18. Consider this code that puts a Location into a bag, changes the name, and then tests whether the original name is in the bag. The Location class is from Figure 2.5 on page 67. What does the code print?

```
ArrayBag<Location> points = new ArrayBag<Location>( );
Location origin = new Location(0,0);
Location moving = new Location(0,0);
points.add(moving);
moving.shift(5,10);
System.out.println(points.countOccurrences(origin));
System.out.println(points.countOccurrences(moving));
```

5.4 GENERIC NODES

Nodes That Contain Object Data

Our node class from Chapter 4 can also be converted to a generic class. In the conversion, the new class, called Node, allows us to build linked lists of nodes in which the type of data in each node is determined by the generic type parameter. Here is a comparison of the original IntNode declaration with our new generic class:

Original IntNode:

```
public class IntNode
{
    private int data;
    IntNode link;

    || The methods use
    || the int data.
}
```

Generic Node Class:

```
public class <E> Node
{
    private E data;
    Node<E> link;

    || The methods use
    || the E data.
}
```

With the new Node class, each node contains a piece of data that is a reference to an E object, but we don't specify exactly what E is. The implementation of the methods is based on the IntNode methods, following the same steps that we used before on page 258. The resulting Node class is given in Appendix E. You may find yourself using it beyond this book, too.

PITFALL

MISUSE OF THE EQUALS METHOD

When you convert a collection class to contain objects, it is tempting to blindly change every occurrence of the "==" operator to use the equals method instead. There are two pitfalls to beware of.

First, beware of null references. You cannot activate the equals method of a null reference. For example, here is the implementation of the new listSearch method from the generic Node class. The method searches the linked list for an occurrence of a particular target, and returns a reference to the node that contains the target. If no such node is found, then the null reference is returned:


```
public static <E> Node<E> listSearch(Node<E> head, E target)
{
   Node<E> cursor;

   if (target == null)
   {  // Search for a node in which the data is the null reference.
      for (cursor = head; cursor != null; cursor = cursor.link)
         if (cursor.data == null)
            return cursor;
   }
   else
   {  // Search for a node that contains the non-null target.
      for (cursor = head; cursor != null; cursor = cursor.link)
         if (target.equals(cursor.data))
            return cursor;
   }

   return null;
}
```

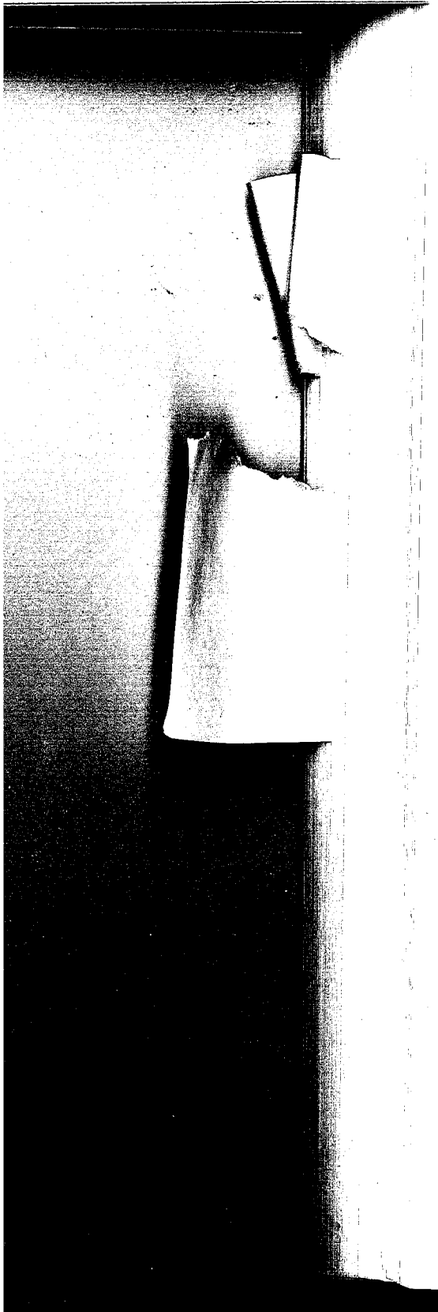
The target may be the null reference—in that case, we're searching for a node in which the data is null. This search is carried out in the first part of the large if-statement. We test whether a particular node contains null data with the boolean expression `cursor.data == null`. On the other hand, for a non-null target, the search is carried out in the else-part of the large if-statement. We test whether a particular node contains a non-null target with the boolean expression `target.equals(cursor.data)`.

In general, the equals method may be used only when you know that the target is non-null.

The second thing to beware of: You should change an equality test "==" to the equals method only where the program is comparing data. Don't change other comparisons. For example, here is the listPart method of the generic Node class:

```
public static <E> Node<E>[ ] listPart(Node<E> start, Node<E> end)
{
   Node<E> copyHead;
   Node<E> copyTail;
   Node<E> cursor;
   Node<E>[ ] answer = new (Node<E>[]) Object[2];

   // Check for illegal null at start or end.
   if (start == null)
      throw new IllegalArgumentException("start is null.");
```



```
    if (end == null)
       throw new IllegalArgumentException("end is null.");

    // Make the first node for the newly created list.
    copyHead = new Node<E>(start.data, null);
    copyTail = copyHead;
    cursor = start;

    // Make the rest of the nodes for the newly created list.
    while (cursor != end)
    {
       cursor = cursor.link;
       if (cursor == null)
          throw new IllegalArgumentException
          ("end node was not found on the list.");
       copyTail.addNodeAfter(cursor.data);
       copyTail = copyTail.link;
    }

    // Return the head and tail references
    answer[0] = copyHead;
    answer[1] = copyTail;
    return answer;
}
```

The method copies part of a linked list, extending from the specified `start` node to the specified end node. The large while-loop does most of the work, and this loop continues while `cursor != end`. This boolean expression is checking whether `cursor` refers to a different node than end. The expression becomes `false` when `cursor` refers to the exact same node that end refers to, and at that point the loop ends. Do not change this expression to use the `equals` method.

When the purpose of a boolean expression is to test whether or not two references refer to the exact same object, then use the "==" or "!=" operator. Do not use the `equals` method.

Other Collections That Use Linked Lists

Using the generic `Node` class from Appendix E, we can implement other collections that use linked lists. For example, we can implement another bag of objects that stores its objects on a linked list. In fact, we'll do exactly this in Section 5.6, but first you need to see a Java feature called *interfaces* that supports generic programming.

Self-Test Exercises for Section 5.4

19. Why was extra code added to the new `listSearch` method?

20. Suppose the while-loop in `listPart` was changed to:

    ```
    while (!cursor.equals(end))
    ```

    Give an example in which this incorrect `listPart` method would not copy all of the nodes that it is supposed to copy.

21. Suppose `x` and `y` are non-null references to two nodes. The data in each node is a non-null `Object`. Write two boolean expressions: (1) an expression that is `true` if `x` and `y` refer to exactly the same node; and (2) an expression that is `true` if the data from the `x` node is equal to the data from the `y` node. Use the `equals` method where appropriate.

5.5 INTERFACES AND ITERATORS

Java provides another feature to support generic programming: *interfaces*, which allow methods to work with objects of a class when only a limited amount of information is known about that class. This section shows you how to use some basic interfaces that are provided as part of Java's Application Programmer's Interface (API). We concentrate on the `Iterator` interface, which is particularly important for data structures programming.

Programmers can also create new interfaces (that aren't part of the API), but we won't cover that topic here.

Interfaces

interface: a list of related methods for a class to implement

A Java **interface** is primarily a list of related methods that a programmer may want to implement in a single class. For example, Java's `AudioClip` interface indicates that a class that implements the `AudioClip` interface will have three methods with these headings:

```
public void loop( )
public void play( )
public void stop( )
```

advantages of implementing a known interface

The `AudioClip` interface is intended for classes in which each object can produce a sound on a computer with sound capabilities. For example, the `play` method is supposed to play the sound one time. A class that has these three methods (and perhaps other methods, too) is said to **implement the `AudioClip` interface**. If you're writing a class with these kinds of audio capabilities, then implementing the `AudioClip` interface will help other programmers understand your intentions and use your class in a way that is consistent with other classes that provide the same capabilities. In addition, using interfaces supports generic programming because programs can be written that use `AudioClip` variables, and these program will work with any possible implementation of the interface.

How to Write a Class That Implements an Interface

Let's look at the steps to implement a Java interface. To illustrate each step, we'll outline a class called `MP3Player`, which implements the `AudioClip` interface. The `MP3Player` class is designed to allow a program to open and play a file that is stored in the popular MP3 music file format, The implementation follows these steps:

1. **Read the documentation that is provided for the interface.** For Java 5.0, this documentation is part of the Application Programmer's Interface (API) at `http://java.sun.com/j2se/1.5.0/docs/api/index.html`. Among other things, this web page lists all of the Java 5.0 classes and interfaces in a menu on the left side of the page. Click on the AudioClip interface, and you'll see a list that describes the intended use of the three methods that are part of this interface (`loop`, `play`, and `stop`).

2. **Tell the compiler that you are implementing an interface.** When you write a class to implement an interface, the keyword `implements` is used in the class head to inform the compiler of your intent. For example, we are implementing the `MP3Player` class, which will implement the `AudioClip` interface. The required syntax for our example is:

   ```
   public class MP3Player implements Audioclip
   ```

   If a class implements several different interfaces, then the names of the interfaces appear one after another, separated by commas. For example, here is the beginning of a class that implements both the `Cloneable` and `AudioClip` interfaces:

   ```
   public class MP3Player implements Cloneable, AudioClip
   ```

3. **Implement the class in the usual way.** Make sure that you implement each of the specified methods. You may also include additional methods and other instance variables if needed. In the case of the `MP3Player`, you might have a constructor that can creates an `MP3Player` object and connects it with an MP3 music file on your hard drive.

Generic Interfaces and the Iterable Interface

generic interface: a list of related methods that depend on one or more unspecified classes

Just like an ordinary interface, a **generic interface** specifies a list of methods, but these methods *depend on one or more unspecified classes*. For example, Java 5.0 defines a generic interface called `Iterator<E>` that depends on an unspecified class called `E`.

Any class that implements the `Iterator<E>` generic interface must be a generic class with its own generic type parameter, `E`. The class must provide methods with these headings:


```
public boolean hasNext( )
public E next( )
public void remove( )
```

As with any generic class, we could have used some name other than `E`, but it's common practice to use the name `E` for "element."

Intuitively, an iterator is able to step through a sequence of elements, each of which is an `E` object. Usually, the sequence comes from a collection such as a generic bag. A program activates `hasNext( )` to determine whether there are any more elements in the sequence. If there are more elements, then `hasNext( )` returns true, and the program can then activate `next( )` to obtain the next element.

An iterator also has a `remove` method. This method removes the element that was given by the most recent call to `next( )`. Sometimes an `Iterator` does not allow elements to be removed. In this case, activating the `remove` method results in an `UnsupportedOperationException`.

We could write many classes that implement the `Iterator` interface. We're going to write one particular generic class called `Lister`. When a `Lister` is constructed, it is given a reference to the head of a generic linked list of objects. The `Lister` stores a copy of this reference variable in its own instance variable called `current`. Each subsequent activation of `next` returns one element from the linked list and moves `current` on to the next element. When the last element has been returned, `hasNext` will return `false`. The `Lister` does not allow removal of elements, so any attempt to activate `remove` results in an exception.

The specification and implementation of the `Lister` class is shown in Figure 5.3.

How to Write a Generic Class That Implements a Generic Interface

A generic class can implement a generic inferface by following the same three steps that we've seen for an ordinary interface:

1. **Read the interface's documentation.** For the Iterator generic interface, this documentation is at the Iterator link of `http://java.sun.com/j2se/1.5.0/docs/api/index.html`

2. **Tell the compiler that you are implementing a generic interface.** For our gereric class, `Lister`, we write:

   ```
   public class <E> Lister implements Iterator<E>
   ```

3. **Implement the class in the usual way.** We'll discuss some of the `Lister` implementation details on page page 280.

FIGURE 5.3 Specification and Implementation for the `Lister` Class

Generic Class Lister

❖ **public class Lister<E> from the package edu.colorado.nodes**

A `Lister<E>` implements Java's `Iterator<E>` generic interface for a linked list of objects of type `E`. Note that this implementation does not support the `remove` method. Any activation of `remove` results in an `UnsupportedOperationException`.

Specification

♦ **Constructor for the Lister<E>**

`public Lister(Node<E> head)`

Initialize a `Lister` with a particular linked list of objects.

Parameters:

`head` – a head reference for a linked list of objects

Postcondition:

Subsequent activations of `next` will return the elements from this linked list, one after another. If the linked list changes in any way before all the elements have been returned, then the subsequent behavior of this `Lister` is unspecified.

♦ **hasNext**

`public boolean hasNext( )`

Determine whether there are any more elements in this `Lister`.

Returns:

`true` if there are more elements in this `Lister`; `false` otherwise.

♦ **next**

`public E next( )`

Retrieve the next element of this `Lister`.

Precondition:

`hasNext( )`

Returns:

The return value is the next element of this `Lister`. Note that each element is returned only once, and then the `Lister` automatically advances to the next element.

Throws: `NoSuchElementException`:

Indicates that `hasNext( )` is `false`.

♦ **next**

`public E next( )`

Although remove is part of the Iterator interface, it is not provided in this implementation.

Throws: `UnsupportedOperationException`:

This exception is always thrown!

(continued)

(FIGURE 5.3 continued)

Implementation

```
// File: Lister.java from the package edu.colorado.nodes
// Documentation is available at the top of this page or from the Lister link in
//   http://www.cs.colorado.edu/~main/docs/package edu.colorado.nodes;

import java.util.Iterator;
import java.util.NoSuchElementException;
import edu.colorado.nodes.Node;

public class Lister<E> implements Iterator<E>
{
   // Invariant of the Lister class:
   //   The instance variable current is the head reference for the linked list that contains
   //   the elements that have not yet been provided by the next method. If there
   //   are no more elements to provide, then current is the null reference.
   private Node<E> current;

   public Lister(Node<E> head)
   {
      current = head;
   }

   public boolean hasNext( )
   {
      return (current != null);
   }

   public E next( )
   {
      E answer;

      if (!hasNext( ))
         throw new NoSuchElementException("The Lister is empty.");

      answer = current.getData( );
      current = current.getLink( );

      return answer;
   }

   public void remove( )
   {
      throw new UnsupportedOperationException("Lister has no remove method.");
   }
}
```

The Lister Class

We implement the `Lister` class as part of our package `edu.colorado.nodes`. It is a generic class that depends on an unspecified type, `E`, which is the type of element in a generic linked list. The `Iterator` interface is part of `java.util`, so the `Lister` implementation starts by importing `java.util.Iterator`. Also, the `next` method throws a `NoSuchElementException` to indicate that it has run out of elements, so we also import `java.util.NoSuchElementException`. The `remove` method also throws an exception but no import statement is needed because `UnsupportedOperationException` is part of `java.lang` (which is automatically imported for any program).

Any program that creates linked lists can use the `Lister` class. For example, here's some code that creates a list of strings and then uses a `Lister` to step through those strings one at a time:

```
import edu.colorado.nodes.Node;
import edu.colorado.nodes.Lister;
...

Node<String> head;      // Head node of a small linked list
Node<String> middle;    // Second node of the same list
Node<String> tail;      // Tail node of the same list
Lister<String> print;   // Used to print the small linked list

// Create a small linked list.
tail   = new Node<String>("Larry", null);
middle = new Node<String>("Curly", tail);
head   = new Node<String>("Moe", middle);
// The list now has "Moe", "Curly", and "Larry". We'll print these strings.
print = new Lister<String>(head);
while (print.hasNext( ))
   System.out.println(print.next( ));
```

The while-loop steps through the elements of the list, printing the three strings:

```
Moe
Curly
Larry
```

PITFALL

DON'T CHANGE A LIST WHILE AN ITERATOR IS BEING USED

The `Lister` contains one warning in the constructor documentation: If the linked list changes in any way before all the elements have been returned, then the subsequent behavior of the `Lister` is unspecified. In other words, while the `Lister` is being used, the underlying linked list must not be altered. The reason for this is that the `Lister` uses the original linked list rather than making a copy of that list. This is the way that many of Java's built-in iterators work. However, an alternative is to make a copy of the original linked list, which we will ask you to do in a self-test exercise.

The Comparable Generic Interface

Java 5.0 has a generic interface called `Comparable<T>` that requires just one method:

```
public int compareTo(T obj)
```

This interface is intended for any class in which two objects x and y can always be compared to each other with one of three possible results:

- x is less than y
- x and y are equal to each other
- y is less than x

Many classes have this kind of natural ordering among objects. For example, two Integer objects can be compared. Therefore, Java's Integer class is implemented as:

```
public class Integer implements Comparable<Integer>...
```

Since the `Integer` class implements the `Comparable<Integer>` interface, it must have a compareTo method with this heading (where the generic type parameter has been instantiated as `Integer`):

```
public int compareTo(Integer obj)
```

FIGURE 5.4 Some Java Classes That Implement a Comparable Interface

In each case, a negative return value means that the value of x is less than the value of y; a zero return value means that the two values are equal; a positive return value means that the value of x is greater than the value of y.

Java class	Implements	What is compared by `x.compareTo(y)`
`Character`	`Comparable<Character>`	The ASCII values of x and y
`Date`	`Comparable<Date>`	A Date is a specific point in time on a specific date. When x.compareTo(y) is negative, the Date x occurred before the Date y.
`Double`	`Comparable<Double>`	The double values of x and y
`Integer`	`Comparable<Integer>`	The int values of x and y
`String`	`Comparable<String>`	The strings x and y are compared lexicographically, which means that strings of lowercase letters are compared alphabetically. When x.compareTo(y) is negative, it means that x is alphabetically before y (if they are strings of lowercase letters).

The documentation for the compareTo method is lengthy, but the gist is simple: Any two objects x and y of the class can be compared by activating x.compartTo(y). The method returns an integer which is negative (if x is less than y) or zero (if x and y are equal) or positive (if x is greater than y).

The Comparable interface uses the name T for the generic type parameter (rather than E). We could have used any name instead of T, but T is common because it refers to any "type" rather than an "element" in a collection class.

Figure 5.4 shows some of the other Java classes that implement a Comparable interface.

Parameters That Use Interfaces

Interfaces support generic programming because it is often possible to implement a method based only on knowledge about an interface. For example, we can write a method that plays an AudioClip a specified number of times:

```
public static void playRepeatedly(AudioClip clip, int n)
{  // Play clip n times.
   int i;

   for (i = 0; i < n; i++)
      clip.play( );
}
```

The data type of the actual argument for the clip parameter may be any data type that implements the AudioClip interface.

Using an Interface as the Type of a Parameter

When an interface name is used as the type of a method's parameter, the actual argument must be a data type that implements the interface.

A generic method may use a generic interface as the data type of a parameter. For example, we can write a method that computes how many of the elements in an array of T objects are less than some non-null target, provided that the data type of the target is some type that implements Comparable<T>. The method's implementation looks like this, depending on the generic type parameter T (as indicated by <T> before the return type):


```
public static <T> int smaller(T[] data, Comparable<T> target)
{  // The return value is the number of objects in the data array that
   // are less than the non-null target b (using b.compareTo to compare
   // b to each object in the data array).
   int answer = 0;

   for (T next : data)
   {
      if (b.compareTo(next) > 0)
      {  // b is greater than the next element of data.
         answer++
      }
   }
   return answer;
}
```

In this example, the argument for data could be a String array and target could be a String (since the String class implements Comparable<String>). Or data could be an Integer array and target could be an Integer. Or data could be any array of T objects, so long as the type of target implements Comparable<T>.

In each of these examples, we used an interface as the type of a parameter. This results in restrictions on what type of argument can be used with the method. In Chapter 13, we'll see how to formulate other, more powerful, restrictions.

Using instanceof to Test Whether a Class Implements an Interface

A programmer can test whether a given object actually does implement a specified interface. For example, suppose you are writing a method to test whether an object called info implements the AudioClip interface, and if so, then the play method is activated. Otherwise, some other technique is used to display some information about the object. The method could begin like this:

```
if (information instanceof AudioClip)
   information.play( );
else
   ...
```

The test is carried out with the instanceof operator, with the general form:

variable instanceof interface-or-class-name

The test is true if the variable on the left is a reference to an object of the specified type. Notice that the type name can be an ordinary class name (such as (information instanceof String)), or it may be the name of an interface (such as (information instanceof AudioClip)).

You can also test whether an object implements a generic interface. You can test for a specific instantiation of the generic interface, such as this:

```
if ((example instanceof Iterator<String>) ...
```

For example, the boolean expression shown above will be true if example is a Lister<String> object, since the Lister<String> class implements the Iterator<String> interface.

The Cloneable Interface

Throughout the book, all of our collection classes have implemented the Cloneable interface, which has a somewhat peculiar meaning. You might think that the Cloneable interface specifies that the class must implement a clone method, but this is wrong; there are no methods specified in the Cloneable interface. So what's the purpose of the Cloneable interface? The purpose comes from the behavior of the clone( ) method of Java's Object class. That clone method carries out these two steps:

1. Check to see whether the class has implemented the Cloneable interface. If not, a CloneNotSupportedException is thrown.
2. Otherwise, the clone is made by copying each of the instance variables of the original.

As you can see, the Object clone method checks to see whether the object has implemented the Cloneable interface. It does this by using the boolean test (obj instanceof Cloneable), and in fact, the only real purpose of implementing the Cloneable interface is so that the Object clone method can test (obj instanceof Cloneable). Therefore, if you write a clone method of your own and that clone method activates super.clone from Java's Object type, then your class must implement Cloneable to avoid a CloneNotSupportedException.

Self-Test Exercises for Section 5.5

22. Find the documentation for Java's CharacterSequence interface in the API. What methods does this interface require?
23. Write a generic method with one parameter that is a head reference for a linked list of nodes. The method looks through all the nodes of the list and returns a count of the number of nodes that contain null data. Use a Lister object to search through the linked list.
24. Write a generic method with one parameter that is a head reference for a linked list of nodes and a second parameter x that is a Comparable<T> object. The precondition of the method requires that x is non-null. The method returns a reference to the data in the first node that it finds in which the data is greater than or equal to x. If there is no such node, then the method returns null.
25. Re-implement the Lister constructor so that a copy of the original linked list is used rather than using the original linked list directly.
26. Write a boolean expression that will be true if and only if the data type of a variable x implements the Comparable<String> interface.
27. How many methods are required for a class that implements Cloneable?
28. Write a class, ArrayIterator, which implements the Iterator<E> interface. The constructor has one argument, which is an array of E objects. The ArrayIterator returns the components of the array one at a time through its next method.