unsynchronized ones.*
> *If exactly one thread accesses a list, you can use an* `ArrayList` *or* `LinkedList`*. If multiple threads access a list, you should use a* `Vector` *or externally synchronize access to the list via an encapsulating class.*[36]

Example 8.3.8:

Create a program that shows that the structure of an unsynchronized list can be corrupted when it is accessed by more than one thread, while using a synchronized list will prevent that problem.

We need to create a program that executes a particular list method and while that method is still executing we need to be able to change the structure of the list without the first method noticing. Therefore we need one list method that executes particularly slow and another that executes quickly. If we used an `ArrayList`, we saw in example 8.3.6 that a particularly slow method is `add`, while access to list elements is provided fast. Therefore, our idea is as follows:

- we create one "slow" thread that will `add` an element to an existing list at the first position, then display the first element of the list[37]
- while that method is executing we start a second "fast" thread that will change the first element in the list, then display its value.

Since the first operation is slow while the second one is fast we might be able to change the structure of the list using the second operation before the first operation is finished. Here is the code that will accomplish this:

- The main class will simply create a large list of type `ArrayList` and start the "slow operation" thread:

```
import java.util.*;

public class CorruptedList
{  public static void main(String args[])
   {  ArrayList list = new ArrayList();
      System.out.println("Creating list, please wait ...");
      for (long i = 0; i < 1000000; i++)
         list.add(new Double(i));
      System.out.println("List created ...");
      SlowOperation slow = new SlowOperation(list);
   }
}
```

- The "slow operation" thread will start a "fast operations" thread then `add` an element to the list at the first position:

```
class SlowOperation extends Thread
{  private List list = null;
   public SlowOperation(List _list)
   {  list = _list;
      System.out.println("Starting slow operation ...");
      start();
```

[36] One of the exercises will ask you to create a synchronized version of our custom `Queue` from definition 8.1.14.
[37] Based on definition 8.1.13 it is clear that adding a new element at the first position of an array-based list will take particularly long because all existing elements need to be moved up by one before a new one can be inserted.

```
    }
    public void run()
    {  FastOperation fast = new FastOperation(list);
       list.add(0, new Double(-10));
       System.out.println("Slow thread sees element: " + list.get(0)); }
}
```

- The "fast operations" thread will change the first element in the list, then display the value of the first element in the list:

```
class FastOperation extends Thread
{  private List list = null;
   public FastOperation(List _list)
   {  list = _list;
      System.out.println("Starting fast operation ...");
      start();
   }
   public void run()
   {  list.set(0, new Double(10));
      System.out.println("Fast thread sees element: " + list.get(0));
   }
}
```

Figure 8.3.7 shows that we have a problem:

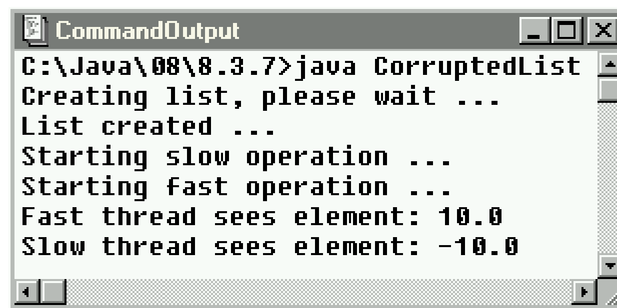


Figure 8.3.7: Output of `CorruptedList` *class using* `ArrayList`

- the slow thread executes first and adds –10 as the first element
- the fast thread executes second and changes the value of the first element to 10
- both threads should see 10 as the first value in the list, but they do not

The `add` operation takes a long time to finish, and while it is executing the second thread quickly changes what is currently the first element of the list. Then the original thread finishes and adds –10 as the first element. The element changed by the second thread has now moved to second position even though the fast thread changed the first element in the list. In particular, figure 8.3.7 shows that the fast thread started second but displays its value of 10 first. After that the slow thread thinks the first element is –10 even though it started before the fast thread.

If we used a synchronized list this problem should go away. As soon as the slow thread starts its `add` operation, a synchronized list will block all access to the list until that `add` operation is finished. All we have to do is change the list type from `ArrayList` to `Vector`:

```
public class SynchronizedList
{  public static void main(String args[])
   {  Vector list = new Vector();
      /* rest unchanged */
   }
```

```
}
class SlowOperation extends Thread
{  /* no change from above */ }
class FastOperation extends Thread
{  /* no change from above */ }
```

The output of running this modified program using a synchronized list is shown in figure 8.3.8:

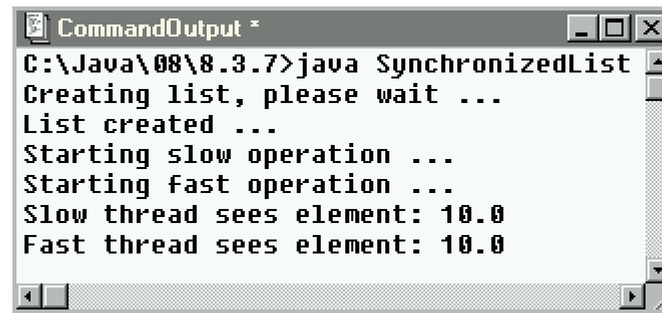


Figure 8.3.8: Output of `SynchronizedList` *using* `Vector`[38]

Again the `add` operation of the slow thread starts first but it blocks access to the list. Therefore, the fast thread is not allowed access to the list elements and is put on hold. When the `add` operation is finished, the fast thread can continue to change the newly added element in the first position. Everything continues to be in sync.

■

Example 8.3.9:

In example 4.38 we created a rather extensive address book program to add, delete, and sort addresses. One drawback of the program we created is that it could store a maximum of 100 addresses only. Change whatever is necessarily in that program so that the number of addresses that the new program can handle is limited only by available memory.

First let's review the components that made up the address book program in example 4.38. There were several utility classes:

`Sortable`:	an interface for sorting an array (as defined in example 4, section 3.6)
`Sorter`:	class to sort an array of `Sortable` items (from example 4, section 3.6)
`YesNo`:	a message dialog with “Yes/No” buttons (see example 4.37)
`ButtonRow`:	a class to arrange buttons in a `FlowLayout` (see example 4.36)

Then there were the classes specifically created for the address book program:

`Address`	represents a "public" address
`PrivateAddress`	represents a "private" address
`AddressBook`	stores different types of addresses in a list-like structure
`AddressDialog`	dialog to add and modify addresses
`AddressBookGUI`	layout of main program, controls action

[38] If you execute `SynchronizedList` several times, the fast thread may occasionally display its element first. It may even happen that the slow thread displays –10 because the fast thread could not change the value before the slow one printed it out. But in this synchronized version it should not happen that the fast thread shows 10 and then the slow thread shows –10.

AddressApplet allows the program to run as an applet

The class that is responsible for storing the addresses and which limits the number of possible addresses is clearly AddressBook, so let's review that class in detail:

```
public class AddressBook
{  private static int MAXNUMBER = 100;
   private Address[] addresses = null;
   private int currentNumber = -1;
   public AddressBook()
   {  addresses = new Address[MAXNUMBER]; }
   public void add(Address newAddress)
   {  if (currentNumber < MAXNUMBER-1)
      {  currentNumber++;
         addresses[currentNumber] = newAddress;
      }
   }
   public void del(int pos)
   {  if (pos < currentNumber)
         for (int i = pos + 1; i <= currentNumber; i++)
            addresses[i-1] = addresses[i];
      currentNumber--;
   }
   public Address get(int pos)
   { return addresses[pos]; }
   public void sort()
   {  Sorter sorter = new Sorter();
      sorter.sort(addresses, currentNumber);
   }
}
```

We now want to change that class so that it uses a list instead of an array to store the various address types. We first need to decide whether to use an ArrayList, a LinkedList, or a Vector. There are clearly no synchronization problems, so we do not need a Vector. A typical use of an address book program is to look up addresses, while new addresses will not be added as frequently once a set of core addresses is available. Therefore we decide to use an ArrayList. To ensure that all other classes can remain unchanged we keep all public method headers of the AddressBook class intact but change their inner workings as follows:

```
import java.util.*;

public class AddressBook
{  private ArrayList addresses = null;
   public AddressBook()
   {  addresses = new ArrayList(); }
   public void add(Address newAddress)
   {  addresses.add(newAddress); }
   public void del(int pos)
   {  addresses.remove(pos); }
   public Address get(int pos)
   { return (Address)addresses.get(pos); }
   public void sort()
   {  Sorter sorter = new Sorter();
      Sortable addressArray[] = new Sortable[addresses.size()];
      for (int i = 0; i < addresses.size(); i++)
         addressArray[i] = (Sortable)addresses.get(i);
      sorter.sort(addressArray);
      for (int i = 0; i < addressArray.length; i++)
         addresses.set(i, addressArray[i]);
   }
```

```
}
```

The new class is significantly easier than the previous version and it can store as many addresses as memory permits. The only method that has gotten more complicated, and in fact slower, is `sort`. That is due to shortcomings of our old `Sorter` class from example 4, section 3.6. If we improved that class so that it could sort lists in addition to arrays, the new `AddressBook` class would be completely easy (see exercise).

Nothing else has to change. If you copy – without change – all classes into one directory and replace the old `AddressBook` class by our new version, the program will function just as before but is now able to handle a lot more addresses.

■

In addition to standard `List` types Java also provides a `Stack` class, which we will introduce next.

Definition 8.3.10: The `Stack` Class

This class implements a LIFO (Last-In-First-Out) structure, as described in definition 8.1.18. The Java API defines the class as follows:

```
public class Stack extends Vector
{  // constructor
   Stack()
   // methods
   public boolean empty()
   public Object peek()
   public Object pop()
   public Object push(Object item)
   public int search(Object o)[39]
}
```

Since this class extends `Vector`*, it provides synchronized access to its elements. The class is part of the* `java.util` *package and must be imported.*

Strictly speaking, this implementation of a `Stack` is not appropriate: since `Stack` extends `Vector`, all public methods available to `Vector` are also available to a Java standard `Stack`. Hence, the integrity of a `Stack` could be destroyed by using methods that are not appropriate for a stack. It is left to the programmer to avoid using those "non-stack" methods.

Example 8.3.11

Redo example 8.1.20 where we pushed the strings "`Java`", "`by`", "`Definition`" into a `Stack`, then `popped` them and printed the results.

The Java build-in Stack does not use exceptions, different from our own Stack from definition 8.1.18. Therefore we need to adjust the code from example 8.1.20 slightly since we do not need to catch any exceptions. The following simple code will work perfectly fine:

```
import java.util.*;

public class StackTest
{  public static void main(String args[])
```

[39] This method returns the height of the object from the top of the stack, if it is found, or –1 otherwise.

```
    {  Stack s = new Stack();
       s.push("Java");
       s.push("by");
       s.push("Definition");
       while (!s.isEmpty())
          System.out.println(s.pop());
    }
}
```

Of course it will display the strings that were pushed into the stack in reverse order since a stack is a LIFO structure.

■

There is no predefined `Queue` class but it is easy to define one that extends `List`. Just as for the build-in `Stack` class that approach is not entirely legal, but since it is so easy to do it is difficult to pass up that opportunity. The details are left as an exercise.

StringTokenizer Class

Our final example of a sequential structure is the one created by the `StringTokenizer` class. That class is quite useful in many situations and is well worth remembering in detail.

Definition 8.3.12: The `StringTokenizer` Class

This class breaks a string into tokens depending on a set of specified delimiters. The tokens can be retrieved in order, either including or excluding the delimiters. For example, if the string "Java by Definition" is tokenized using the space character as delimiter, it results in the five tokens "Java", `SPACE`*, "by",* `SPACE`*, and "Definition". The Java API defines the* `StringTokenizer` *class as follows:*

```
public class StringTokenizer extends Object implements Enumeration
{  // constructor
   StringTokenizer(String str)
   StringTokenizer(String str, String delim)
   StringTokenizer(String str, String delim, boolean returnTokens)
   // selected methods
   int countTokens()
   boolean hasMoreTokens()
   String nextToken()
}
```

Each character in the second and third constructor's input string `delim` *will act as a delimiter to break up the original string. The default delimiters are "* `\t\n\r\f`*", i.e. space, tab, new line, return, and form-feed. The class is part of the* `java.util` *package and must be imported.*

The `StringTokenizer` class is frequently used to break apart strings that contains several data values separated by some kind of delimiter. Such data is quite common. Microsoft Excel, for example, can save spreadsheet data in such delimited form and a `StringTokenizer` could be used to separate the individual values from each other.

Example 8.3.13:

Consider the String "10|-1|41". Use the StringTokenizer class to extract the three numbers from the string, convert them to integers, and add them up.

Without the StringTokenizer we would have to search through the string and locate the delimiter character '|' manually. The StringTokenizer class simplifies and automates that process:

```
import java.util.*;

public class SumOfTokens
{  public static void main(String args[])
   {  String s = "10|-1|41";
      StringTokenizer tokens = new StringTokenizer(s, "|");
      int sum = 0;
      while (tokens.hasMoreTokens())
         sum += Integer.parseInt(tokens.nextToken());
      System.out.println("Sum is: " + sum);
   }
}
```

■

The next example uses multiple delimiters to tokenize a string and will be pursued further in the exercises:

Example 8.3.14:

Use a StringTokenizer class to break up a mathematical expression into tokens. The delimiters should be the standard binary operators '+', '-', '/', '*', and '^' as well as the standard parenthesis ')' and '('. The delimiters should be included in the list of tokens. Use the Console class to get the expression from the user and print out the list of tokens after the user enters an expression.

We will use a while loop to ask for user input until the user enters "q". Otherwise, we will instantiate a new StringTokenizer object, using the expression as input and the string "+-/*^()" as delimiters. We also add a boolean value to the constructor to indicate that we want the operators as part of the token list:

```
import java.util.*;

public class MathTokenize
{  public static void main(String args[])
   {  String input = new String("");
      String delim = "+-/*^()";
      while (!input.equals("q"))
      {  StringTokenizer tokens = new StringTokenizer(input, delim, true);
         while (tokens.hasMoreTokens())
            System.out.println(tokens.nextToken().trim());
         System.out.print("Enter expression [q to quit]: ");
         input = Console.readString();
      }
   }
}
```

A sample run of this program is shown in figure 8.3.9.

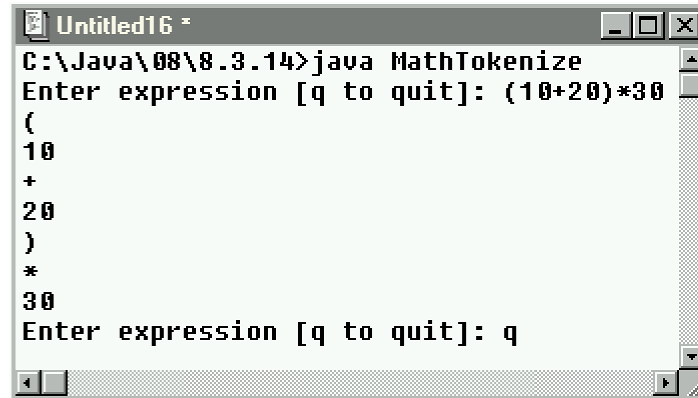


Figure 8.3.9: Sample run of `MathTokenize`

■

Iterators

When using sequential structures such as lists we often need to provide code that traverses the entire list to process each element. With our own hand-made list from definition 8.1.4 that can be accomplished with the methods `first`, `next`, and `isLast`. Neither of the standard Java `List` structures from definition 8.3.1, however, provide a `next` method, so a different approach is necessary to traverse these lists. At first glance the solution seems easy: use the method `Object get(int index)` together with `size()` as in the following code:

```
for (int i = 0; i < list.size(); i++)
{  Object obj = list.get(i);
   // process the objects
}
```

As usual, `list` could be of type `ArrayList`, `LinkedList`, or `Vector`. But for large lists this code could be very slow, especially for a `LinkedList`. After all, we have seen in example 8.3.6 that accessing elements of a `LinkedList` is a particularly slow operation.

Example 8.3.15:

Create a program that creates a `LinkedList` containing 10,000 random double values. Then traverse the list in a loop to find the average of these numbers. If you created a similar program using our own type of List as defined in definition 8.1.4, which program executes faster?

Both programs are very simple. The first can use a `for` loop to compute the average, while the second needs to use a `while` loop. Here is the code for both classes:

```
import java.util.*;

public class TraverseLinkedList
{  public static void main(String args[])
   {  int SIZE = 10000;
      System.out.println("Creating list ...");
      LinkedList list = new LinkedList();
      for (int i = 0; i < SIZE; i++)
         list.add(new Double(Math.random()));
```

```
      System.out.println("Computing average ...");
      long start = System.currentTimeMillis();
      double sum = 0.0;
      for (int i = 0; i < SIZE; i++)
         sum += ((Double)list.get(i)).doubleValue();
      long stop  = System.currentTimeMillis();
      System.out.println("Average: " + sum / SIZE);
      System.out.println("Time   : " + (stop - start));
   }
}

public class TraverseCustomList
{  public static void main(String args[])
   {  int SIZE = 10000;
      System.out.println("Creating list ...");
      List list = new List();
      for (int i = 0; i < SIZE; i++)
         list.insert(new Double(Math.random()));
      System.out.println("Computing average ...");
      try
      {  long start = System.currentTimeMillis();
         double sum = 0.0;
         list.first();
         while (!list.isLast())
         {  sum += ((Double)list.retrieve()).doubleValue();
            list.next();
         }
         sum += ((Double)list.retrieve()).doubleValue();
         long stop  = System.currentTimeMillis();
         System.out.println("Average: " + sum / SIZE);
         System.out.println("Time   : " + (stop - start));
      }
      catch(ListException le)
      {  System.err.println("List Exception: " + le); }
   }
}
```

The code using our custom `List` is slightly more complicated but as figure 8.3.10 shows it is *a lot* faster than the program using the build-in `LinkedList` structure.[40]

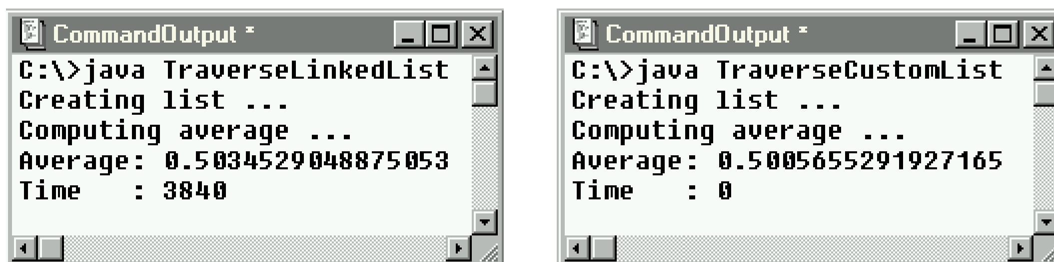


Figure 8.3.10: Comparing TraverseLinkedList with TraverseCustomList

■

To remedy this problem and to simply traversal of a list, Java offers an `Iterator` interface. Support for that interface is build in to all list structures.

[40] In "big-O" notation the get method for a `LinkedList` is O(n) which makes the code for `TraverseLinkedList` a O(n2) operation. A similar loop for `TraverseCustomList` is only O(n).

Definition 8.3.16: The `Iterator` Interface

This interface simplifies sequential processing of any standard Java list structure by providing access to the list elements in sequential order. The Java API defines the `Iterator` *as follows:*

```
public abstract interface Iterator
{  boolean hasNext()
   Object next()
}
```

The `next` *method returns the current object and advances an internal pointer to point to the next element in the list. An* `Iterator` *can be established by using the* `iterator` *method that is part of the* `List` *interface and hence part of all standard Java* `List` *structures. An* `Iterator` *allows sequential processing of the elements in a list via a fast and simple* `for` *loop:*

```
for (Iterator iterator = list.iterator(); iterator.hasNext(); )
   System.out.println(iterator.next());
```

Example 8.3.17:

Redo example 8.3.15 but use an `Iterator` for the `TraverseLinkedList` program. Compare the resulting times against those displayed in figure 8.3.10.

The new program should use an `iterator` instead of the `list.get` method. Using a `for` loop as in definition 8.3.16 the code will change as follows, with the changes in bold and italics:

```
import java.util.*;

public class TraverseLinkedListWithIterator
{  public static void main(String args[])
   {  int SIZE = 10000;
      System.out.println("Creating list ...");
      LinkedList list = new LinkedList();
      for (int i = 0; i < SIZE; i++)
         list.add(new Double(Math.random()));
      System.out.println("Computing average ...");
      long start = System.currentTimeMillis();
      double sum = 0.0;
      for (Iterator iterator = list.iterator(); iterator.hasNext(); )
         sum += ((Double)iterator.next()).doubleValue();
      long stop  = System.currentTimeMillis();
      System.out.println("Average: " + sum / SIZE);
      System.out.println("Time    : " + (stop - start));
   }
}
```

The changes were minimal, yet the result is dramatically faster, as shown in figure 8.3.11.

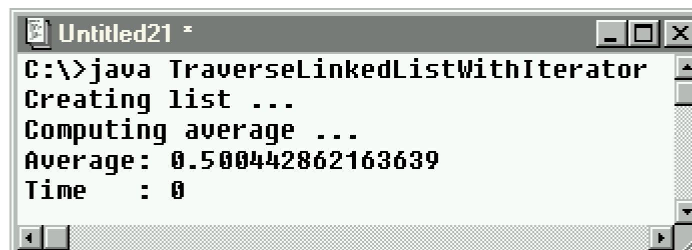


Figure 8.3.11: Timing the `TraverseLinkedListWithIterator` *program*

Now our `LinkedList` can be traversed just as fast as the custom `List`, or in fact as fast as an `ArrayList`. It might take a little practice, especially since the modification piece of the `for` loop is moved inside the loop, but it is well worth using an `Iterator` to speed up sequential access to the build-in list structures.

■

Here is one more example using an `Iterator` to get used to the funny `for` loop it uses.

Example 8.3.18:

In example 8.3.3 we created a `Name` class to handle a first and last name. Create a list of `Name` objects, then use the `Iterator` interface to print out the complete list.

The code to create the list is just as it was in the previous exercises. We then use the sample code from the above definition to process each list element. Note that we can use an `ArrayList`, a `LinkedList`, or a `Vector` as the underlying list structure. Let's use a `Vector` for variety:

```
import java.util.*;

public class NameIterator
{  public static void main(String args[])
   {  Vector list = new Vector();
      list.add(new Name("Bert", "Wachsmuth"));
      list.add(new Name("John", "Doe"));
      list.add(new Name("Jane", "Doe"));
      for (Iterator iterator = list.iterator(); iterator.hasNext(); )
         System.out.println(iterator.next());
   }
}
```

■

An `Iterator` allows processing the elements in a list in forward order only, because the `next` method will advance the internal pointer to the next element. A second type of `Iterator` is more flexible, allowing forward and backward processing of a list.

Definition 8.3.19: The `ListIterator` Interface

This interface simplifies sequential processing of any standard Java list structure in forward and backward order by providing access to the list elements in sequential order. The Java API defines the `ListIterator` *as follows:*

```
public abstract interface ListIterator extends Iterator
{  boolean hasNext()
   boolean hasPrevious()
   Object next()
   Object previous()
}
```

A `ListIterator` *can be established by using one of the two* `listIterator` *method that are part of the* `List` *interface and hence part of all standard Java list structures. A* `ListIterator` *allows sequential processing of the elements in a list in forward as well as reverse direction via fast and simple* `for` *loops:*

Forward order:

```
for (ListIterator iter = list.listIterator(0);
                   iter.hasNext(); )
   System.out.println(iter.next());
```

Backwards order:

```
for (ListIterator iter = list.listIterator(list.size());
                   iter.hasPrevious(); )
   System.out.println(iter.previous());
```

Example 8.3.20:

Redo example 8.3.18, using a `ListIterator` instead of an `Iterator`, and print the list of names first in regular order, then backwards.

All we have to do is change the type and instantiation of `Iterator` to `ListIterator`, providing a starting index of 0 to start the iteration at the beginning of the linked list for the forward direction and at `list.size()` for reverse processing.

```
import java.util.*;

public class NameListIterator
{  public static void main(String args[])
   {  LinkedList list = new LinkedList();
      list.add(new Name("Bert", "Wachsmuth"));
      list.add(new Name("John", "Doe"));
      list.add(new Name("Jane", "Doe"));
      System.out.println("Printing list forward");
      for (ListIterator iter = list.listIterator(0);
                         iter.hasNext(); )
         System.out.println(iter.next());
      System.out.println("Printing list backwards");
      for (ListIterator iter = list.listIterator(list.size());
                         iter.hasPrevious(); )
         System.out.println(iter.previous());
   }
}
```

We could redo this example so that the second loop picks up right where the first loop terminates:

```
import java.util.*;

public class NameListIterator
{  public static void main(String args[])
   {  LinkedList list = new LinkedList();
      list.add(new Name("Bert", "Wachsmuth"));
      list.add(new Name("John", "Doe"));
      list.add(new Name("Jane", "Doe"));
      ListIterator iter = list.listIterator(0);
      System.out.println("Printing list forward");
      for ( ; iter.hasNext(); )
         System.out.println(iter.next());
      System.out.println("Printing list backwards");
      for ( ; iter.hasPrevious(); )
         System.out.println(iter.previous());
   }
}
```

The `for` loops in this version may look strange. But if we consider how our custom `List` contained a reference to the `current` element of the list it is easy to understand how `Iterator` and `ListIterator` work.

■

Now we have sophisticated data structures at our disposal that we could use in place of arrays whenever a dynamical structure is needed. We can now move on to the last segment of this text, file processing and network computing.

Case Study: A Mathematical Expression Parser

Our next example will use stacks and queues extensively in a calculator program where the user can enter the expression to be computed in one line, including parentheses. Our program will analyze the input, evaluate it if possible, and compute the answer or display an appropriate error message. This is significantly more complicated than creating a calculator based on a graphical user interface with buttons for the numbers and operators[41]. The advantage of a button-based calculator is that the program knows by virtue of the buttons pressed whether the user has entered a number or an operator. If we allow the user to enter an entire expression we first need to take that expression apart and determine where the numbers and where the operators are. In addition, the user could commit a wide variety of errors by incorrectly entering numbers, operators, and parenthesis and our program has to be prepared to catch any irregularity, while still evaluating proper expressions correctly. Our new calculator program should, for example, correctly evaluate the expression:

```
((20.3 - 11.2) * 2 / (3 + 5)) * 2^(3-5)
```

which evaluates to 0.56875, while recognizing an expression like:

```
((20.4 - 11.2) * 2 / (3 + 5) * 2^(3-)
```

as incorrect (the last term (3-) is incorrect and the parentheses do not match either). Our "line input" calculator will be somewhat involved, so we will proceed in three stages.

Stage 1: The Tokenizer

Example 8.2.1:

Our goal is to create a calculator that can evaluate expressions involving numbers, standard round parenthesis, and the binary operators +, -, *, /, and ^, using the "natural" order of precedence of these operators. As a first stage towards this goal, create a method that analyzes a string representing a mathematical expression and categorizes the expression as numbers and operators.

What we want to do is start with an input string such as, for example, `10 + 20`, and end up, somehow, with a classification such as "number 10", "operator +", "number 20". Therefore, we need two different kind of tokens, or signifiers: one to indicate a valid "number", and another to indicate a valid "operator". Thus, we will create three classes before anything else:

- a superclass `Token`
- a class `TokenNumeric` as a subclass of `Token`

[41] We have already created such a button-based calculator in example 4.5.8.

- a class `TokenOperator` as a subclass of `Token`

The `Token` class will, for now, only contain the method `toString`. We don't have any information to arrive at the string representation of a `Token`, so we will declare that method as `abstract` (and hence the `Token` class will be abstract)

```
public abstract class Token
{  public abstract String toString(); }
```

To implement `TokenNumeric` and `TokenOperator` we will allow for the possibility of constructing invalid versions, i.e. attempts to create a `TokenNumeric` for an invalid number, or a `TokenOperator` for an invalid operator. Therefore we need a `TokenException` class which can be thrown when constructing an invalid token. Here is the exception class, as usual:

```
public class TokenException extends Exception
{  public TokenException()
   {  super("Invalid token"); }
   public TokenException(String msg)
   {  super(msg); }
}
```

The purpose of `TokenNumeric` is to store a number. Thus, the class needs a field to store the number it represents as well as a constructor to set the number. Since the number will be constructed from an input string, we will overload the constructor to either take a number as input or a string which will get converted to a number if possible. If the conversion is not possible, a `TokenException` is thrown. To retrieve the stored number, we add an appropriate `getNumber` method. Finally, we implement the `toString` method to generate the `String` representation of the number. Here is the code:

```
public class TokenNumeric extends Token
{  private double number = 0.0;
   public TokenNumeric(double _number)
   {  number = _number; }
   public TokenNumeric(String _number) throws TokenException
   {  try
      {  number = Double.valueOf(_number).doubleValue();
      }
      catch(NumberFormatException ne)
      {  throw new TokenException("Invalid numeric format: " + _number);
      }
   }
   public double getNumber()
   {  return number; }
   public String toString()
   {  return String.valueOf(number); }
}
```

As for the `TokenOperator` class, it clearly needs to store an "operator", so we will use a field of type `String` for that. Only the operators '+', '-', '/', '*', '^' are valid, but we will also consider the parentheses ')' and '(' as valid operators. To define these valid operators, we will use a constant string `OPERATORS`. The operator is set via the constructor and if it is not valid a `TokenException` is thrown. We will also implement a `static` method `isValidOp` that determines if the input character represents a valid operator. The `toString` method creates an appropriate string representation of our operator. We really should create a method `getOperator` to access the operator but as it will turn out later that will not be necessary:

```
public class TokenOperator extends Token
```

```
{  private static String OPERATORS  = "+-/*^()";
   private String token;
   public TokenOperator(String _token) throws TokenException
   {  if (OPERATORS.indexOf(_token) >= 0)
         token = _token;
      else
         throw new TokenException("invalid operator: " + _token);
   }
   public static boolean isValidOp(char op)
   {  return (OPERATORS.indexOf(op) >= 0); }
   public String toString()
   {  return "Operator: " + token; }
}
```

Now we are in a position to break up the input expression and classify it as either `TokenNumeric` or `TokenOperator`. We will collect these tokens in a queue in the order they appear in the input string and later analyze them for correct syntax. Our `static` method `tokenize` will work as follows:

- Create an empty queue `Q` and an empty string `token`.
- Process the input expression one character at a time.
  - If the current character is not a valid operator, it is assumed to be part of a number, so append it to the string `token`.
  - Otherwise the number currently stored in token is complete, so we try to construct a `TokenNumeric` object from it and insert it into the queue. Also, we try to construct a `TokenOperator` from the current character and `enqueue` it as well. Finally we reset the string `token` to be empty to receive the next number.

Here is the method implementing this algorithm:

```
protected static Queue tokenize(String input) throws TokenException
{  Queue  Q = new Queue();
   String token = new String("");
   for (int i = 0; i < input.length(); i++)
   {  char cc = input.charAt(i);
      if (!TokenOperator.isValidOp(cc))
         token += cc;
      else
      {  if (!token.trim().equals(""))
            Q.enqueue(new TokenNumeric(token.trim()));
         Character operator = new Character(cc);
         Q.enqueue(new TokenOperator(operator.toString()));
         token = new String("");
      }
   }
   if (!token.trim().equals(""))
      Q.enqueue(new TokenNumeric(token.trim()));
   return Q;
}
```

To test our method, we embed it in a simple test class, using the `readString` method of `Console` to get the input from the user, `tokenize` the input string, and print out the queue that is created. Let's call our test class `Calculator`, in view of our final goal:

```
public class Calculator
{  private static Queue tokens = null;
   protected static Queue tokenize(String input) throws TokenException
   { /* as above */ }
   public static void main(String args[])
   {  System.out.print("Calc [q to quit] => ");
```

```
            String input = Console.readString();
            while (!input.equalsIgnoreCase("q"))
            {  try
               {  tokens = tokenize(input);
                  while (!tokens.isEmpty())
                     System.out.println(tokens.dequeue());
               }
               catch(TokenException te)
               {  System.err.println(te); }
               catch(QueueException qe)
               {  System.err.println(qe); }
               System.out.print("Calc [q to quit] => ");
               input = Console.readString();
            }
         }
      }
```

At this state the program will correctly `tokenize` any input string that represents a valid expression. It will also process *some* mathematically incorrect expressions, while throwing an exception for others

- everything between two valid operators must represent a valid number
- valid operators in sequence will be classified correctly, even though mathematically incorrect

For example, the input strings "`10 + 20.44`" and "`-((10 - 20) -+* (20 - 49`" will both be processed correctly even though the second does not make sense mathematically. Expressions such as "`10.b + 99`" or "`10 20 30`" or "`10 % 20`" are all recognized as invalid.

■

At this stage we can differentiate between numbers and operators and all numbers and operators that make it into the queue will be valid and proper (but not necessarily in the right order). Our next stage could be to try to evaluate the tokens. However, the way we normally enter valid mathematical expressions is not easy to evaluate via a computer algorithm. Therefore, we will first convert the queue of valid tokens into another queue based on the order of precedence of the operators and the level of parentheses. That new queue will turn out to be quite easy to evaluate. As a side benefit, the process of converting one queue into the other can also check whether the expression is mathematically correct, at least for the most part. The two different orderings of the queues, or mathematical expressions, are known as "infix" and "postfix" notation.

Stage 2: Converting to Postfix

Definition 8.2.2: infix and postfix Notation

Infix and postfix notations are two different ways to denote mathematical expressions.

- *In infix notation, two numbers are always separated by operators. The order in which an infix expression is to be evaluated depends on the order of precedence of the various operators as well as on grouping symbols such as standard parentheses.*
- *In postfix notation, numbers occur first, followed by operators that are relevant to numbers preceding the operator. There are no parentheses in postfix notation and the order in which a postfix expression is evaluated is always from left to right.*

Either notation can be converted into the other. Infix notation is the "usual" way we write a mathematical expression, while the equivalent postfix notation is easier to evaluate via a computer algorithm. Before we continue, here are a few examples:

infix Notation	postfix Notation
`10.5 + 20.3`	`10.5 20.3 +`
`10 + 20 * 30`	`10 20 30 * +`
`10 * 20 + 30`	`10 20 * 30 +`
`(10 + 20) * 30`	`10 20 + 30 *`
`10 * 3^2 -1`	`10 3 2 ^ * 1 -`
`10 * 3^(2 - 1)`	`10 3 2 1 - ^ *`

Table 8.2.1: Examples of expression in infix and postfix notation

It takes some practice to convert infix to postfix notation "by hand", but once postfix notation is mastered it does offer plenty of advantages. In fact, older Hewlett Packard calculators used "postfix" notation exclusively, i.e. you *had* to enter an expression in postfix notation in order for these calculators to compute the answer. They we very popular among engineers and scientists, because complex calculations could be entered quite efficiently. Today, however, infix notation is the predominant way to enter mathematical expression, even though many computers programs (and calculators) still convert the expression to postfix notation internally before evaluating it.

Example 8.2.3:

Our goal is to create a calculator that can evaluate expressions involving numbers, standard round parenthesis, and the binary operators +, -, *, /, and ^, using the "natural" order of precedence of these operators. In example 7.2.1 we created a `tokenize` method to turn a string representing a mathematical expression into a queue of tokens. Now create a method that converts a queue of tokens in infix notation into the corresponding queue of tokens in postfix notation.

To convert infix to postfix notation, we need to worry about the order of precedence of our mathematical operators. For example, the expressions "`10 + 20 * 30`" and "`10*20 + 30`" are converted quite differently in table 8.2.1. Here is an algorithm for converting infix to postfix (assuming, for now, that there are no parenthesis to change the order of precedence):

The assumption is that our expression has been tokenized and is available in a queue. We need a "postfix queue" to contain the resulting expression in postfix notation, and an "operator stack" to ensure that operators are entered into the postfix queue in the correct order:

- start with the first token in an expression queue, an empty result queue, and an empty operator stack
  - if the token is a number, `enqueue` it into the result queue
  - otherwise (token is an operator) do the following:
    - if the operator stack is empty, push it into the operator stack
    - otherwise (operator stack not empty) do the following:
      - if the token at the top of the operator stack has precedence less than the current token, `push` the current token into the stack
      - otherwise `pop` all tokens from the stack whose precedence is greater than or equal to the current token and `enqueue` them into the result queue. Then `push` the current token into the stack
  - continue this process with the next token
- when all tokens in the expression queue are processed, `pop` all remaining tokens from the operator stack (if any) and `enqueue` them into the postfix queue.

Let's apply this algorithm to the expression "10 + 20 * 30":

State	Description
infix: 10 | num, + | op, 20 | num, * | op, 30 | num postfix: op-stack:	At the start, postfix queue and operator stack are empty
infix: + | op, 20 | num, * | op, 30 | num postfix: 10 | num op-stack:	First token is 10 (number), so it goes into the postfix queue
infix: 20 | num, * | op, 30 | num postfix: 10 | num op-stack: + | op	Next token is + (operator) and stack is empty, so it goes into the stack
infix: * | op, 30 | num postfix: 10 | num, 20 | num op-stack: + | op	Next token is 20 (number), so it goes into the postfix queue
infix: 30 | num postfix: 10 | num, 20 | num op-stack: * | op, + | op	Next token is * (operator) and token at top of stack has lesser precedence, so it goes into stack
infix: postfix: 10 | num, 20 | num, 30 | num op-stack: * | op, + | op	Next token is 30 (number), so it goes into the postfix queue
infix: postfix: 10 | num, 20 | num, 30 | num, * | op, + | op op-stack:	Finally, all operators are popped and put in postfix queue

Let's apply the same algorithm to the expression "10 * 20 + 30":

State	Description
infix: 10 | num, * | op, 20 | num, + | op, 30 | num postfix: op-stack:	At the start, postfix queue and operator stack are empty
infix: * | op, 20 | num, + | op, 30 | num postfix: 10 | num op-stack:	First token is 10 (number), so it goes into the postfix queue
infix: 20 | num, + | op, 30 | num postfix: 10 | num op-stack: * | op	Next token is * (operator) and stack is empty, so it goes into the stack
infix: + | op, 30 | num postfix: 10 | num, 20 | num op-stack: * | op	Next token is 20 (number), so it goes into the postfix queue
infix: 30 | num postfix: 10 | num, 20 | num, * | op op-stack: + | op	Next token is + (operator) but token at top of stack has higher precedence. Stack is popped and + is pushed into it
infix: postfix: 10 | num, 20 | num, * | op, 30 | num op-stack: + | op	Next token is 30 (number), so it goes into the postfix queue
infix: postfix: 10 | num, 20 | num, * | op, 30 | num, + | op op-stack:	Finally, all operators are popped and put in postfix queue

Ignoring any exceptions, this algorithm translates roughly into the following Java code (it will not yet compile):

```
protected static Queue toPostfix(Queue infix) throws TokenException
{  Stack stack   = new Stack();
   Queue postfix = new Queue();
   while (!infix.isEmpty())
   {  Object token = infix.dequeue();
      if (token instanceof TokenNumeric)
         postfix.enqueue(token);
      else if (stack.isEmpty())
         stack.push(token);
```

```
         else if (stack.peekToken().order() < token.order())
            stack.push(token);
         else
         {  while ((!stack.isEmpty()) &&
                  (stack.peekToken().order() >= token. order()))
               postfix.enqueue(stack.popToken());
            stack.push(token);
         }
      }
      while (!stack.isEmpty())
         postfix.enqueue(stack.popToken());
      return postfix;
   }
```

This code, of course, will not compile: first we are not catching a `QueueExcpetion` when we `dequeue` an object from the expression queue, and second our `Token` classes do not have an `order` method to determine the precedence of an operator. And most importantly, our algorithm does not take into account that the expression could contain parenthesis.

So, we will first adjust our algorithm to ensure it handles parenthesis correctly, then provide a complete implementation of the `toPostfix` method (which means adding an `order` method to the appropriate `Token` class). To allow for parenthesis turns out to be easy:

- An open parenthesis is pushed into the operator stack.
- When a closing parenthesis is found all operators up to and including the first open parenthesis are popped from the stack.
- The operators are added to the postfix queue in the order they are popped, while the parentheses are ignored.

Before we can create the complete method we need to adjust our `TokenOperator` class so that it can identify open or closing parenthesis and return the precedence of an operator. We will give the operators "+" and "-" precedence 1, "/" and "*" precedence 2, and "^" gets precedence 3. We also need to make sure that operators are still added to the operator queue if an open parenthesis is on top of the stack. Therefore, the open bracket will get precedence 0. To make sure that a closing parenthesis forces all operators to be popped we give the closing bracket a precedence of 9. Here is our new `TokenOperator` class, with the new fields and method in italics:

```
public class TokenOperator extends Token
{  private static String OPERATORS  = "+-/*^()";
   private static int[]  PRECEDENCE = {1,1,2,2,3,0,9};
   private String token;
   public TokenOperator(String _token) throws TokenException
   {  if (OPERATORS.indexOf(_token) >= 0)
         token = _token;
      else
         throw new TokenException("invalid operator: " + _token);
   }
   public boolean isOpenParen()
   {  return token.equals("(");   }
   public boolean isCloseParen()
   {  return token.equals(")");   }
   public int order()
   {  return PRECEDENCE[OPERATORS.indexOf(token)]; }
   public static boolean isValidOp(char op)
   {  return (OPERATORS.indexOf(op) >= 0); }
   public String toString()
   {  return "Operator: " + token; }
}
```

Note the trick we are using to return the correct precedence for an operator: we find the position of the token in the list of operators and return the integer at *that* position in the array of precedence numbers. That works because the order of the operators in the field `OPERATORS` matches the order of the precedence integers in the field `PRECEDENCE`.

Here is the finished version of the `toPostfix` method, using our new `TokenOperator` class and the above algorithm adjusted to handle parenthesis:

```
protected static Queue toPostfix(Queue infix) throws TokenException
{  Stack stack   = new Stack();
   Queue postfix = new Queue();
   try
   {  while (!infix.isEmpty())
      {  Object token = infix.dequeue();
         if (token instanceof TokenNumeric)
            postfix.enqueue(token);
         else
         {  TokenOperator op = (TokenOperator)token;
            if (op.isOpenParen())
               stack.push(op);
            else if (op.isCloseParen())
            {  while (!((TokenOperator)stack.peek()).isOpenParen())
                  postfix.enqueue(stack.pop());
               Object dummy = stack.pop();
            }
            else if (stack.isEmpty())
               stack.push(op);
            else if (((TokenOperator)stack.peek()).order() < op.order())
               stack.push(op);
            else
            {  while ((!stack.isEmpty()) &&
                     (((TokenOperator)stack.peek()).order() >= op.order()))
                  postfix.enqueue(stack.pop());
               stack.push(op);
            }
         }
      }
      while (!stack.isEmpty())
      {  if (!((TokenOperator)stack.peek()).isOpenParen())
            postfix.enqueue(stack.pop());
         else
            throw new TokenException("unmatched bracket");
      }
      return postfix;
   }
   catch(StackException se)
   {  throw new TokenException("unmatched bracket or invalid input"); }
   catch(QueueException qe)
   {  throw new TokenException("unknown exception"); }
}
```

The method is easy to check: first add this method to the `Calculator` class of the previous example, then change the line in the `main` method of that class:

```
tokens = tokenize(input);
```

to the line

```
tokens =
      toPostfix(tokenize(input));
```

and run it. It will now print out the postfix notation of an expression entered as a string in infix notation. As a side benefit, our `toPostfix` method can also determine if parentheses are matching.

Figure 8.2.2 shows a sample run of our current `Calculator` program.

```
CommandOutput *
C:\Java\08\8.2.3>java Calculator
Calc [q to quit] => 10+20*30
10.0
20.0
30.0
Operator: *
Operator: +
Calc [q to quit] => 10*20+30
10.0
20.0
Operator: *
30.0
Operator: +
Calc [q to quit] => (10+20)*30
10.0
20.0
Operator: +
30.0
Operator: *
Calc [q to quit] => (10+20*30
TokenException: unmatched bracket
Calc [q to quit] => q
```

Figure 8.2.2: Trial run of `Calculator`

■

Stage 3: Postfix Evaluation

Now we are ready to finish our calculator program. We can so far turn a string into an expression of valid tokens and convert these tokens from infix to postfix notation. What is left is a method that evaluates an expression in postfix notation. However, that will be easy, which is of course why postfix notation is so useful.

Example 8.2.4:

Our goal is to create a calculator that can evaluate expressions involving numbers, standard round parenthesis, and the binary operators +, -, *, /, and ^, using the "natural" order of precedence of these operators. In examples 7.2.1 and 7.2.3 we created methods to turn a string representing a mathematical expression into a queue of tokens, and to convert that queue into postfix notation. Create a method `evaluatePostfix` that can evaluate a queue of tokens in postfix notation and use it to finish a complete line-based calculator program.

The algorithm to evaluate a postfix expression where every operator is a binary operator (i.e. requires two numeric inputs such as `+`, `-`, `*`, `/`, and `^`) is simple:

- start with the first token in the postfix queue
  - if token is a number, `push` it into a stack
  - otherwise (token is an operator) do the following:
    - `pop` two numbers from the operator stack if possible
    - evaluate the operator token with the two numbers just popped and `push` the numeric answer back into the stack
  - continue this process with the next token

- when done, there should be exactly one number in the stack: the answer. Otherwise there's been an error in the expression

If the stack was empty when trying to pop two numbers from it, there also must be an error in the original expression.

This algorithm is simple, so we could proceed to code the method right away. However, we have not settled the question of how exactly we are going to evaluate any expression, or - more precisely - who is going to do the evaluation. In fact, the `TokenOperator` knows the valid operators, and therefore it, too, should be responsible for evaluation. That would be beneficial because if we later added another legal binary operator such as the '%' operator, the only class to change would be the `TokenOperator` class. We therefore add an `evaluate` method to `TokenOperator` that takes two numbers (doubles) as input and evaluates these numbers according to the current operator (which is a field of the object):

```
public class TokenOperator extends Token
{  /* fields and methods as before, but add this new method: */
   public TokenNumeric evaluate(double num1, double num2)
          throws TokenException
   {  if (token.equals("+"))
         return new TokenNumeric(num2 + num1);
      else if (token.equals("-"))
         return new TokenNumeric(num2 - num1);
      else if (token.equals("*"))
         return new TokenNumeric(num2 * num1);
      else if (token.equals("/"))
         return new TokenNumeric(num2 / num1);
      else if (token.equals("^"))
         return new TokenNumeric(Math.pow(num2, num1));
      else
         throw new TokenException("invalid token, can not evaluate");
   }
}
```

Now we can implement the `evaluatePostfix` method according to the above algorithm as follows:

```
protected static double evaluatePostfix(Queue postfix) throws TokenException
{  Stack numbers = new Stack();
   try
   {  while (!postfix.isEmpty())
      {  Object token = postfix.dequeue();
         if (token instanceof TokenOperator)
         {  TokenNumeric n1 = (TokenNumeric)numbers.pop();
            TokenNumeric n2 = (TokenNumeric)numbers.pop();
            TokenOperator op = (TokenOperator)token;
            numbers.push(op.evaluate(n1.getNumber(),n2.getNumber()));
         }
         else
            numbers.push(token);
      }
      TokenNumeric answer = (TokenNumeric)numbers.pop();
      if (numbers.isEmpty())
         return answer.getNumber();
      else
         throw new TokenException("invalid expression");
   }
   catch(StackException se)
   {  throw new TokenException("invalid expression"); }
   catch(QueueException qe)
   {  throw new TokenException("unknown error"); }
}
```

We attempt to pop two numbers from the stack without checking whether there really are two tokens available. After all, if the postfix queue does not represent a valid mathematical expression there may not be two numbers in the stack. However, in that case a pop operation would result in a StackException, which we catch and re-throw as an appropriate TokenException. In fact, we re-throw every exception as an appropriate TokenException, which simplifies our code quite a lot. Hence, our method not only evaluates mathematically correct expressions, but also acts as the final check for correctness of our expression.

Now we can use our three methods tokenize, toPostfix, and evaluatePostfix to conclude our Calculator program in its entirety. The class is similar to before, but it does not have to print out any stacks or queues any more.

```
public class Calculator
{  protected static double evaluatePostfix(Queue postfix)
                   throws TokenException
   { /* as above */ }
   protected static Queue toPostfix(Queue infix) throws TokenException
   { /* as before */ }
   protected static Queue tokenize(String input) throws TokenException
   { /* as before */ }
   public static void main(String args[])
   {  String prompt = "Calc [q to quit] => ";
      String input  = "";
      while (!input.equalsIgnoreCase("q"))
      {  if (!input.equals(""))
         {  try
            {  double ans = evaluatePostfix(toPostfix(tokenize(input)));
               System.out.println(prompt + input + " = " + ans);
            }
            catch(TokenException te)
            {  System.err.println(te); }
         }
         System.out.print(prompt);
         input = Console.readString();
      }
   }
}
```

Here are a few calculations with our new Calculator program:

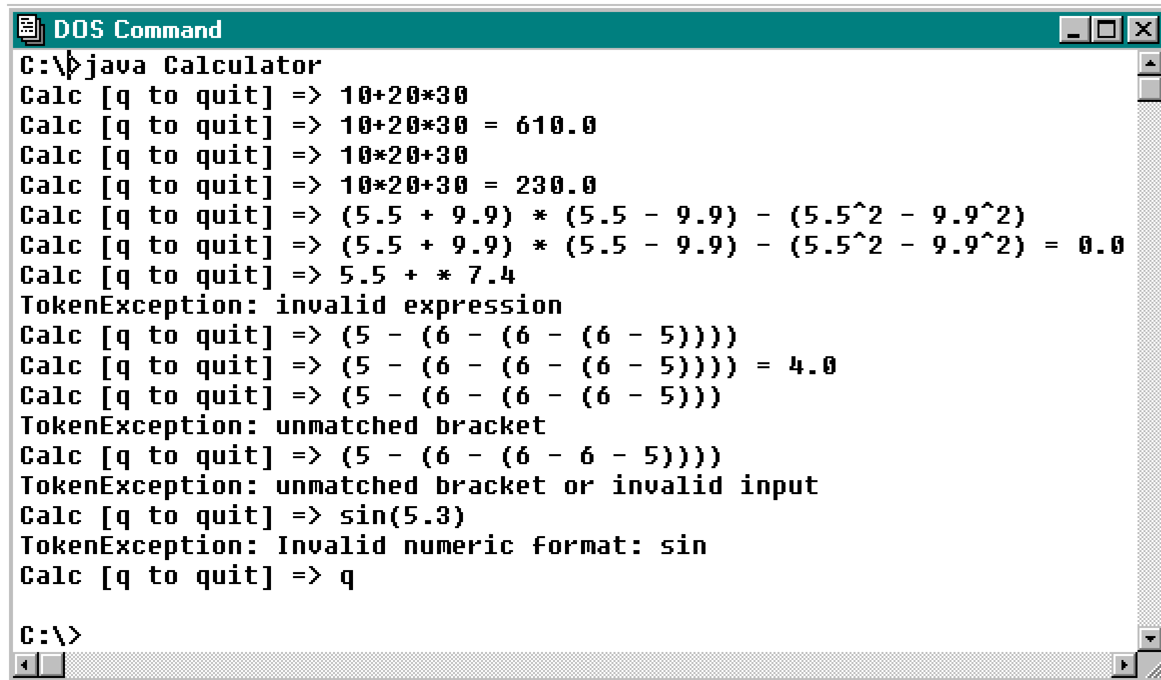


Figure 8.2.3: Output of `Calculator` *Interpreter*

■

Our calculator program, i.e. the three static methods `tokenize`, `toPostfix`, and `evaluatePostfix`, rely on the fact that all legal operators are *binary* operators, i.e. they require two numbers as input. That seems to be a valid assumption at first glance, but one particular operator could be a binary as well as a unitary operator (a unitary operator applies to only one number). Of course it is the minus sign '-':

- used as a binary operator, it subtracts the second number from the first
- used as a unitary operator, it changes the sign of its argument

But our `Calculator` methods *require* binary operators. Therefore, our calculator can not handle perfectly legal mathematical expressions such as `-10 + 10` (but `10 - 10` would work fine, of course). For expressions such as `-10 + 10` our program would produce an exception with the message "`TokenException: invalid expression`".

Fixing this problem is a typical good news, bad news situation:

- The bad news is that virtual all our methods and classes need to be modified to allow for the unitary operator '-' to work correctly, and it is not entirely trivial to do it correctly.
- The good new is that once the single unitary operator '-' works correctly, then it is very easy to make other unitary operators such as `sine`, `cosine`, `tan`, and in fact all standard mathematical functions work correctly.

We will leave further the details as an exercise, however.

Chapter Summary