
Chapter Objectives

After studying this chapter, you should be able to:

- Store data in an array, access a single element, process all elements, search for a particular element, and put the elements in order.
- Declare, allocate, and initialize an array.
- Handle changing numbers of elements in an array, including inserting a new element and deleting an existing one.
- Enlarge or shrink the size of an array.
- Manipulate data stored in a multi-dimensional array.

We often work with lists in our daily lives: grocery lists, to-do lists, lists of books needed for a particular course, the invitation list for our next party, and so on. To be useful, computers must also work with lists: a list of the `Thing` objects in a `City`, a list of concert tickets, or a list of bank accounts, to identify just a few.

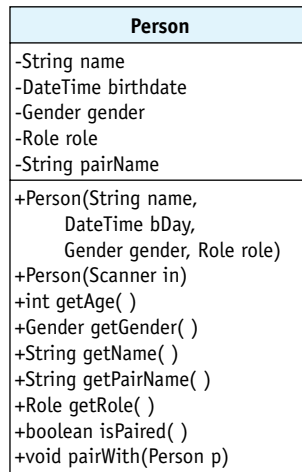
There are several ways to implement lists in Java. One of the most fundamental approaches is with an array, a kind of variable. Once a list is stored in the array we can do many things: tick off the third item in our to-do list, print the entire list of books for a course, search our list of invitations to verify that it includes James Gosling, or sort the list alphabetically.

In Section 8.5, we studied classes in the Java library that are similar to arrays in that they store a collection of objects. Some of these, such as `ArrayList`, are thinly disguised arrays. Others, such as `HashMap`, provide more sophisticated ways to find objects in the collection. But underneath it all, many of these classes use an array.

10.1 Using Arrays

Big Brothers/Big Sisters is a charitable association that matches men and women with boys and girls between the ages of 6 and 16 who could benefit from an older friend and role model. In many cases the boys and girls are missing a parent due to death or divorce and don't have many positive role models in their lives.

Obviously, an association like Big Brothers/Big Sisters keeps lists. One of the most crucial is the list of “bigs” (the adults) and “littles” (the girls and boys) participating in the association. In this chapter we will consider a computer program that maintains a list of `Person` objects (see Figure 10-1) in an array. An **array** is a kind of variable that can store many items, such as the items in a list. We will learn how to print the entire list of people or just the people that meet certain qualifications, such as being a six-year-old girl. We will learn how to search the list for a specific person and learn to find the person that meets a maximum or minimum criterion (such as the oldest or youngest). Of course, all these techniques will apply to lists of other kinds of objects as well.



The simplified version of `Person`, shown in Figure 10-1, uses two enumerations: `Gender` and `Role`. The first enumeration provides the values `MALE` and `FEMALE`; the `Role` enumeration provides the values `BIG` to represent an adult participant and `LITTLE` to represent a young person. The `pairWith` command will pair this person with the person, `p`, specified as a parameter. It does this by setting the `pairName` appropriately in both objects.

Throughout this section, we will assume that we have an array named `persons` containing a list of `Person` objects. In Section 10.2, we will learn how to create such a variable and fill it with data.

KEY IDEA

See www.bbbsc.ca or www.bbbsa.org for more information on Big Brothers/Big Sisters.

KEY IDEA

There are many algorithms that work with lists of things.

(figure 10-1)

Class diagram for `Person`

LOOKING BACK

Enumerations are new in Java 1.5 and are discussed in Section 7.3.4.

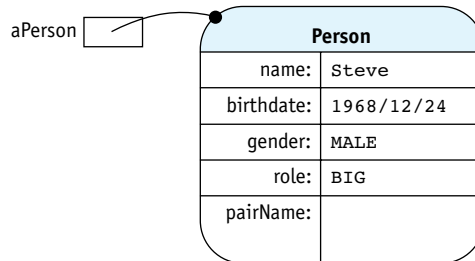
10.1.1 Visualizing an Array

LOOKING BACK

Object diagrams were first discussed in Chapter 1. References were discussed in Section 8.2.

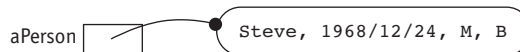
(figure 10-2)

Object diagram showing a variable referring to a Person object



(figure 10-3)

Abbreviated object diagram

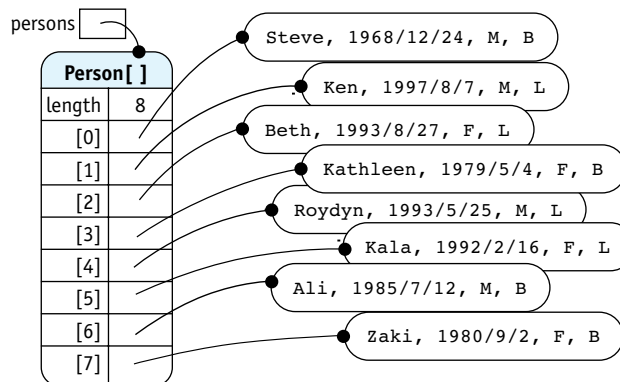


In both diagrams, the box labeled `aPerson` is a variable that refers to an object—the round-cornered box labeled `Person`.

So, what does an array look like? Figure 10-4 shows a visualization of an array of `Person` objects. The reference variable `persons` refers to an array object. The array object refers to many `Person` objects. Each reference, called an **element** of the array, is numbered beginning with zero. This number is called the **index**.

(figure 10-4)

Visualizing an array of Person objects



Notice that an array is illustrated almost exactly like other kinds of objects. Similarities include a variable, such as `persons`, that refers to the array object just as the variable `karel` referred to a `Robot` object in earlier chapters. An array object contains a public final instance variable named `length`, but has no methods. `length` stores the number of elements in the array.

The crucial difference between arrays and objects is that the array has instance variables that are accessed with square brackets and a number instead of a name. This is illustrated in Figure 10-4 with variables named `[0]`, `[1]`, and so on. The numbering always starts at zero. This language rule often causes beginning programmers grief because most people naturally begin numbering with one. Furthermore, the indices run from zero to one less than the number stored in `length`. For example, in Figure 10-4, `length` is 8 but the indices run from 0 to 7.

The fact that the elements in the array are numbered gives them an order. It makes sense to speak of the first element (the element numbered 0), the second element, and the last element.

10.1.2 Accessing One Array Element

Accessing a specific element in an array is as easy as accessing a normal variable—except that the index of the desired element must also be specified. If we had a simple variable named `aPerson` we could print the name with the following line of code:

```
System.out.println(aPerson.getName());
```

Printing the name of the first person in our array is almost as easy. Instead of only naming the variable, we name the array and the position of the element we want:

```
System.out.println(persons[0].getName());
```

The index of the desired element is given by appending square brackets to the name of the array. The index appears between the brackets. You may use the result in exactly the same ways that you use a variable of the same type.

Here is another code fragment that shows the `persons` array in use. In each case, `persons` is followed by the index of a specific element in the array.

```
1 // Check if Kathleen (see Figure 10-4) is a "Big"
2 if (persons[3].getRole() == Role.BIG)
3 { System.out.println(persons[3].getName() + " is a Big.");
4 }
```

KEY IDEA

Elements in an array are numbered beginning with zero.

KEY IDEA

Each element has an index giving its position in the array.

KEY IDEA

Arrays are indexed with square brackets and an integer expression.

KEY IDEA

An element in an array can be assigned to another variable.

It is also possible to assign a reference from the array to a regular variable. For example, the previous code fragment could have been written like this:

```

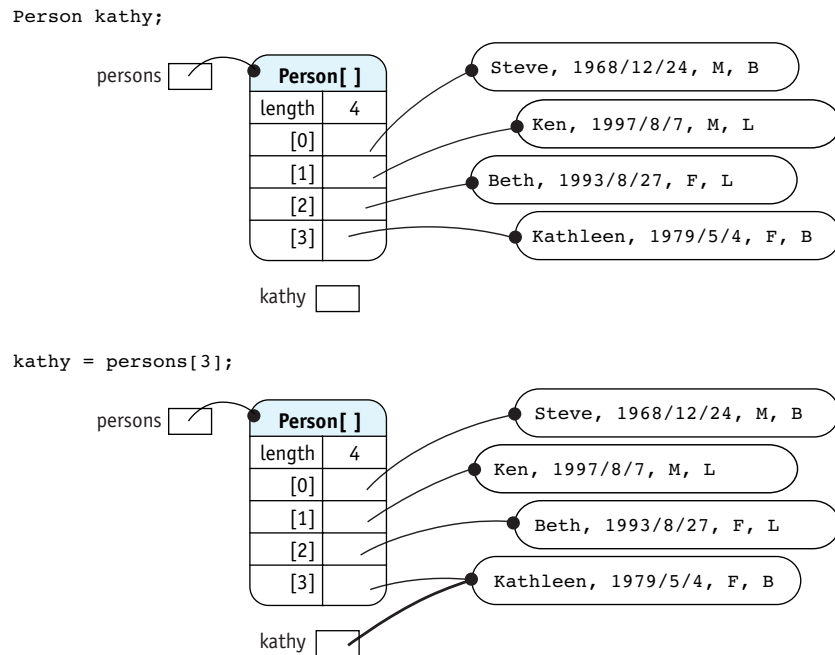
1 Person kathy;
2 // Check if Kathleen (see Figure 10-4) is a "Big"
3 kathy = persons[3];
4 if (kathy.getRole() == Role.BIG)
5 { System.out.println(kathy.getName() + " is a Big.");
6 }

```

The effect of the reference assignment in line 3 is just like assigning references between non-array variables and is traced in Figure 10-5. Assigning a reference from an array to an appropriately named temporary variable can make code much more understandable.

(figure 10-5)

Tracing a reference assignment using an array and a non-array variable



References stored in an array may also be passed as arguments. For example, Kathleen and Beth could be paired as Big and Little Sisters with the following sequence of statements:

```

// Pair Kathleen and Beth
Person kathy = persons[3];
Person beth = persons[2];
kathy.pairWith(beth);

```

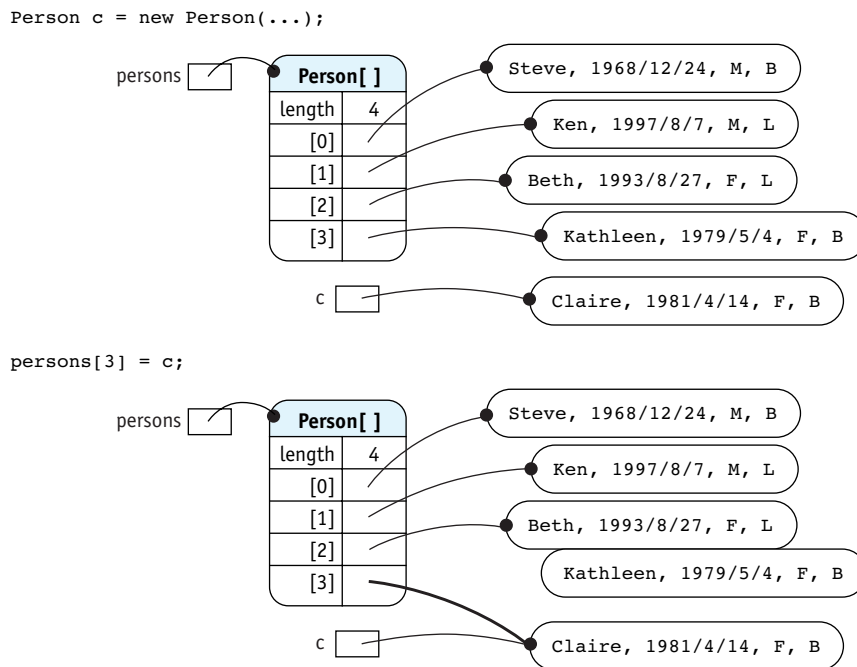
However, because elements of an array can be used just like a regular variable, we could also pair Kathleen and Beth this way:

```
// Pair Kathleen and Beth
persons[3].pairWith(persons[2]);
```

Finally, we can also assign a reference to an array element. For example, suppose Kathleen is replaced by her friend Claire. The following code constructs an object to represent Claire and then replaces the reference to Kathleen's object with a reference to Claire's object.

```
Person c = new Person("Claire", new DateTime(1981,4,14),
                      Gender.FEMALE, Role.BIG);
persons[3] = c;
```

This code fragment is traced in Figure 10-6.



(figure 10-6)

Tracing the assignment of a reference into an array element


The object modeling Kathleen will be garbage collected unless another variable is referencing it.

LOOKING BACK

When an object has no references to it, the resources it uses are recycled. See Section 8.2.3.

10.1.3 Swapping Array Elements

We can easily exchange, or swap, two elements in an array. For example, suppose we wanted to switch the places of Ken and Beth within the array. A temporary variable is needed to store a reference to one of the elements while the swap is taking place. A method to perform a swap follows. It takes two arguments, the indices of the two elements to swap. Note that we are now assuming that `persons` is an instance variable.

FIND THE CODE 
ch10/bbbs/

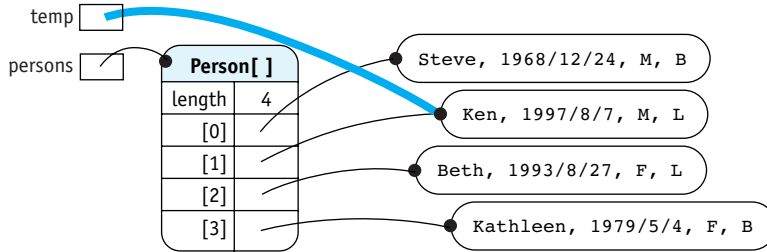
```
class BigBroBigSis extends Object
{ ... persons ...

    /** Swap the person object at index a with the object at index b. */
    public void swap(int a, int b)
    { Person temp = this.persons[a];
      this.persons[a] = this.persons[b];
      this.persons[b] = temp;
    }
}
```

After the `swap` method finishes executing, the temporary variable `temp` will cease to exist. The object it referenced, however, is still referenced by one element in the array and will not be garbage collected.

Figure 10-7 traces the execution of `swap(1, 2)`.

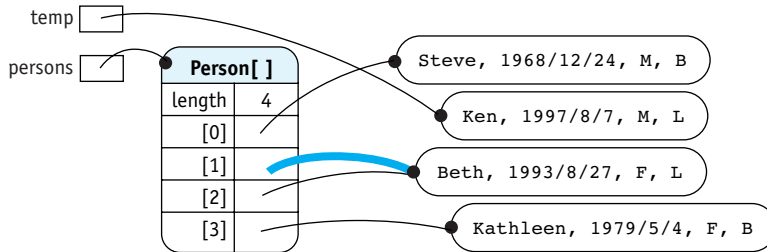
```
{ Person temp = this.persons[a];
```



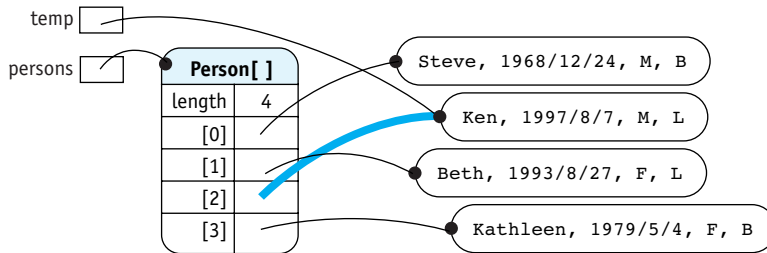
(figure 10-7)

Tracing `swap(1, 2)`; the parameter `a` has the value 1 and `b` has the value 2

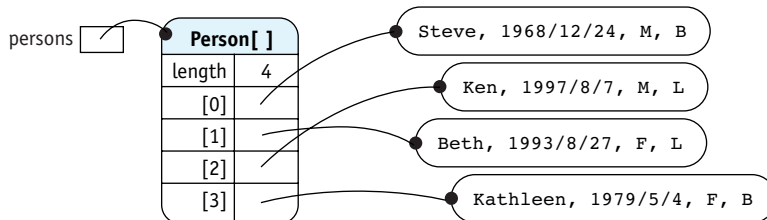
```
this.persons[a] = this.persons[b];
```



```
this.persons[b] = temp;
```



```
// After the swap method finishes
```



10.1.4 Processing All the Elements in an Array

Accessing an element of an array using a number may not seem particularly helpful. We could, after all, simply declare many variables that just have a number in each name:

```
Person person0;
Person person1;
Person person2;
```

But consider printing the name of each person in the list. Without an array, we would need statements for each named variable:

```
System.out.println(person0.getName());
System.out.println(person1.getName());
System.out.println(person2.getName());
...
```

If the list contained 1,000 people, the method to print their names would have about 1,000 lines. What a pain!

KEY IDEA

Arrays may be indexed with variables.



PATTERN

Process All Elements

Fortunately, an array's index may be a variable—or any other expression that evaluates to an integer. This is where the power of arrays really becomes apparent. By putting the `println` statement inside a loop that increments a variable index, we can print the entire array with only three lines of code—no matter how many elements are in it.

```
// Print the the name of every person in the array.
for (int i = 0; i < this.persons.length; i++)
{ System.out.println(this.persons[i].getName());
}
```

KEY IDEA

The number of elements in an array can be found with `.length`.

KEY IDEA

The last index is one less than the length of the array.

One item of note in this code fragment is the test in the `for` loop. The length of an array can always be found with the array's public final instance variable, `length`. If the array is as illustrated in Figure 10-4, `this.persons.length` will return 8, the number of elements in the array. The index, `i`, takes values starting with 0 and ending with 7, one less than the array's length. The length of the array is 8 but the index of the last element is one less, 7. This is surely one of the most confusing aspects of arrays for beginning programmers.

So far we have encountered three different mechanisms to find the number of elements in a collection. Arrays use the public instance variable, `length`. The number of characters in a string is found with a method, `length()`. Finally, Java's collection classes such as `ArrayList` and `HashMap` also use a method to find the number of elements, but it has a different name, `size()`.

Another task that uses a loop to access each element in turn is to calculate the average age of the people in the array. For this task, we will use a variable to accumulate the ages while we loop through the array. After we have added all the ages, we'll divide by the length of the array to find the average age.

```
/** Calculate the average age of persons in the array. */
public double calcAverageAge()
{ int sumAges = 0;
  for (int i = 0; i < this.persons.length; i++)
  { Person p = this.persons[i];
    sumAges = sumAges + p.getAge();
  }
  return (double)sumAges/this.persons.length;
}
```



ch10/bbbs/

The variable `sumAges` has the role of a gatherer: It gathers all the individual ages together. That value is then used to find the average age.

The loop controlling the index, `i`, is exactly the same in `calcAverageAge` as it was in the example to print all the names. This looping idiom—starting the index at 0 and incrementing by one as long as it is less than the length of the array—is extremely common when using arrays. Using it should become an automatic response for every programmer confronted with processing all the elements in an array.

Using the *foreach* Loop

You may remember that processing each element was also a common activity when using the collection classes, such as `ArrayList` and `HashSet`. In that situation, we used the *foreach* loop introduced with Java 1.5. The *foreach* loop also works with arrays. The following loop is equivalent to the one used in `calcAverageAge`, shown earlier.

```
for (Person p : this.persons)
{ sumAges = sumAges + p.getAge();
}
```



Process All Elements

The *foreach* loop is a generalized loop designed for use with unordered data structures such as maps and trees, for which asking for element *n* makes no sense. Hence, a *foreach* loop has no index. Instead, one element from the collection is provided for each iteration of the loop until all of the elements have been processed.

Programmers should be familiar with both looping styles. To emphasize this, we'll alternate between the two.

10.1.5 Processing Matching Elements

The method just written, `calcAverageAge()`, does not seem nearly as useful as a method to find the average age of only the littles or only the bigs. In the previous example, we added the age of every element in the array. To find the average age of only the

littles, we want to include the ages only if the person is, in fact, a little. This logic is shown in the following pseudocode:



Process Matching
Elements

```
for each person in the array
{ if (the person is a little)
  { include this person in the average
  }
}
return average
```

By adding the `if` statement inside the loop, we restrict its effects to only those elements that match the test. We process the matching elements. Notice that this pattern is very similar to the Process All Elements pattern.

This pseudocode translates to Java as follows:

FIND THE CODE ↓
ch10/bbbs/

```
/** Find the average age of the "littles". */
public double getAverageLittleAge()
{ int sumAges = 0;
  int numLittles = 0;
  for (Person p : this.persons)
  { if (p.getRole() == Role.LITTLE)
    { sumAges = sumAges + p.getAge();
      numLittles = numLittles + 1;
    }
  }
  return (double) sumAges/numLittles;
}
```

LOOKING AHEAD

We'll learn how to generalize these methods with interfaces and polymorphism in Chapter 12.

Of course, by changing the test in the `if` statement, we change which objects we process. By changing the body of the `if` statement, we change how they are processed. For example, the following code fragment prints all the “bigs” who have not been paired with a “little.”

```
// Print the names of unpaired "bigs"
for (int i = 0; i < this.persons.length; i++)
{ Person p = this.persons[i];
  if (p.getRole() == Person.BIG && !p.isPaired())
  { System.out.println(p.getName());
  }
}
```

10.1.6 Searching for a Specified Element

In one of our first examples we paired Beth, the person at index 2, with Kathleen, the person at index 3. But when we've decided to pair Beth and Kathleen, how do we find their positions in the array? We search for them.

Searching involves using some identifying information—such as a name, telephone number, or government identification number—and finding the corresponding object in the array. The identifying information is often called a **key**. If each key is unique, then at most one object in the array will match the key. Government identification numbers usually identify a unique person. On the other hand, names and telephone numbers may match several different people. In that case, a search generally returns the first object that matches.

In most cases we don't know that our search will be successful. It might be that no object matches the key. Therefore, we need a way to indicate failure. This is usually done by returning a special value such as `null` or `-1`. We can use `null` when the search method returns the object that was found and `-1` when the search method returns the array index where the object was found. We use `null` and `-1` for this role because `null` is never a legal reference to an object and `-1` is never a legal array index.

The easiest way to write a search method is a variation of the Process Matching Elements pattern—except that the “processing” is to exit the loop and return the answer. Suppose we are looking for a person using their name as a key. The logic is shown in the following pseudocode:

```
for each person in the array
{ if (the person's name matches the key)
  { exit the loop and return the person
  }
}
return null
```

We can exit the loop when we find the right person with the `return` statement. If we examine all of the people in the array and do not find one matching the key, the code will exit the loop at the bottom and return `null`, indicating the search failed.

In Java, this can be implemented as the method shown in Listing 10-1.

Listing 10-1: Searching an array

```
1 /** Search for the first person object matching the given name.
2  * @param name The name of the person to find (the key).
3  * @return The first matching person object; null if there is none. */
4 public Person search(String name)
5 { for (int i = 0; i < this.persons.length; i++)
6   { Person p = this.persons[i];
7     if (p.getName().equalsIgnoreCase(name))
8       { return p; // Success. Exit the loop and return the person found.
9     }
10  }
11  return null; // Failure.
12 }
```

PATTERN 
Linear Search

 **FIND THE CODE**
[ch10/bbbs/](#)

PATTERN 
Linear Search

The search method can also be written without the temporary variable `p`, as follows:

```
public Person search(String name)
{ for (int i = 0; i < this.persons.length; i++)
  { if (this.persons[i].getName().equalsIgnoreCase(name))
    { return this.persons[i];          // Search succeeded.
    }
  }
  return null;                          // Search failed.
}
```

LOOKING BACK

The `Prompt` class was discussed in Section 9.4.2.

We can use the search method to pair Kathleen and Beth as follows:

```
String bigName = Prompt.forString("Big's Name:");
Person big = this.search(bigName);
String littleName = Prompt.forString("Little's Name:");
Person little = this.search(littleName);
big.pairWith(little);          // Dangerous code!
```

KEY IDEA

Always confirm a search was successful before proceeding.

The last line is marked as dangerous code because one or both of the searches may have failed, in which case `big` or `little` will contain the value `null`. Then a `NullPointerException` will be generated when the last line executes. The outcome of a search should *always* be verified and failure handled. The following is better code because it checks that the searches were successful.

```
String bigName = Prompt.forString("Big's Name:");
Person big = this.search(bigName);
while (big == null)
{ System.out.println(bigName + " not found.");
  bigName = Prompt.forString("Big's Name:");
  big = this.search(bigName);
}

// Repeat the above to find the little.

big.pairWith(little); // Safe because both big and little have been found.
```

Another Approach to Searching

Many people think it is a bad idea to exit a loop early. They think that a line such as the following is like a contract between the programmer and the reader.

```
for (int i = 0; i < this.persons.length; i++)
```

The contract says this code will execute one time for every person in the array. Returning from the middle of the loop, like the search in Listing 10-1, breaks the contract.

A search algorithm that respects this view uses a `while` loop, which does not imply that every element in the array will be visited. The core idea is to repeatedly increment an index variable so that elements of the array are examined in turn. This is Step 1 of the Four-Step Process for constructing a `while` loop. The loop stops (Step 2) when either the end of the array is reached or the desired element is found, whichever comes first. Therefore, the loop continues as long as we have *not* reached the end of the array and we have not found the desired element. The loop is assembled (Step 3) with the results of Steps 1 and 2. Finally, after the loop (Step 4), we need to determine the answer and return it.

The logic is shown in the following pseudocode:

```
while (not at the end of the array and matching object not found)
{ increment index to examine the next object
}
if (at the end of the array)
{ the search failed; return null
} else
{ the search succeeded; return the object
}
```

Making this pseudocode concrete to search for a person results in Listing 10-2.

Listing 10-2: Another approach to searching an array

```
1 /** Search for the first person object matching the given name.
2  * @param name The name of the person to find (the key). */
3 public Person searchAlt(String name)
4 { int i = 0;
5   while (i < this.persons.length &&
6         !this.persons[i].getName().equalsIgnoreCase(name))
7     { i++;
8     }
9
10  if (i == this.persons.length)
11  { return null;           // Failure: got to the end without finding it.
12  } else
13  { return this.persons[i]; // Success.
14  }
15 }
```

LOOKING BACK

The Four-Step Process for constructing a loop is discussed in Section 5.1.2.



PATTERN

Linear Search



[ch10/bbbs/](#)

10.1.7 Finding an Extreme Element

An extreme element has the most of something or the least of something. It might be the person with the most age (oldest person) or the least age (youngest person). In other contexts, extreme elements might be the employee with the highest salary, the robot with the most things, the stock with the highest price/earnings ratio, or the name appearing first in dictionary ordering.

The strategy is to step through the array using the Process All Elements pattern. As we go, we'll remember the element that best meets the criteria so far. For each new element we examine, we'll ask if it meets the criteria better than the one we're remembering. If it does, remember it instead. Expressed in pseudocode, this algorithm is:



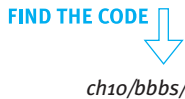
```

remember the first element as the best seen so far
for each remaining element in the array
{ if (the current element is better than the best seen so far)
  { remember the current element as the best seen so far
  }
}
return the best seen so far

```

Listing 10-3 applies this algorithm to the problem of finding the oldest person in the array. It begins, in line 3, by remembering the first person in the array (at index 0) as the oldest we've seen so far. This must be true, because we haven't looked at anyone else.

In line 5, we start looking at the rest of the people in the array. Lines 6–8 check if the current person matches the criteria better than `oldestSoFar`. If it does, the old value of `oldestSoFar` is replaced with `currentPerson`. When the loop ends, `oldestSoFar`



will contain the oldest person in the entire list.

Listing 10-3: An example of finding an extreme element: the oldest person in the array

```

1  /** Find oldest person in the list. (Assumes there is at least one person in the array.) */
2  public Person findOldestPerson()
3  { Person oldestSoFar = this.persons[0];
4
5    for (Person currentPerson : this.persons)
6    { if (currentPerson.getAge() > oldestSoFar.getAge())
7      { oldestSoFar = currentPerson;
8      }
9    }
10 return oldestSoFar;

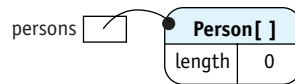
```

11 }

What happens if two elements in the array meet the criteria equally well? What if two people have the same age? The algorithm given here will return the first one found and ignore anyone occurring later in the array who happens to be the same age. Changing the `>` in line 6 to `>=` results in finding the oldest person who appears last.

Listing 10-3 returns the extreme element. Sometimes it is desirable to return the index of that element instead. Implementing such a method requires replacing the `foreach` loop with a regular `for` loop which makes the index explicit.

Java allows an empty array (an array with length zero), as shown in Figure 10-8.



(figure 10-8)

Empty array

The code in Listing 10-3 will fail on such an array with an `ArrayIndexOutOfBoundsException` at line 3. Programmers should always be aware of such a possibility and decide how to handle it. Options include the following:

- Document that calling the method with an empty array is an error. Check for that situation and throw an exception, if required.
- Document the value the method will return if the array is empty. This would typically be `null` if the method returns the extreme element and `-1` if it returns the index of the extreme element. Of course, a check must be made for empty arrays so the correct value can be returned.

10.1.8 Sorting an Array

Collections of things are often easier to work with if they are sorted. Card players usually sort the collection of playing cards in their hands. A collection of words in a dictionary is usually sorted in alphabetical order, as are names in a telephone book. A collection of banking transactions are sorted by date on the bank statement.

Different algorithms can sort an array. Many of these algorithms have been given names: Insertion Sort, Selection Sort, QuickSort, HeapSort, ShellSort, MergeSort, and so on. Selection Sort is one of the easiest sorting algorithms to master. It builds on three patterns we have already seen: Process All Elements, Find an Extreme, and Swap Two Elements.

These sorting algorithms vary widely in their efficiency and in their ease of implemen-

tation. Insertion Sort and Selection Sort are easy to implement but slow to execute. QuickSort, HeapSort, ShellSort, and MergeSort are all much, much faster for large arrays but are more difficult to implement. They are typically included in a second year Computer Science course.

Understanding Selection Sort

Diagrams help us understand how a sort works. For simplicity, our diagrams will use an array of letters; when the array is sorted, the letters will be in alphabetical order.

The core idea of Selection Sort is to divide the array into two parts, as shown in Figure 10-9: the part that is already sorted (shown with a dark background) and the part that isn't (shown with a white background).

(figure 10-9)

Dividing an array into two parts

0	1	2	3	4	5	6
A	B	C	G	E	D	F

At each step in the algorithm, we extend the sorted portion of the array by one element. The next element to add to the sorted portion is the smallest element in the unsorted portion of the array, D. It goes in the position currently occupied by G. These two elements are highlighted in Figure 10-10.

(figure 10-10)

Extending the array

0	1	2	3	4	5	6
A	B	C	G	E	D	F

(figure 10-11)

The last part of this step is to swap these two elements, thus extending the sorted portion of the array by one element. See Figure 10-11.

Swapping the two elements; extending the sorted part of the array

0	1	2	3	4	5	6
A	B	C	D	E	G	F

These two actions—finding the element that belongs in the next position and swapping it with the one already there—are performed repeatedly until the entire array is sorted. The algorithm begins with the sorted portion of the array being empty and the unsorted portion consuming the entire array. Figure 10-12 shows the entire sorting operation on a small array.

	0	1	2	3	4	5	6
The initial, unsorted array.	F	E	A	G	B	D	C
Find the element that belongs at index 0.	F	E	A	G	B	D	C
Swap elements at 0 and 2, extending sorted part.	A	E	F	G	B	D	C
Find the element that belongs at index 1.	A	E	F	G	B	D	C
Swap elements at 1 and 4, extending sorted part.	A	B	F	G	E	D	C
Find the element that belongs at index 2.	A	B	F	G	E	D	C
Swap elements at 2 and 6, extending sorted part.	A	B	C	G	E	D	F
Find the element that belongs at index 3.	A	B	C	G	E	D	F
Swap elements at 3 and 5, extending sorted part.	A	B	C	D	E	G	F
Find the element that belongs at index 4.	A	B	C	D	E	G	F
Swap elements at 4 and 4, extending sorted part.	A	B	C	D	E	G	F
Find the element that belongs at index 5.	A	B	C	D	E	G	F
Swap elements at 5 and 6, extending sorted part.	A	B	C	D	E	F	G

(figure 10-12)

Sorting an array of letters into alphabetical order

Two points in this example are worth elaboration. First, notice that when the element in the next to last position (index 5) is swapped into position, the last element (index 6) is automatically placed correctly as well. A moment's thought will explain why: When all the elements but the last are in their correct places, the last one must also be in its correct place because there is no where else for it to be.

Second, when it was time to look for the element to place at index 4, the element just happened to already be there. In this case, we would not need to perform the swapping step. We will anyway, however, because the “cure” of testing for this condition for every position in the array is worse than the “disease” of performing the swap every once in a while.

Coding Selection Sort

Based on this example, we see that two actions are repeated: Find the element that belongs in the next position and swap it with the one already there. These actions are performed for each position in the array, in ascending order, except for the last one. These observations yield the following pseudocode:

```

for each position in the array except the last
{ find the element that should go in this position
  swap that element with the element currently there
}

```

In this case the *foreach* loop is inappropriate because we will *not* be examining every element in the array and because we need the index of the current element.

We can use this algorithm to sort our list of persons, but first we need to decide on the order we want. Sorted by age? Sorted by name in alphabetical order? Something else?



In the first example, we will sort the array by name. To do so, we'll use the `compareTo` method in the `String` class. If we have two `String` variables, `s1` and `s2`, then `s1.compareTo(s2)` returns 0 if the two strings are equal, a negative number if `s1` comes before `s2` in dictionary order, and a positive number if `s1` comes after `s2`.

Listing 10-4 shows the Selection Sort algorithm coded in Java. Let's look briefly at the patterns it uses.

First, the `sort` method uses a very slight variation of the Process All Elements pattern. The difference is that it processes all the elements except the last one. As noted earlier, by the time all the other elements are in their place, the last one must be in its place as well.

Second, the helper method uses a variation of the Find an Extreme pattern. It differs from the pattern in Section 10.1.7 in two ways:

- ▶ It finds the extreme in only the unsorted part of the array. We pass the index of the first element it should consider as an argument.
- ▶ We are concerned with the position of the extreme element, not the element itself. So our most-wanted holder variable in `findExtreme`, `indexBestSoFar`, stores the index of the best `Person` object seen so far rather than a reference to the object.

Third, the `swap` helper method is exactly as we saw before.

FIND THE CODE ↓

ch10/bbbs/

 PATTERN
Selection Sort

Listing 10-4: *Implementing Selection Sort to sort an array of `Person` objects by name*

```

1 public class BBBS extends Object
2 { ... persons ...           // an array of Person objects
3
4 /** Sort the list of persons in alphabetical order by name. */
5 public void sort()
6 { for (int firstUnsorted = 0;
7       firstUnsorted < this.persons.length-1;
8       firstUnsorted++)
9     { int extremeIndex = this.findExtreme(firstUnsorted);
10      this.swap(firstUnsorted, extremeIndex);
11    }
12 }
13
14 /** Find the extreme element in the unsorted portion of the array.
15  * @param indexToStart The smallest index in the unsorted portion of the array.
16  * @return The index of the extreme element. */
17 private int findExtreme(int indexToStart)
18 { int indexBestSoFar = indexToStart;
19   String nameBestSoFar =
20     this.persons[indexBestSoFar].getName();

```

Listing 10-4: *Implementing Selection Sort to sort an array of Person objects by name* (continued)

```

21     for (int i=indexToStart+1; i<this.persons.length; i++)
22     { String currPersonName = this.persons[i].getName();
23       if (currPersonName.compareTo(nameBestSoFar) < 0)
24       { indexBestSoFar = i;
25         nameBestSoFar = this.persons[i].getName();
26       }
27     }
28     return indexBestSoFar;
29 }
30
31 /** Swap the elements at indices a and b. */
32 private void swap(int a, int b)
33 { Person temp = this.persons[a];
34   this.persons[a] = this.persons[b];
35   this.persons[b] = temp;
36 }
37 }

```

Sorting without Helper Methods (optional)

Sorting is performed so frequently that a great deal of effort has been spent to make the operation as fast as possible. The greatest gains in efficiency have been made by employing different algorithms. QuickSort and HeapSort are among the best, but are beyond the scope of this book.

Selection Sort can be made faster by eliminating the helper methods. Normally, eliminating helper methods just to speed up an algorithm is *not* a good idea. In this case, however, it may be justified because the algorithm is still relatively understandable. Listing 10-5 implements `sortByAge` as a single method. The age comparison is somewhat simpler than comparing names and so some temporary variables have been eliminated as well.

Listing 10-5: *Implementing Selection Sort in a single method to sort an array of Person objects by age*

```

1 public class BigBroBigSis extends Object
2 { ... persons ...           // An array of Person objects.
3
4   /** Sort the persons array in increasing order by age. */
5   public void sortByAge()
6   { for (int firstUnsorted=0;

```

LOOKING AHEAD

This code will be made more flexible and reusable in Listing 12.18 in Section 12.5.



[ch10/bbbs/](#)



Listing 10-5: *Implementing Selection Sort in a single method to sort an array of Person objects by age (continued)*

```

7         firstUnsorted<this.persons.length-1;
8         firstUnsorted++)
9     { // Find the index of the youngest unsorted person.
10        int extremeIndex = firstUnsorted;
11        for (int i = firstUnsorted + 1;
12             i < this.persons.length; i++)
13            { if (this.persons[i].getAge() <
14                this.persons[extremeIndex].getAge())
15                { extremeIndex = i;
16                  }
17            }
18
19        // Swap the youngest unsorted person with the person at firstUnsorted.
20        Person temp = this.persons[extremeIndex];
21        this.persons[extremeIndex] =
22                this.persons[firstUnsorted];
23        this.persons[firstUnsorted] = temp;
24    }
25 }
26 }

```

Sorting with the Java Library

Sorting an array is a very common activity and so it's natural that the Java library provides support for it via the `java.util.Arrays` class. It provides methods to sort arrays of all of the primitive types as well as arrays of objects.

The ordering of the primitive types is defined naturally by their values. Not so with arrays of objects. When sorting an array of `Person` objects, for example, how does the library sort know whether to sort by age or name or some other criteria?

The library sorts use two different approaches, both of which are explained in Chapter 12. One approach depends on the objects being sorted implementing the `Comparable` interface. This interface specifies a single method, `compareTo`, that compares two objects and returns a number indicating which should come first. Classes that implement this interface include `String`, `DateTime`, `File`, and enumerated types such as `Direction`. Sorting a list of strings, for example, can be accomplished with the code in Listing 10-6.

The vast majority of the code, lines 11–19 and 25–28, is concerned with reading the strings from the user and printing out the sorted list. The actual sorting is accomplished by a single line of code calling a method in the Java library (line 22).

Listing 10-6: *Sorting strings read from the console*

```
1 import java.util.Arrays;
2 import java.util.Scanner;
3
4 /** Sort the strings read from a file.
5  *
6  * @author Byron Weber Becker */
7 public class Sort
8 {
9     public static void main(String[] args)
10    { // Get the strings from the user.
11        Scanner in = new Scanner(System.in);
12        System.out.print("How many strings:");
13        int num = in.nextInt();
14        in.nextLine();
15
16        String[] strings = new String[num];
17        for (int i = 0; i < num; i++)
18        { strings[i] = in.nextLine();
19        }
20
21        // Sort the strings.
22        Arrays.sort(strings);
23
24        // Display the sorted list of strings.
25        System.out.println("The sorted strings:");
26        for (int i = 0; i < strings.length; i++)
27        { System.out.println(strings[i]);
28        }
29    }
30 }
```

[ch10/librarySort/](#)

The second approach to ordering objects is to pass the sort method the list to sort and an object implementing the `Comparator` interface. This is the most flexible approach and is discussed in Chapter 12.

10.1.9 Comparing Arrays and Files

Some beginning programmers have a hard time distinguishing an array from a file. After all, both store an ordered collection of objects. Both often use algorithms that process all of the objects in the collection.

So what's the difference? The core difference is that a file stores the objects on a disk drive or a related device. An array is stored in the computer's memory.

One consequence is that accessing an array is much faster than accessing a file. The disk drive holding your file has moving parts; waiting for them to move makes accessing a file slow. Memory, on the other hand, stores the array by arranging electrons in its chips. Manipulating electrons is *much* faster.

Files are linear structures. When a file is stored on the disk, all the information is placed into one long line. It's processed by reading the first item of information from the line, then the second, and so on. It's possible to read an item from the middle of the line, but you have to know exactly where to start in considerable detail. You need to know not just that you want the 132nd item, but the exact length of the 131 items that come before it.

Arrays, on the other hand, support **random access** naturally. If you want the 132nd item, use 131 as the index into the array (because arrays are indexed starting at 0). Random access makes sorting an array easy but sorting a file difficult.

So why do we use files at all? Why not store everything in an array? Because storing information on a disk drive is *much* cheaper and because disk drives retain the information even when the power is off; memory does not.

Arrays and files are complementary. We often store information in files while we aren't working on it. When we begin to use the information, we use a program that loads the information from the file into an array. After we're done, usually as one of the last things a program does, the information is written from the array back to the disk where it waits until the next time we use it.

10.2 Creating an Array

So far we have assumed that the `BBBS` class contains an instance variable that is an array of `Person` objects. In this section, we'll see how to create such an array.

KEY IDEA

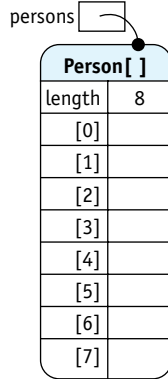
Creating an array has three steps: declaration, allocation, and initialization.

Briefly, creating an array has three steps: declaring the variable, allocating the memory, and initializing each element in the array to a desired value. In some ways, creating an array is like hosting a dinner party. The declaration states your intent to have an array—like sending out invitations to your dinner party. When you allocate memory you decide how many elements your array will have—like counting up the responses to your invitation and setting that many dinner places at the table. Finally, initialization puts a value in each element of the array—like seating one of your guests at each place around your table. These three steps are illustrated in Figure 10-13.

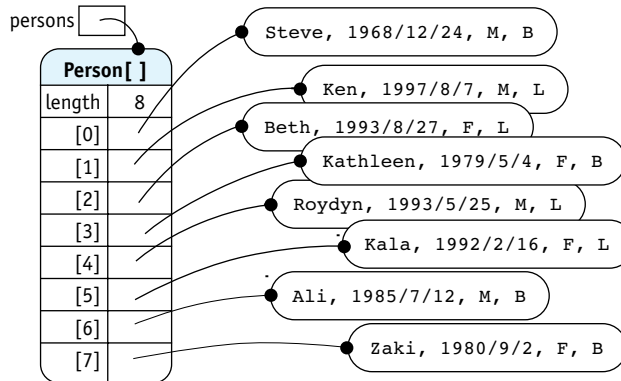
Step 1: Declare the array

persons

Step 2: Allocate space for the references



Step 3: Initialize each element of the array



(figure 10-13)

Three steps in preparing an array for use

10.2.1 Declaration

Declaring an array is like declaring any other reference variable. A type such as `Person` or `Robot` is required, followed by the name of the variable. If the array is an instance variable, then an access modifier such as `private` is appropriate.

The only trick is knowing the type. The type for an array of `Person` objects is `Person []` and the type for an array of `Robot` objects is `Robot []`. Simply add a set of square brackets after the type of elements the array will hold. You might think of the brackets as making the type plural. A variable of type `Person` holds one person. A variable of type `Person []` holds many persons.

KEY IDEA

The type of an array is the same as the type of each element, but with [] appended.

With this background, we can replace the following code:

```
public class BBBS extends Object
{ ... persons ... // An array of Person objects.
```

shown in the listings in Section 10.1 with the complete declaration:

```
public class BBBS extends Object
{ private Person[] persons; // An array of Person objects.
```

The persons array can only hold Person objects.

LOOKING AHEAD

In Chapter 12, we will see that the persons array can also hold subclasses of Person.

10.2.2 Allocation

The declaration of an array does not create the array, but only a place to hold a reference to an array. See Step 1 in Figure 10-13. We also need to allocate the array object itself, similar to constructing any other kind of object. See Step 2 in Figure 10-13.

KEY IDEA

Use the new keyword to set aside space for a specific number of elements.

The following code fragment constructs an array object, allocating space for eight elements. It uses the new keyword followed by the type of the elements the array will store. In square brackets is the number of elements the array will be able to hold.

```
this.persons = new Person[8];
```

Of course, including a different number in place of the 8 would allocate space for a different number of elements. The 8 in this example can also be replaced with any expression that evaluates to an integer, including a simple variable or a complex calculation. This calculation may, for example, be based on information obtained from a user, as shown in the following code fragment:

```
public class BBBS extends Object
{ private Person[] persons;
  ...

  private void createArray()
  { Scanner in = new Scanner(System.in);
    System.out.print("How many persons:");
    int numPersons = in.nextInt();

    this.persons = new Person[numPersons];
    ...
  }
}
```

KEY IDEA

An array may be declared and allocated in one statement when you know how many elements it will hold.

The programmer often knows how many elements will be in the array when the program is written. In this case, the declaration and the allocation may be combined:

```
private Person[] persons = new Person[100];
```

10.2.3 Initialization

The final step in creating an array is to initialize each element, as illustrated in Step 3 of Figure 10-13. The simplest approach is to call an appropriate constructor for each element in the array. For example, a small array of `Person` objects could be initialized like this:

```
this.persons[0] = new Person("Steve", "1968/12/24",
                             Gender.MALE, Role.BIG);
this.persons[1] = new Person("Ken", "1997/8/7",
                             Gender.MALE, Role.LITTLE);
this.persons[2] = new Person("Beth", "1993/8/27",
                             Gender.FEMALE, Role.LITTLE);
```

This approach works, but is impractical for a large number of elements. Array initialization is often performed by reading information from a file and constructing an object for each of the file's records.

The main problem is knowing how many records are in the file. This information is needed to allocate the correct number of elements for the array.

One approach is to simply count the records. The file is opened and the records are read, counting each one. When the end of the file is reached, it is closed and then opened again. The array is allocated using the count just obtained. The entire file is then read a second time, storing each object in the array.

Listing 10-7 shows the constructor to the `BBBS` class in lines 14–36. The initialization of the array takes place in the constructor. The relevant points are:

- The array is declared at line 10.
- In lines 18–24, the file is opened, every record is read and counted, and then the file is closed.
- In line 27, the array is allocated using the count of the records in the file.
- In lines 30–33, the file is again opened and the records read. This time, however, the objects created with the data are stored in the array at line 32. The file is closed again in line 34 after all of the records have been read.

LOOKING AHEAD

Reading objects from a file was discussed in Section 9.2.1.

Listing 10-7: Initializing an array from a file

```
1 import java.util.Scanner;
2
3 /** A list of the "bigs" and "littles" associated with a Big Brother/Big Sister program.
4  * "Bigs" are the Big Brothers and Big Sisters; "littles" are the Little Brothers and Sisters
5  * they are (potentially) paired with.
6
7  * @author Byron Weber Becker */
```



[ch10/bbbs/](#)

Listing 10-7: *Initializing an array from a file* (continued)

```
8 public class BigBroBigSis extends Object
9 {
10     private Person[] persons;           // the list of bigs and littles
11
12     /** Construct a new object by reading all the bigs and littles from a file.
13     * @param fileName the name of the file storing the information for bigs and littles */
14     public BigBroBigSis(String fileName)
15     { super();
16
17         // Count the number of Persons in the file.
18         int count = 0;
19         Scanner in = this.openFile(fileName);
20         while (in.hasNextLine())
21         { Person p = new Person(in);
22           count++;
23         }
24         in.close();
25
26         // Allocate an array to hold each object we read.
27         this.persons = new Person[count];
28
29         // Read the data, storing a reference to each object in the array.
30         in = this.openFile(fileName);
31         for (int i = 0; i < count; i++)
32         { this.persons[i] = new Person(in);
33         }
34         in.close();
35
36     }
37     ...
99 }
```

One disadvantage of reading the file twice is inefficiency. Reading from a file is inherently slow and it would be more efficient to avoid reading the entire file twice.

Another approach is to store the number of records as the first item in the file, as shown in Figure 10-14. The constructor can simply read this data item and allocate the array. The records can then be read and stored into the array the first time the file is read.

```
5
Kenneth A Parsons
1997/8/7 M L
Beth A Reyburn
1993/8/27 F L
Kathleen A Waller
1979/5/4 F B
Roydyn A. Clayton
1993/5/25 M L
Christopher Aaron Fairles
1981/2/2 M B
```

(figure 10-14)

File with the number of records stored as the first data item

A disadvantage of this approach is that the number of records must be kept accurate. This may be hard to guarantee if the file is edited directly by users. However, it is not difficult if the file is always created by a program.

LOOKING AHEAD

An array that appears to grow can also solve this problem. See Section 10.4.

Listing 10-8 shows a constructor using this approach. It could be substituted for the constructor shown in Listing 10-7, provided the data file were changed to include the number of records in the file.

Listing 10-8: Initializing an array when the data file contains the number of records

```
1 public BigBroBigSis(String fileName)
2 { super();
3   Scanner in = this.openFile(fileName);
4
5   // Get the number of records in the file.
6   int count = in.nextInt();
7   in.nextLine();
8
9   // Allocate an array to hold each record we read.
10  this.persons = new Person[count];
11
12  // Read the data, storing a reference to each object in the array.
13  for (int i = 0; i < count; i++)
14  { this.persons[i] = new Person(in);
15  }
16  in.close();
17 }
```

Array Initializers (optional)

Java provides a handy shortcut to initialize an array if you know its contents when you write the program. Essentially, you place the array elements in a comma-separated list between curly braces, as shown in the following example:

```
bbbs.persons = new Person[ ]
{ new Person("Byron", "1961/3/21",
  Gender.MALE, Role.BIG),
  new Person("Ann", "1960/12/3",
  Gender.FEMALE, Role.BIG),
  new Person("Luke", "1990/10/1",
  Gender.MALE, Role.LITTLE),
  new Person("Joel", "1994/2/28",
  Gender.MALE, Role.LITTLE)
};
```

Java will automatically create an array of the right length to hold all the elements listed. In fact, if you try to specify the size yourself, the compiler will give you an error.

10.3 Passing and Returning Arrays

Like other reference variables, references to arrays can be passed to a method via parameters and returned from a method using the `return` keyword.

LOOKING AHEAD

Problem 12.13 generalizes this method with interfaces and polymorphism.

One common activity that demonstrates both passing and returning arrays is to extract a subset from a larger array. For example, return an array of `Person` objects that contains only “bigs” who are female. To make the method more versatile, we’ll pass the desired gender and role as arguments. The method’s signature is as follows:

```
public Person[ ] extractSubset(Gender g, Role r)
```

The return type of `Person[]` indicates that the method will return a reference to an array of `Person` objects.

To solve this problem, we need to create an appropriately sized array—which means figuring out the size of the subset. Then we need to fill the array. In pseudocode, we can state our tasks as follows:

```
size = count number of elements in the subset
subset = a new array to store size elements
fill subset with the appropriate objects
return subset
```

The first step, counting the size of the subset, is an application of the Process Matching Elements pattern in which the process performed is simply counting. Its signature and method documentation are as follows; implementing it is Problem 10.7.

```
/** Count the number of persons matching the given gender and role.
 * @param g    The gender of persons to be included in the subset.
 * @param r    The role of the persons to be included in the subset. */
private int countSubset(Gender g, Role r)
```

The second step, allocating a temporary array, illustrates that declaring and allocating an array within a method is both possible and useful. As always, the access modifier, such as `private`, is omitted when declaring a temporary variable.

```
Person[] subset = new Person[size];
```

The third step, filling the `subset` array, is the tricky one. We'll pass the method the gender and role of the `Person` objects desired, as well as a reference to the temporary array. The method's signature will be:

```
private void fillSubset(Person[] ss, Gender g, Role r)
```

Again, notice the type `Person[]`. The parameter variable `ss` will refer to an array of `Person` objects. Like other references passed as parameters, `ss` will contain an alias to `subset`; both references refer to the same array and both can be used to access and change the contents of the array. The reference itself cannot be changed, but the thing it refers to can be changed.

Inside the method, we'll repeatedly find the next person object with the appropriate gender and role, copying a reference to it into the next available space in the temporary array. This will require two index variables, one to keep track of where we are in the `persons` array and the other to track our position in the `subset` array.

Figure 10-15 shows the situation immediately after the first `Person` object has been inserted into the subset. The index variable `ssPos` ("subset position") gives the index of the next available position in the `subset` array. The variable `arrPos` ("array position") gives the index of the next `Person` object to consider. The colored arrows show `Person` objects that have yet to be copied.



PATTERN

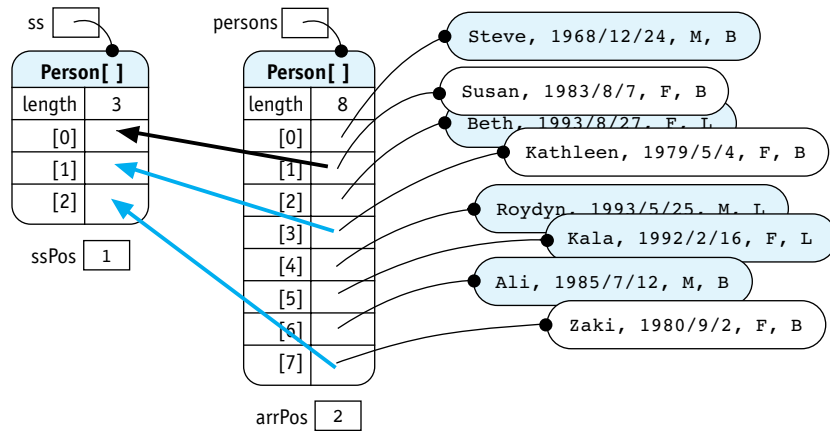
*Process Matching
Elements*

LOOKING AHEAD

*Aliases were
discussed in
Section 8.2.2.*

(figure 10-15)

Filling the subset array, immediately after the first Person object reference has been copied to the subset array



The code for the helper method is shown in lines 27–38 of Listing 10-9. Notice that `ssPos` is only incremented when a new element is added to the subset (line 34) but that `arrPos` is incremented each time a new `Person` object is considered (line 36).

The final step in the `extractSubset` method is to return a reference to the subset array (line 13).

FIND THE CODE



ch10/bbbs/

Listing 10-9: Completed code for the `extractSubset` method

```

1 public class BigBroBigSis extends Object
2 { private Person[] persons; // The list of bigs and littles.
3
4 ...
5
6 /** Extract a subset of all the persons who have the given gender and role.
7  * @param g The gender of all members of the subset.
8  * @param r The role of all members of the subset. */
9 public Person[] extractSubset(Gender g, Role r)
10 { int ssSize = this.countSubset(g, r);
11   Person[] subset = new Person[ssSize];
12   this.fillSubset(subset, g, r);
13   return subset;
14 }
15
16 /** Count the number of persons matching the given gender and role.
17  * @param g The gender of persons to be counted.
18  * @param r The role of the persons to be counted. */
19 private int countSubset(Gender g, Role r)
20 { // to be completed as an exercise
21 }

```


10.4 Dynamic Arrays

So far, the number of elements stored in our arrays has been fixed. We've neither added elements nor removed them. To be truly useful, this must change. For example, in the Big Brother/Big Sister program, we need a method to add a new person to the `persons` array:

```
/** Add another person to the array of Person objects.
 * @param p The Person object to add. */
public void add(Person p)
```

To implement `add`, we must figure out how to “create” additional space in the array. In this section, we'll explore two approaches to this problem, and ultimately conclude that the best solution uses features of both.

10.4.1 Partially Filled Arrays

KEY IDEA

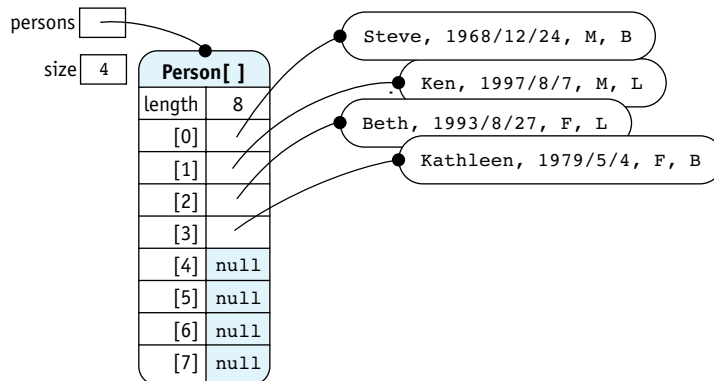
Allocate extra space for the array. Use the first elements to store data. Keep the number of elements in use in another variable.

The first approach uses a simple idea: Create an array with room to grow, if necessary. This separates the notion of the size of the array (the number of elements it currently stores) from the length of the array (the maximum number of elements it can store). This requires an auxiliary variable that we usually name `size`. Such an array is usually only partly filled, so we'll call it a **partially filled array**.

We will adopt a convention that indices in the range `0..size-1` will hold the valid elements while indices `size..length-1` will be “empty.” This is illustrated in Figure 10-16.

(figure 10-16)

Partially filled array with four elements



The auxiliary variable, `size`, can be interpreted two ways. First, it can be interpreted as the number of elements in the array that store valid data. This interpretation is useful for the Process All Elements and related patterns. For example, to print all the names in the partially filled `persons` array, we write

```
for (int i = 0; i < this.size; i++)
{ Person p = this.persons[i];
  System.out.println(p.getName());
}
```

Notice the use of `this.size` rather than `this.persons.length` to control the loop. If the array is as shown in Figure 10-16, using `length` would result in a `NullPointerException` when the name for `persons[4]` is printed because `p` would be null.

The other Process All Elements idiom, using the *foreach* loop, will not work with partially filled arrays. Writing `for (Person p : this.persons)` is the same as writing `for (int i = 0; i < this.persons.length; i++)`.

The second interpretation of `size` is as the first element of the “empty” portion of the array. This interpretation is the natural one for the `add` method because it tells us where to put the new element.

```
public void add(Person p)
{ this.persons[this.size] = p;
  this.size++;
}
```

After a new element is added, the auxiliary variable must be incremented.

Of course, if the array is already full (`size` has the same value as `persons.length`), the `add` method will fail with an `ArrayIndexOutOfBoundsException`. We will investigate a solution to this problem shortly.

Inserting into a Sorted Array

If the array is already sorted and you want to keep it sorted, simply adding the new element to the end isn’t good enough. One approach would be to add to the end and then sort the entire array, but that is inefficient. A much better approach is to move elements larger than the new element down in the array. The new element can then be inserted in the resulting “hole.” These steps are shown in Figure 10-17.

KEY IDEA

size says how many elements have valid data.

KEY IDEA

The foreach loop doesn’t work for partially filled arrays.

KEY IDEA

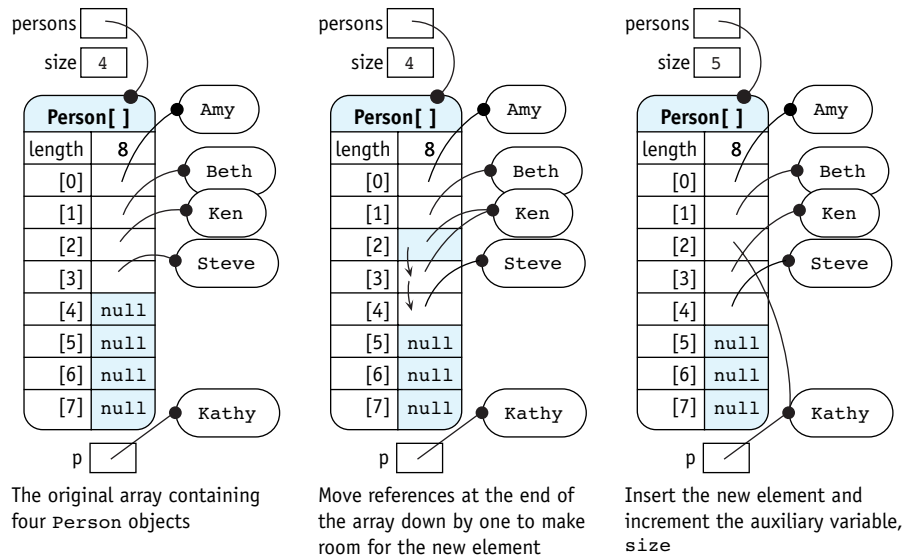
size also says where the next element should be added.

LOOKING AHEAD

Implementing this algorithm is Problem 10.8.

(figure 10-17)

Inserting a new element in
an array sorted by name



Deletion

When deleting an element, we need to fill the “hole” left by the deleted element so that all the valid array elements are kept at the beginning of the partially filled array and all the unused space at the end. We’ll use the following algorithm:

d = find the index of the element to delete
 fill *d* with another element from the array
 decrement *size*, the auxiliary variable
 assign null to the element at *size*

The first step may be trivial if we are given the index of the element to delete. In other situations, we may need to search for the element to find the index.

The second step varies, depending on whether a sorted order must be maintained. If the array is unsorted, use the last element of the array to replace the element being deleted. In a sorted array, the elements with indices larger than *d* all need to be moved up one position in the array.

The third step recognizes that there is now one less element in the array.

LOOKING AHEAD

Written Exercise 10.1 asks you to explain why this step is optional.

The last step is not strictly necessary, however it is a good idea to assign null to the element for two reasons. First, it can make debugging easier because accidentally accessing an element in the unused portion of the partially filled array will generate a `NullPointerException`, quickly informing us that we made a mistake. Second, it may free an object for garbage collection, thereby reducing the memory required by our program.

Problems with Partially Filled Arrays

Unfortunately, partially filled arrays pose two significant problems. First, a partially filled array solves the problem of adding elements to an array, but only up to a point. There is still a limit. If the array is initially allocated to hold 500 elements, we can't insert 501. The last one just won't fit. Using the algorithms discussed earlier will result in an `ArrayIndexOutOfBoundsException`. If this abrupt ending to the program isn't desired, a check with a friendlier message can be made:

```
public void add(Person p)
{ if (this.size < this.persons.length)
  { this.persons[this.size] = p;
    this.size++;
  } else
  { // error message
  }
}
```

One way of addressing the first problem is to allocate arrays with more space than we think we'll ever use. Unfortunately, this leads to the second problem with partially filled arrays—wasting lots of memory. In addition, history is filled with programmers who dramatically misjudged how much data would be poured into their programs. For example, a program written to handle people associated with the local chapter of Big Brothers/Big Sisters might be deployed nationally and suddenly need to deal with *much* more information.

In spite of these two problems, partially filled arrays are a great solution where the amount of data can be reliably estimated.

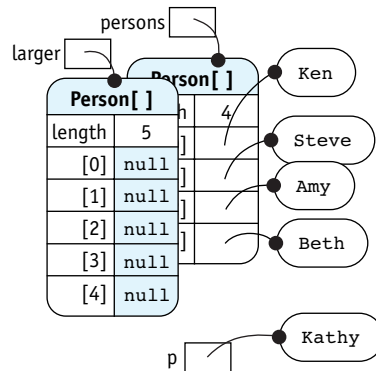
10.4.2 Resizing Arrays

A second approach to the problem of adding and deleting elements in an array is to “change” the size of the array. Once an array is allocated, its size can't be changed, but we can allocate a new array with a different size and then copy the elements from the old array to the new array. After updating the array's reference to point to the new array, it appears as though the array has simply grown. The new element can then be added. These four steps are shown in Figure 10-18.

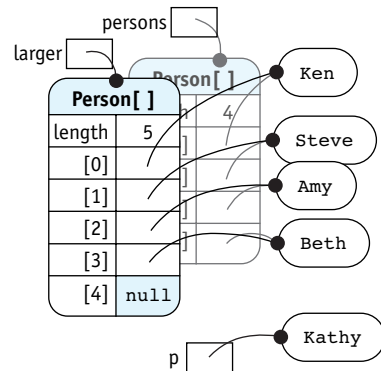
KEY IDEA

Arrays can't change size, but we can make it appear as if they do.

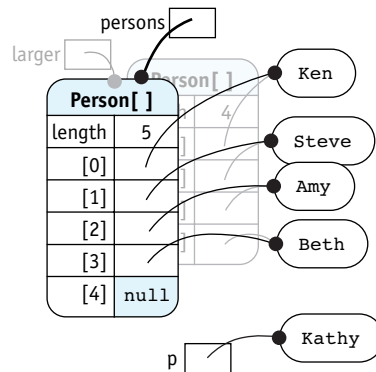
(figure 10-18)
Reallocating an array



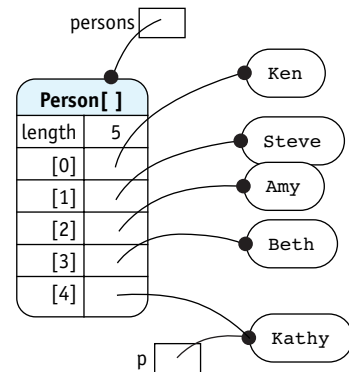
Step 1: Allocate a new, larger array



Step 2: Copy the contents to the larger array



Step 3: Reassign the array reference



Step 4: Add the new element

The code to add a person to an unordered array is shown in Listing 10-10.

Listing 10-10: Adding a Person object to an unordered array

```

1 public class BBBS extends Object
2 { private Person[] persons;
3
4   ...
5
6   /** Add a new person to the persons array.
7    * @param p      The new person to add. */
8   public void add(Person p)
9   { // Step 1: Allocate a larger array.
10      Person[] larger = new Person[this.persons.length + 1];
11

```

Listing 10-10: Adding a Person object to an unordered array (continued)

```

12 // Step 2: Copy elements from the old array to the new, larger array.
13 for (int i = 0; i < this.persons.length; i++)
14 { this.larger[i] = this.persons[i];
15 }
16
17 // Step 3: Reassign the array reference.
18 this.persons = larger;
19
20 // Step 4: Add the new element.
21 this.persons[this.persons.length-1] = p;
22 }
23 }

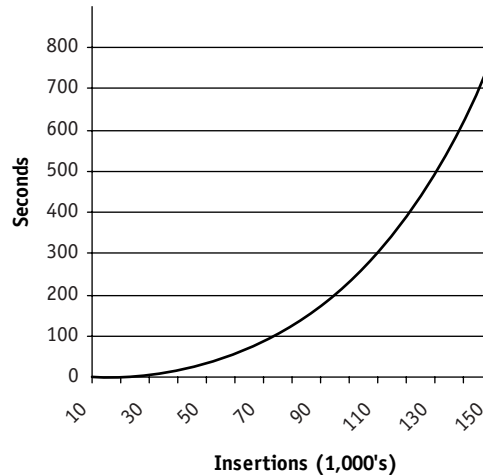
```

There is, however, a big disadvantage to this approach. Inserting many elements is very time consuming because so much copying is required. For example, one test¹ produced the data shown in Figure 10-19. The first column shows the number of insertions. The second column shows the time, in seconds, required to make the insertions into an array that grows by one with each insertion. The last column shows the number of seconds required to insert the same data into a partially filled array.

a) Time to insert into an array

Insertions	Grow	PFA
10,000	0.4	0.000
20,000	1.8	0.000
30,000	6.1	0.000
40,000	14.2	0.015
50,000	28.4	0.015
60,000	46.8	0.015
70,000	78.3	0.015
80,000	123.3	0.015
90,000	179.8	0.015
100,000	239.2	0.015
110,000	304.4	0.015
120,000	389.6	0.015
130,000	476.8	0.015
140,000	623.7	0.015
150,000	779.8	0.015

b) Graphing the time to insert data into an array that grows



(figure 10-19)

Inserting elements in an array

¹ Using the code in `examples/ch10/growArrayTest` on a machine with a 2.8GHz Pentium 4 CPU and 1G of RAM running Windows XP and Java 5.

The test clearly shows that the more insertions there are, the worse the problem is. For example, the time taken to insert the first 10,000 items is less than half a second. Inserting the last 10,000 items, however, requires more than three minutes. Meanwhile, inserting 150,000 items into a partially filled array is so fast the computer's clock isn't accurate enough to time it and on the graph it can't be distinguished from the x axis.

10.4.3 Combining Approaches

The disadvantages of a partially filled array are an upper limit on the number of insertions and wasted space if some program executions use lots of data but most do not. On the other hand, expanding the array with each insertion solves those two problems, but introduces a performance problem.

KEY IDEA

Expandable, partially filled arrays give the best of both approaches.

Combining the two approaches addresses all three issues. The strategy is to use a partially filled array. When it gets full, allocate a larger array. However, don't increase the array by only one element. Instead, double the size of the array. That typically wastes some space, but not more than a factor of two. If that's too much, the array could be increased by 25% each time it is enlarged.

The same test as shown in Figure 10-19 takes only 0.047 seconds to insert 150,000 items—a little worse than a partially filled array that is initially allocated to hold 150,000 items, but not nearly as bad as growing the array by one each time.

The `ArrayList` class in the Java library uses exactly this approach. It is simply a partially filled array that can grow when it gets full, wrapped in a class.

Listing 10-11 shows an `add` method for a partially filled array that is doubled whenever it becomes full. Note that this same method can be used in the constructor, eliminating the need to count the number of items in the file (compare Listing 10-11 with Listing 10-7).

FIND THE CODE



`ch10/
bbbsPartiallyFilled/`

Listing 10-11: Initializing and adding to an expandable, partially filled array

```

1 public class BigBroBigSis extends Object
2 {
3     private Person[] persons = new Person[1]; // List of bigs and littles.
4     private int size; // Actual number of persons.
5
6     /** Construct a new object by reading all the bigs and littles from a file.
7     * @param fileName The name of the file storing the information for bigs and littles. */
8     public BigBroBigSis(String fileName)
9     { super();
10

```

Listing 10-11: *Initializing and adding to an expandable, partially filled array* (continued)

```
11 // Read the data, adding each person to the array
12 Scanner in = this.openFile(fileName);
13 while (in.hasNextLine())
14 { this.add(new Person(in));
15 }
16 in.close();
17 }
18
19 /** Add a person to the the list of persons. */
20 public void add(Person p)
21 { if (this.persons.length == this.size)
22 { // The array is full -- grow it.
23     Person[] larger = new Person[this.size * 2];
24     for (int i = 0; i < this.size; i++)
25     { larger[i] = this.persons[i];
26     }
27     this.persons = larger;
28 }
29 this.persons[this.size] = p;
30 this.size++;
31 }
32 }
```

10.5 Arrays of Primitive Types

So far we have only discussed arrays of objects. Java also allows arrays of primitive types such as integers, Booleans, and doubles. Arrays of primitives and arrays of objects share many similarities. For example, declaring and allocating an array of four doubles bears a striking resemblance to declaring and allocating an array of four `Person` objects:

```
Person[] persons = new Person[4];
double[] interests = new double[4];
```

In these examples, each element in `persons` is automatically initialized to `null` and each element in `interests` is automatically initialized to `0.0`.

10.5.1 Using an Array of double

The `Person` class used in the Big Brother/Big Sister program defines four variables to store potential interests of the participants: the extent to which they like sports, crafts,

games, and the outdoors. A value of 0.0 indicates they don't have an interest in it at all whereas a value of 1.0 indicates a very high interest. Before two people are paired, their compatibility is determined with the `getCompatibility` query:

```
public double getCompatibility(Person p)
{ return (this.likesCrafts * p.likesCrafts
        + this.likesGames * p.likesGames
        + this.likesOutdoors * p.likesOutdoors
        + this.likesSports * p.likesSports)
        /4.0;
}
```

Suppose it was determined that these four interests need to be supplemented with an additional 16, for a total of 20 different interests. Using separate variables for each one would be tedious; an array is a much better choice. Using an array, the `Person` class is written as shown in Listing 10-12.

Listing 10-12: *Using an array of doubles to represent interests*

```
1 public class Person extends Object
2 { ...
3   private static final int NUM_INTERESTS = 20;
4   private double[] interests = new double[NUM_INTERESTS];
5   ...
6
7   public Person(Scanner in)
8   { ...
9     // Read this person's interests from the file.
10    for (int i = 0; i < Person.NUM_INTERESTS; i++)
11    { this.interests[i] = in.nextDouble();
12    }
13    ...
14  }
15
16  /** How compatible is this person with person p? A score of 0.0 means not at all
17   * compatible; 1.0 means extremely compatible. */
18  public double getCompatibility(Person p)
19  { double compat = 0.0;
20    for (int i = 0; i < Person.NUM_INTERESTS; i++)
21    { compat = compat + this.interests[i] * p.interests[i];
22    }
23    return compat / Person.NUM_INTERESTS;
24  }
25 }
```

10.5.2 Meaningful Indices

So far the indices of our arrays have been just positions. They haven't had any meaning attached to them, though it is sometimes useful to do just that. Suppose, for example, that we wanted to know the distribution of ages of the people participating in the Big Brother/Big Sister program. That is, we want to know how many people are 10 years old, how many are 11, and so on. We'll assume no one is over 200 years old.

To solve this problem we can allocate an array named `ageCounters` with 200 elements. Each element will be a counter for a particular year. Which year? The year corresponding to the index. Thus, `ageCounters[10]` will be the number of 10-year-olds and `ageCounters[25]` will be the number of 25 year-olds. We'll have a counter for everyone between 0 and 199 years old, inclusive.

The method shown in Listing 10-13, when added to the `BigBroBigSis` class, will return a filled array giving the number of participants for each age. It could be used like this:

```
int[] ages = bbbs.getAgeCounts();
for (int i = 0; i < ages.length; i++)
{ if (ages[i] > 0)
  { System.out.println ("There are " + ages[i] +
    " participants that are " + i + " years old.");
  }
}
```

Listing 10-13: *A method to count the participants in each age group*

```
1 public class BigBroBigSis extends Object
2 { private Person[] persons;
3   private int size = 0;
4
5   ...
6
7   /** Find the number of participants in each age group.
8    * @return A filled array where a[i] is the number of people i years old. */
9   public int[] getAgeCounts()
10  { int[] ageCounters = new int[200];
11    for (int i = 0; i < this.size; i++)
12    { int age = this.persons[i].getAge();
13      ageCounters[age]++;
14    }
15    return ageCounters;
16  }
17 }
```

 [FIND THE CODE](#)

`ch10/
bbbsPartiallyFilled/`

In the last example, the indices naturally matched ages because both ranges start at 0. Sometimes that isn't the case. Consider a slight modification of this problem: Count the number of coins in a collection by the year they were minted. Assume the oldest coin was minted in 1850.

This problem could be solved by allocating an array with 1850 unused elements. A better approach is to offset the indices by 1850, as shown in Listing 10-14. The crucial lines are 5, 16, and 22. In line 5, the constants `EARLIEST` and `LATEST` are used to calculate the actual number of elements or counters that are needed. This avoids the unused elements at the beginning of the array. In line 16, the year entered by the user is reduced by the appropriate amount so that it can be used as an index into the array. In line 22, the reverse is done to map the index to the appropriate year.

Listing 10-14: *Offsetting an index to start at zero*

```
1  /** Count the number of coins minted in each year. */
2  public static void main(String[] args)
3  { int EARLIEST = 1850;
4    int LATEST = 2008;
5    int[] ages = new int[LATEST - EARLIEST + 1];
6
7    // Count the coins.
8    Scanner in = new Scanner(System.in);
9    while (true)
10   { System.out.print("Enter a mint year or -1 to exit: ");
11     int yr = in.nextInt();
12     if (yr == -1)
13     { break;
14     }
15
16     ages[yr - EARLIEST]++;
17   }
18
19   // Print out the number of coins for each year.
20   for (int i = 0; i < ages.length; i++)
21   { System.out.println(ages[i] +
22     " coins minted in " + (i + EARLIEST));
23   }
24 }
```

10.6 Multi-Dimensional Arrays

Sometimes an array with more than one dimension is useful. For example, consider a two-dimensional (2D) array recording the money given to Big Brothers/Big Sisters by month and source. Figure 10-20 shows the source of the money across the top in categories such as United Way and government grants. Down the left side are the months. At the intersection of each row and column is the amount of money received in a particular category in a particular month. For example, the cell in the column labeled “Individual Donations” and in the row labeled “Apr” indicates that \$4,833 were received in April from individual donations.

	United Way	Corporate Donations	Individual Donations	Fundraising	Govt. Grants
Jan	0	3,000	6,915	0	15,500
Feb	0	2,125	4,606	0	5,500
Mar	0	2,000	5,448	0	5,500
Apr	0	3,000	4,833	13,983	15,500
May	20,569	2,000	6,091	0	5,500
Jun	0	8,000	4,867	0	5,500
Jul	0	3,000	4,196	0	15,500
Aug	0	2,550	4,736	0	5,500
Sep	0	2,000	4,305	0	5,500
Oct	0	3,000	5,286	32,254	15,500
Nov	0	2,000	6,834	0	5,500
Dec	9,351	2,000	7,459	0	5,500

(figure 10-20)

Two-dimensional array recording income by source and month

Java uses one pair of brackets for each dimension of an array. The one-dimensional arrays we used earlier in the chapter use one pair of brackets; the two-dimensional array shown in Figure 10-20 uses two. Of course, a three-dimensional array uses three pairs. The pattern continues for as many dimensions as you need.

```
int[][] income = new int[12][5]
```

The declaration on the left side of the equal sign specifies a 2D array where each cell stores an integer. The allocation on the right side specifies that the array has 12 rows and five columns.

Figure 10-20 is actually a bit misleading, for the following reasons:

- Column names like “Corporate Donations” and row names like “May” are not directly associated with an array. The array itself is declared to store only integers. It cannot store strings as column or row labels.

KEY IDEA

The first pair of brackets is for the rows; the second pair of brackets is for the columns.

- Rows and columns must be accessed using integer indices.
- The variable name, `income`, actually refers to memory that holds the array; it isn't the array itself.

A more accurate picture of the array is shown in Figure 10-21 which takes all this into account.

(figure 10-21)
More accurate
visualization of a two-
dimensional array

		int[][]				
		0	1	2	3	4
0	0	3,000	6,915	0	15,500	
1	0	2,125	4,606	0	5,500	
2	0	2,000	5,448	0	5,500	
3	0	3,000	4,833	13,983	15,500	
4	20,569	2,000	6,091	0	5,500	
5	0	8,000	4,867	0	5,500	
6	0	3,000	4,196	0	15,500	
7	0	2,550	4,736	0	5,500	
8	0	2,000	4,305	0	5,500	
9	0	3,000	5,286	32,254	15,500	
10	0	2,000	6,834	0	5,500	
11	9,351	2,000	7,459	0	5,500	

10.6.1 2D Array Algorithms

Most algorithms that process a 2D array use two nested loops. The outside loop generally specifies which row to access and the inside loop generally specifies the column. A number of the following algorithms will display this general pattern. We say that such an algorithm accesses the array in **row-major order**. Some algorithms access the array in **column-major order**—the columns are indexed by the outer loop.

Printing Every Element

For example, to print the `income` array we could use a method like the one shown in Listing 10-15.

Listing 10-15: *Printing a 2D array*

```

1 public class BBBSIncome extends Object
2 { // income by month (row) and source (column)
3     private int[][] income;
4
5     ...
6
7     /** Print the income chart. */
8     public void printIncomeChart()
9     { for (int r = 0; r < this.income.length; r++)
10        { for (int c = 0; c < this.income[r].length; c++)
11            { System.out.print(this.income[r][c] + "\t");
12                }
13            System.out.println();
14        }
15    }
16 }

```



ch10/income/

The inside loop, lines 10–12, prints one entire row each time it executes. The row it prints is specified by the outer loop, row `r`. After the row is printed, line 13 ends the current line of text and begins a new line. This process of printing a row is repeated for each row specified by the outer loop.

Notice that the number of rows is found in line 9 with `this.income.length` while the number of columns in a particular row is found in line 10 with `this.income[r].length`. They differ because in Java a 2D array can be ragged—each row may have its own length. We will see an example of this in Section 10.6.3.

Sum Every Element

The same nested looping pattern can be used to find the total income, from all sources, for the entire year:

```

/** Calculate the total income for the year. */
public int getTotalIncome()
{ int total = 0;
  for (int r = 0; r < this.income.length; r++)
    { for (int c = 0; c < this.income[r].length; c++)
        { total = total + this.income[r][c];
        }
    }
  return total;
}

```

KEY IDEA

It's possible to find the number of rows in a 2D array, as well as the number of columns in each row.

Every time you need to examine every cell in a 2D array, you will likely use this nested looping pattern.

Summing a Column

To find the total of the individual donations in one year, we need to sum column 2 in the `income` array. This task requires a single loop because it is working in a single dimension—moving down the column. Passing the column index as a parameter makes the method more flexible:

```
/** Calculate the total income for a given category for the year.
 * @param columnNum    The index of the column containing the desired category. */
public int getTotalByCategory(int columnNum)
{ int total = 0;
  for (int r = 0; r < this.income.length; r++)
  { total = total + this.income[r][columnNum];
  }
  return total;
}
```

10.6.2 Allocating and Initializing a 2D Array

As with a one-dimensional array, the declaration and allocation of the array can be split. This means that determining the size of an array can be delayed until the program is actually executing. For example, the array could be initialized from a file where the first two numbers indicate the number of rows and columns, respectively.

The first five rows of such a data file are shown in Figure 10-22. The constructor shown in Listing 10-16 shows how the array is allocated and then initialized using this data. The size of the array is determined in lines 11 and 12. The array itself is allocated using those sizes in line 16. Finally, the data is read and stored in the array using the by now familiar double loop in lines 19-24. The calls to `nextLine` in lines 13 and 23 are not strictly necessary because `nextInt` will read across line boundaries; however, using `nextLine` shows where line endings are expected in the file and adds to the clarity of the code.

(figure 10-22)
Sample data file

```
12 5
0    3000    6915    0    15500
0    2125    4606    0    5500
0    2000    5448    0    5500
0    3000    4833    13983 15500
...
```

Listing 10-16: *Allocating and initializing a 2D array from a file*

```
1 public class BBBSIncome extends Object
2 {
3     // Income by month (row) and source (column).
4     private int[][] income;
5
6     /** Read the income data from a file.
7      * @param in      The open file containing the data. */
8     public BBBSIncome(Scanner in)
9     { super();
10
11         // Get the size of the array.
12         int rows = in.nextInt();
13         int cols = in.nextInt();
14         in.nextLine();
15
16         // Allocate the array.
17         this.income = new int[rows][cols];
18
19         // Fill the array.
20         for (int r = 0; r < this.income.length; r++)
21         { for (int c = 0; c < this.income[r].length; c++)
22             { this.income[r][c] = in.nextInt();
23             }
24         in.nextLine();
25     }
26 }
```

[ch10/income/](#)

10.6.3 Arrays of Arrays

The picture we've used so far of a 2D array having rows and columns is adequate in most circumstances (see Figure 10-21). However, it doesn't match reality and sometimes knowing all the details is useful.

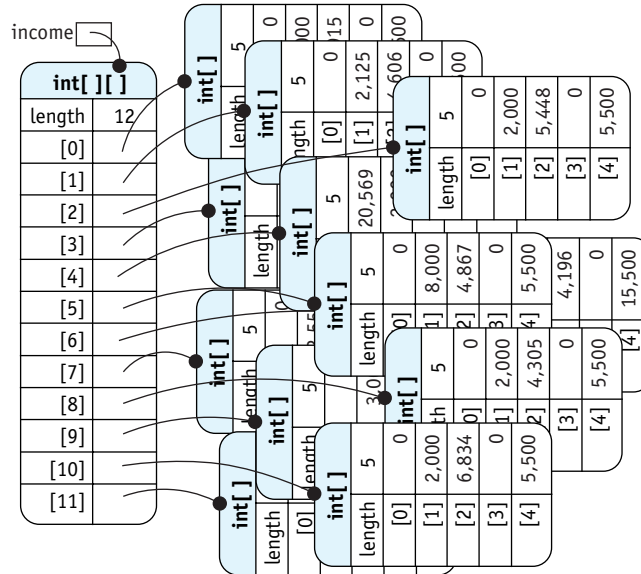
In reality, a 2D array is an array of arrays, as illustrated in Figure 10-23. The variable `income` actually refers to a one-dimensional array with 12 elements. Each element in that 1D array refers to an array with five elements—a “row” of the 2D array.

We can now understand accessing the number of rows and columns in an array. When we write `this.income.length`, it returns the length of the array holding the rows—the number of rows in the 2D array. When we write `this.income[r].length`, it

returns the length of the array stored in `income[r]`—the length of row `r`, or the number of columns in that row.

(figure 10-23)

Viewing a 2D array as an array of arrays



Sometimes, viewing a 2D array this way can work to our advantage in writing a program, too. For example, suppose you want to swap row `r` and row `s` in the array `income`. Rather than swap each element in row `r` with the corresponding element in row `s`, we can write:

```
int[] temp = income[r];
income[r] = income[s];
income[s] = temp;
```

The first line declares a temporary variable to store a 1D array. Then the rows are swapped by swapping their references. There is no equivalent way to swap columns.

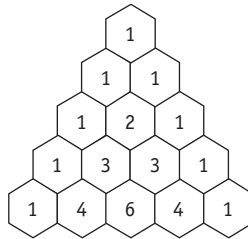
Another way in which the array of arrays viewpoint can make a difference in our code is a method that takes an entire row as a parameter. For example, we might already have a simple utility method to sum a 1D array:

```
private int sum(int[] a)
{ int sum = 0;
  for (int i = 0; i < a.length; i++)
  { sum = sum + a[i];
  }
  return sum;
}
```

We can find the sum of the entire `income` array by passing `sum` a row at a time:

```
public int getTotalIncome()
{ int total = 0;
  for (int r = 0; r < this.income.length; r++)
  { total = total + this.sum(this.income[r]);
  }
  return total;
}
```

A final use of the array-of-arrays view is when rows of the array have different lengths. For example, Blaise Pascal explored the many properties of a pattern of numbers that has come to be known as “Pascal’s Triangle.” The first five rows of the triangle are shown in Figure 10-24. The first and last element of each row is 1. The elements in between are the sum of two elements from the row before it.



(figure 10-24)

Pascal’s Triangle

A 2D array to store the first 10 rows of Pascal’s Triangle can be declared and allocated with the following statement:

```
int[][] pascal = new int[10][];
```

Notice that the last pair of brackets is empty. This causes Java to allocate only one dimension of the array. We can now allocate the rest of the array—with each row having the appropriate length—with the following loop. It first allocates a 1D array the correct length and then inserts it into the `pascal` array.

```
for (int r = 0; r < pascal.length; r++)
{ pascal[r] = new int[r+1];

  // the array must still be initialized with the correct values!
}
```

LOOKING AHEAD

See Problem 10.12.

This solution provides two interesting elements: First, because each row is just the right length, no space is wasted. Second, the array can still be printed with our standard nested loop, as follows:

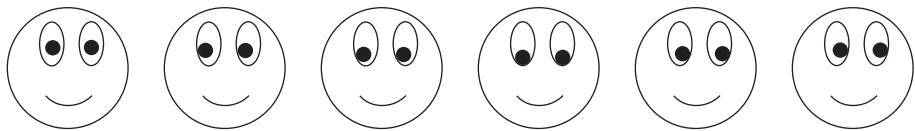
```
for (int r = 0; r < pascal.length; r++)
{ for (int c = 0; c < pascal[r].length; c++)
  { System.out.print(pascal[r][c] + "\t");
  }
  System.out.println();
}
```

10.7 GUI: Animation

There are several ways to perform animation in a graphical user interface. We've already seen a primitive animation in the Thermometer example in Section 6.7.3. In that example, the line representing the mercury in the thermometer was drawn several times, each time a little longer than before.

In Chapter 9, we saw how to display a single image from a file. In this section, we'll combine that capability with arrays to display a simple animation. The principle of this animation approach is to store a sequence of images in an array. The image displayed is switched from one image to the next quickly enough that it fools the eye into thinking there is smooth motion. Our example will use the six images shown in Figure 10-25. When shown repeatedly in quick succession, the eyes appear to roll. The images themselves were created with a graphics program that can create `.gif` files.

(figure 10-25)
Six images used in an
animation



Listing 10-17 and Listing 10-18 work together to show two happy face images, one with the eyes rolling clockwise and the other with the eyes rolling counterclockwise. One goes through the array forward as it displays the images; the other goes through the array backward as it displays the images.

The main method for the program is shown in Listing 10-17 and follows our standard pattern: Create the components we need (two instances of a custom component named `AnimateImage`), put them in an instance of `JPanel`, and then put the panel in an instance of `JFrame`.

Lines 25-28 start two threads, one for each animation. Just like threads allowed robots in Section 3.5.2 to move independently and simultaneously, these threads allow each animation to run independently of the other.

Listing 10-17: *The main method for an animation*

```
1 import javax.swing.*;
2
3 /** Create an animated image.
4  *
5  * @author Byron Weber Becker */
6 public class Main extends Object
7 {
8     public static void main(String[] args)
9     { // Create two animated components.
10        AnimateImage anim1 = new AnimateImage("img", 6, ".gif", 1);
11        AnimateImage anim2 = new AnimateImage("img", 6, ".gif", -1);
12
13        // Put the components in a panel and then in a frame.
14        JPanel contents = new JPanel();
15        contents.add(anim1);
16        contents.add(anim2);
17
18        JFrame f = new JFrame("Animations");
19        f.setContentPane(contents);
20        f.pack();
21        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22        f.setVisible(true);
23
24        // Run each animation in its own thread.
25        Thread t1 = new Thread(anim1);
26        t1.start();
27        Thread t2 = new Thread(anim2);
28        t2.start();
29    }
30 }
```




[ch10/animation/](#)

The component that actually does the animation is shown in Listing 10-18. Its key features are the following:

- An array to store the images comprising the animation is declared (line 10) and initialized with the images (lines 28–31).
- An instance variable, `currentImage`, holds the array index of the image currently being displayed.

- A method overriding `paintComponent` paints the image indexed by `currentImage` on the screen.
- A run method is required to implement the interface `Runnable`. When the thread is started in the main method, this is the method that runs. It loops forever. With each iteration, it advances `currentImage` to be either the next image or the previous image, depending on the value stored in the instance variable `direction`. After requesting that the system repaint the component by calling `repaint`, the method sleeps for 0.10 seconds to give the user time to see the new image.

FIND THE CODE 
 ch10/animation/

Listing 10-18: *A component that shows images in sequence to produce an animation*

```

1 import javax.swing.*;
2 import java.awt.*;
3
4 /** Instances of AnimateImage show a sequence of images to produce an animation.
5  *
6  * @author Byron Weber Becker */
7 public class AnimateImage extends JComponent
8     implements Runnable
9 {
10     private ImageIcon[] images;
11     private int currentImage = 0;
12     private int direction;
13
14     /** Construct a new animation component, loading all the images. Images are read from
15     * files whose names have three parts: a root string, a sequence number, and an extension.
16     *
17     * @param fileNameRoot    The root of the image filenames.
18     * @param numImages        The number of images in the animation.
19     * @param extension        The extension used for the images (e.g., .gif)
20     * @param dir              1 to animate going forward through the array; -1 to animate
21     *                          going backward through the array. */
22     public AnimateImage(String fileNameRoot, int numImages,
23                         String extension, int dir)
24     { super();
25       this.images = new ImageIcon[numImages];
26       this.direction = dir;
27
28       for (int i=0; i<numImages; i++)
29         { String fileName = fileNameRoot + i + extension;
30           this.images[i] = new ImageIcon(fileName);
31         }
32     }

```

Listing 10-18: *A component that shows images in sequence to produce an animation* (continued)

```
33     this.setPreferredSize(new Dimension(
34         this.images[0].getIconWidth(),
35         this.images[0].getIconHeight()));
36     }
37
38     /** Paint the current image on the screen. */
39     public void paintComponent(Graphics g)
40     { super.paintComponent(g);
41       Image img = this.images[this.currentImage].getImage();
42       g.drawImage(img, 0, 0, null);
43     }
44
45     /** Run the animation. */
46     public void run()
47     { while (true)
48       { // Select the next image and call for the system to repaint the component.
49         // If this.dir is negative, the remainder operator doesn't work as desired. Add
50         // this.images.length to compensate.
51         this.currentImage = (this.currentImage + this.direction
52             + this.images.length) % this.images.length;
53         this.repaint();
54         try
55         { Thread.sleep(100); // Use the sleep method in the Java library.
56         } catch (InterruptedException ex)
57         { // ignore
58         }
59       }
60     }
61 }
```

10.8 Patterns

Many patterns involve arrays. They include initialization and changing the size of an array, as well as many algorithms. This section contains only a sampling of what could be considered patterns in this chapter.

10.8.1 The Process All Elements Pattern

Name: Process All Elements

Context: You have a collection of values stored in an array and need to perform the same operation on all of them.

Solution: Use a `for` loop to process each element of the array, one element with each iteration of the loop. The following code template applies:

```
for («elementType» «elementName» : «arrayName»)  
{ «statements to process element»  
}
```

For example, to print the names of all the elements in the `persons` array:

```
for (Person p : this.persons)  
{ System.out.println(p.getName());  
}
```

Consequences: Each element in the array is processed by the statements inside the loop. If the array happens to be partially filled, the preceding form will cause a null pointer exception. Then the alternate form, which uses an explicit index and an auxiliary variable, should be used.

Related Patterns: The Process Matching Elements, Find an Extreme, Selection Sort, and many other patterns are specializations of the Process All Elements pattern.

10.8.2 The Linear Search Pattern

Name: Linear Search

Context: You have an indexed collection and are interested in objects in the collection that satisfy a particular property. You want to do one of the following tasks:

- determine whether an element satisfying the property exists in the collection
- determine the position of the first or last element in the collection that satisfies the property
- retrieve the first or last element in the collection that satisfies the property

Solution: Write a method that takes the criteria that identify the desired element as one or more parameters. Use the Process All Elements pattern to test each element of the array against the criteria. An element satisfying them can be saved and returned after the loop, or more efficiently, returned as soon as it is found. The following code template uses the early return approach and assumes a partially filled array.

```

public «typeOfElement» «methodName»(«type» «criteria»)
{ for (int i = 0; i < «auxVar»; i++)
  { «typeOfElement» «elem» = «arrayName»[i];
    if («elem» satisfies «criteria»)
      { return «elem»;
        }
    }
  return «failureValue»;
}

```

This basic pattern has many variations. Some of the differences are whether the array is partially filled, whether the element is guaranteed to be found, and whether you want to know whether such an element exists, its position, or the element itself.

Many people prefer to use a `while` loop instead of a `for` loop. In that case, use the following variant of the pattern. The `while` loop depends on short circuit evaluation to stop the loop when the element is not found. For this to work, the test for the index being in bounds must be first.

```

public «typeOfElement» «methodName»(«type» «criteria»)
{ int i = 0;
  while (i < «auxVar» &&
        !(«arrayName»[i] satisfies «criteria»))
    { i++;
      }

  if (i == «auxVar») { return «failureValue»; }
  else               { return «arrayName»[i]; }
}

```

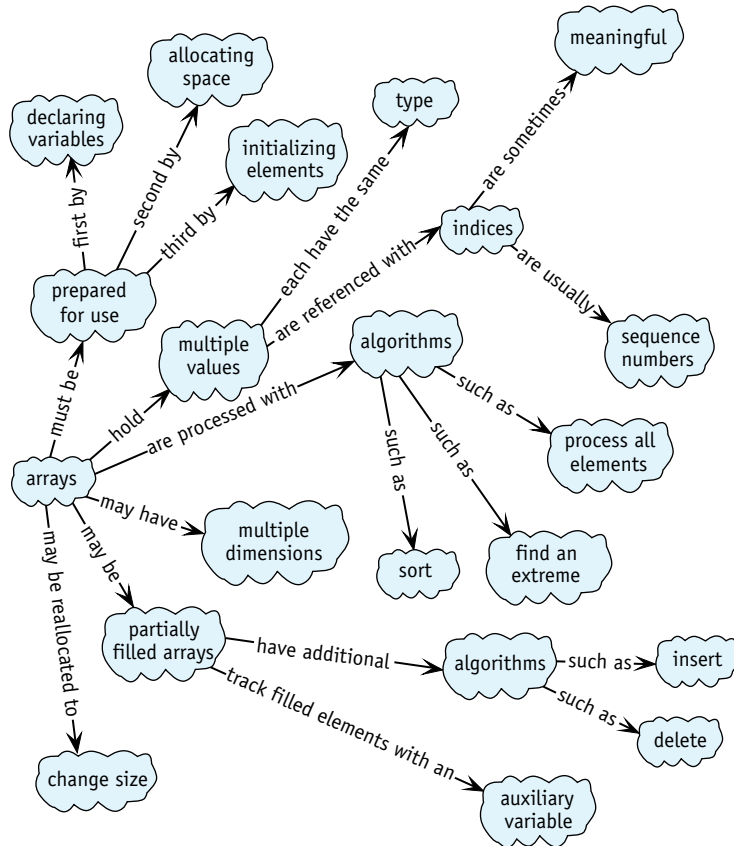
Consequences: The desired element is either found and returned, or a designated *«failureValue»* is returned. If the array contains objects, the *«failureValue»* is null. If the array contains primitive values, the failure value must be chosen carefully to avoid all valid values that could be stored in the array. If no such value exists, another technique must be used such as setting an instance variable as an error flag, returning an object that contains the primitive value or is null, or throwing an exception.

Related Patterns: Some variations of this pattern are similar to the Process All Elements pattern.

10.9 Summary and Concept Map

This chapter has focused on arrays, a fundamental programming structure for storing multiple values using a single name, with individual values referenced using an integer index. Arrays are closely related to collection classes such as `ArrayList`. An array should be used when efficiency matters or when more precise control over the size of the array is desired.

Many important algorithms apply to collections that are stored in an array. Examples include Process All Elements, Find an Extreme, Search, and Sort.



10.10 Problem Set

Written Exercises

- 10.1 In Section 10.4.1, it was noted that assigning `null` to an unused element after a deletion from a partially filled array is not strictly necessary. Explain why a program should work as implemented without that step. Drawing pictures may help.
- 10.2 Consider the code shown in Section 10.6.3 that swaps two rows of a 2D array.
 - a. Draw four diagrams, each one similar to Figure 10-23, that trace the three lines of code. Assume the array has five rows with three columns each and that `r` is one and `s` is three.

- b. Write pseudocode for a method that swaps two rows by swapping individual elements rather than entire rows.
- 10.3 Write patterns, in the same style as Section 10.8, for the following:
- a. Declaring, allocating, and initializing a filled array where the initial values are read from a file
 - b. Finding an extreme element
 - c. Deleting an element from a specified index in an unsorted, partially filled array
 - d. Inserting an element into a sorted, partially filled array
 - e. Enlarging a partially filled array

Programming Exercises

- 10.4 In Section 10.1.7, we found the oldest person by comparing the ages of everyone in the array. This, however, is accurate only to the nearest year. On 364 days of the year, a person born April 1, 1987 and another born April 2, 1987 will be the same number of years old—yet one is clearly older than the other. Rewrite the `findOldestPerson` method to compare their birth dates rather than their ages. With this modification, two people must be born on exactly the same day and year to be considered equally old. You will need to add a method to the `Person` class.
- 10.5 Write a method named `split`. This method is passed a `Scanner` object. It reads all of the tokens up to the end of the file, returning them as a filled array of strings (no blanks or nulls). Do not use the `split` method in the `String` class nor any of the collection classes.
- 10.6 The package `becker.xtras.hangman` includes classes implementing the game of Hangman. Figure 7-13 has an image of the user interface. Extend `SampleHangman`. Your new constructor should read a file of phrases that you create and store them in an array. Override the `newGame()` method to choose a random phrase from the array and then call the other `newGame` method with the chosen phrase. Create a `main` method, as shown in the package overview, to run your program.
- 10.7 Complete the `countSubset` helper method discussed in Section 10.3.
- 10.8 Write a method named `add` that adds a new `Person` object into a sorted, partially filled array. You may find Figure 10-17 helpful for this.
- 10.9 Implement a method with the signature `void delete(int d)` that deletes the element at index `d` from a partially filled array.
- a. Assume the partially filled array is unsorted.
 - b. Assume the array is sorted.
- 10.10 Write a program that reads a series of daily high temperatures from a file. Print out the number of days that each high was achieved. If you normally think of temperatures in degrees Celsius, assume the temperatures fall between -40° and 50° . If you normally think in Fahrenheit, assume they fall between -40° and 110° .

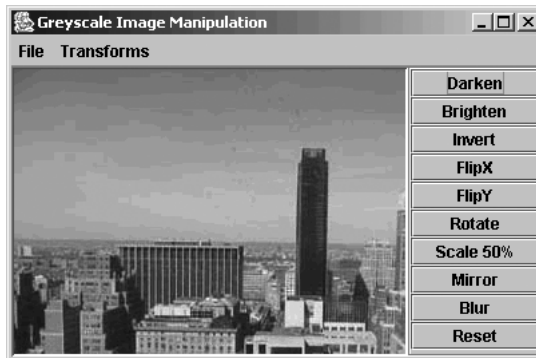
- 10.11 Write methods in the `BBBSIncome` class to:
- Find the month with the largest income in a given category.
 - Find the category with the largest income for a given month.
 - Find the month with the largest total income from all sources.
- 10.12 Review Pascal's Triangle and the code after Figure 10-24 to allocate an array for it.
- Draw an object diagram, similar to Figure 10-23, showing Pascal's Triangle as an array of arrays.
 - Complete the initialization code. Print the triangle using the algorithm in Listing 10-15 to verify the correctness of your code.
 - Write a method, `printFormatted`, that prints an array representing Pascal's Triangle with appropriate spacing. Your output will be spaced similarly to Figure 10-24, but will not display the background grid. You may find the `printf` method useful; see Section 7.2.4.
 - Write a method, `rowsSumToPowers`. It verifies that the sum of the numbers in each row is 2^n , where n is the row number. That is, the sum of row 0 is 2^0 (or 1) and the sum of row 1 is 2^1 (or 2). Use the `pow` method in the `Math` class to calculate 2^n .
 - Write a method, `naturalNumbers`. It should verify that the elements next to the end of each row except the first, when taken in sequence, are the natural numbers. For example, the 2nd element in row 1 is 1. The second element in row 2 is 2, and the second element in row 3 is 3. The same is true for the element next to the end of each row. Return `true` if the property holds; `false` otherwise.

Programming Projects

- 10.13 Write a program implementing a robot bucket brigade. The bucket brigade consists of some number of `RobotSEs` positioned on consecutive intersections. There are a number of `Thing` objects (buckets) on the same intersection as the first robot in the brigade. When the program executes, the first robot will pick up one `Thing` and move it to the next robot's intersection, put it down, and return to its original position. The next robot will then move the `Thing` one more position down the line, and so on. When the brigade is finished, all the `Things` will be at the other end of the line of robots, one intersection beyond the last robot.
- 10.14 Implement a class named `SortTest`. It asks the user for an array size, a filename, and a sorting algorithm. It then allocates an array of strings the given length and fills it by reading tokens from the file. If the file doesn't have enough tokens, close it and begin reading again from the beginning. When the array is filled, sort it using either Selection Sort or the sort method implemented in `java.util.Arrays` (an implementation of MergeSort). Use the

program to construct a graph for each algorithm comparing the number of tokens on the x axis with the time to sort on the y axis. What conclusions can you draw about the performance of the two algorithms? (*Hint*: A good source for tokens is a book such as *Moby Dick*, available from www.gutenberg.org.)

- 10.15 The user interface for graphing mathematical functions presented in Problem 7.14 is also capable of graphing polynomial functions. Polynomials have n terms added together. Each term has the form $a_i x^i$, where a_i is called the coefficient. The overall form of a polynomial is $a_n x^n + a_{n-1} x^{n-1} + \dots + a_0 x^0$. Write a class named `PolyFunc` that extends `Object` and implements `IPolynomialFunction`. Write another class, `Main`, that includes a `main` method to run the program.
- Use `PolyFunc` to graph $a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0$, using $a_4=0.5$, $a_3=-0.75$, $a_2=0.1$, $a_1=0.0$, and $a_0=-1.0$.
 - Without changing `PolyFunc` in any way, graph $a_6 x^6 + a_5 x^5 + a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0$ (You may, however, change your `main` method.) Choose your own coefficients.
- 10.16 Explore the documentation for `becker.xtras.imageTransformation`. This package provides a graphical user interface for a program to transform images by rotating, cropping, brightening, darkening, stretching, and so on. See Figure 10-26. The actual transformations are provided by a class implementing the `ITransformations` interface.



(figure 10-26)

Image transformation graphical user interface

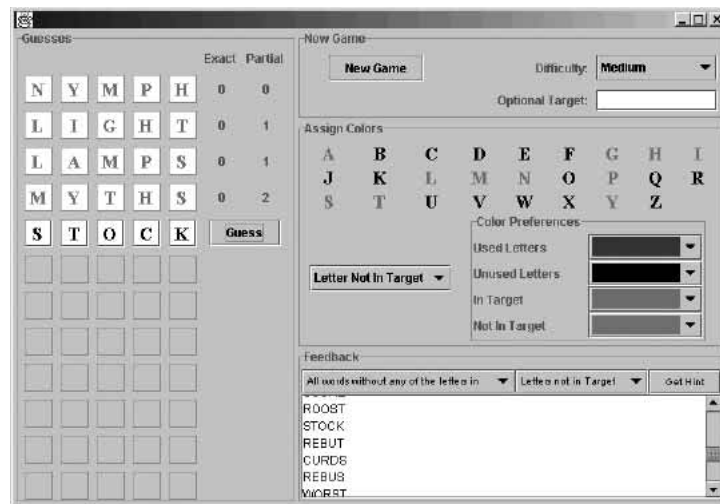
Write a class named `Transformer` that implements `ITransformations` and provides a `reset` function to reset the image to the original image that was provided as a parameter to `setPixels`. (*Hint*: Assigning references will not be enough. You need to actually copy the array.) Add code to implement the following transformations:

- “Darken” divides the intensity of each pixel by two.
- “Brighten” multiplies the intensity of each pixel by two; pixels that have a resulting value larger than 255 are set to 255.
- “Invert” makes the light pixels dark and the dark pixels light.

- d. “FlipX” turns the picture upside down.
 - e. “FlipY” reverses the left and right sides of the image.
 - f. “FlipDiag” reverses the lower left and upper right corners.
 - g. “Rotate” turns the image $\frac{1}{4}$ turn to the left (be careful that you don’t inadvertently implement “FlipDiag”).
 - h. “Scale50” removes every other row and every other column from the image, making the result .25 times the size of the original.
 - i. “Mirror” makes an image that is twice as wide as the original image, where the left half contains the original and the right side contains a mirror image.
 - j. “Blur” sets each pixel to the average of its neighbors.
- 10.17 Explore the documentation for the package `becker.xtras.jotto`. A graphical user interface, as shown in Figure 10-27, is provided in the package.

(figure 10-27)

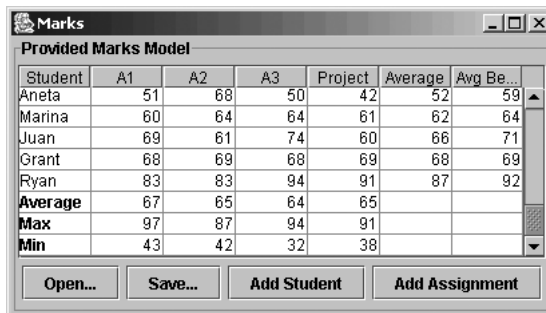
Jotto’s graphical user interface



- a. Write a `main` method, as described in the package overview, so that you can play a game of Jotto using the supplied `SampleWordList` and `SampleGuessEvaluator` classes together with the supplied user interface.
- b. Write a class named `WordList` that implements the interface `IWordList`. Modify your `main` method to run the program using your new class. Implement it using a completely filled array.
- c. Write a class named `WordList` that implements the interface `IWordList`. Modify your `main` method to run the program using your new class. Implement it using a partially filled array that includes an `addWord` method which enlarges the array as required.

- d. Write a class named `HintContainsLetter` that extends `Hint` and contains the code shown in the documentation for the `Hint` class. Modify your `main` method so you can play the game and use your new hint mechanism.
- e. Write a class named `HintExcludesLetter`. It will be similar to the class written in part (d) except that `isOK` will return true when the specified word does *not* contain the given character.
- f. Write a class named `HintContainsLetters`. It will extend `Hint` and its `isOK` method will return true if the specified word contains all of the letters obtained with the `getLetters` method in the `IHintData` object passed as a parameter.
- g. Write a class named `HintExcludesLetters`. It will extend `Hint` and its `isOK` method will return true if the specified word does not contain any of the letters obtained with the `getLetters` method in the `IHintData` object passed as a parameter.
- h. Write a class named `HintContains3Letters`. It will extend `Hint` and its `isOK` method will return true when the specified word contains at least 3 of the letters obtained with the `getLetters` method in the `IHintData` object passed as a parameter.
- i. Generalize the class described in part (h) so that the number of letters can be specified when the object is constructed. Name the class `HintContainsNLetters`.

10.18 Explore the documentation for the package `becker.xtras.marks`. Write a class named `Marks` that implements the interface `IMarks`. Write another class named `Main` that contains a `main` method as shown in the documentation. The result should appear similar to Figure 10-28.



Student	A1	A2	A3	Project	Average	Avg Be...
Aneta	51	68	50	42	52	59
Marina	60	64	64	61	62	64
Juan	69	61	74	60	66	71
Grant	68	69	68	69	68	69
Ryan	83	83	94	91	87	92
Average	67	65	64	65		
Max	97	87	94	91		
Min	43	42	32	38		

(figure 10-28)

Graphical user interface for a spreadsheet storing marks or grades

10.19 Consider Table 10-1. It gives distances between pairs of cities, similar to the charts found in some road atlases. Write a class, `Distances`, that has an instance variable referring to a 2D array storing the distances. Initialize the array from a file.

(table 10-1)

Distances in kilometers
between cities in southern
Ontario

	Kitchener	London	Stratford	Toronto
Kitchener	0	110	45	107
London	110	0	61	194
Stratford	45	61	0	149
Toronto	107	194	149	0

Add the following methods:

- `displayFarthestPair` finds and prints the pair of cities that is farthest apart.
- `displayClosestPair` finds and prints the pair of distinct cities that are closest together.
- `isSymmetrical` verifies that the table is symmetrical; that is, it returns `true` if the distance from X to Y is the same as from Y to X for each pair of cities, and if the distance from X to X is 0.
- `getDistance` returns the distance between two cities, given their names. (*Hint:* You'll need to add a 1D array of `Strings` to store the city names. Finding "Stratford" at index *i* indicates that *i* should be used as the index in the row or column of the 2D array of distances. You may need to adjust the format of your input file to include the city names.)
- `getTripDistance` returns the total distance for a trip when given an array of city names. The order of the names in the array corresponds to the order the cities are visited on the trip.

10.20 Notice that less than half of the data in the distance chart shown in Table 10-1 is actually needed. The upper half of the chart isn't needed because the array is symmetrical. Write a program that reads data from a file such as

```
4
110
45    61
107  194    149
```

where the first line gives the number of cities and the remaining lines give the distances between cities X and Y where the index of city X is less than the index of city Y. Note that this data corresponds to the lower left corner of Table 10-1.

- Write a constructor that reads this data but constructs a full 2D array, the same as Problem 10.19.
- Write a constructor that reads this data into a 2D array where each row is only long enough to store the required data.
- Add methods that perform the same calculations as a, b, d, and e in Problem 10.19.

