5 Generics, Collections, and Testing

Objectives

- An introduction to the use of the Java generic construct
- An introduction to the Java Collections Framework, including the Java `LinkedList` data structure
- An introduction to the use of the Java `iterator` construct
- Implementation of a user-constructed Collection class
- Testing using `JUnit`

Key Terms

collection	foreach	iterator
elements	generics	JCF

Introduction

We claimed at the outset that this was a text on computing using Java as the vehicle, not a text on programming in Java. This chapter will come closest to violating that principle because we will cover three specifics of Java that aid in the writing of correct and understandable programs. By using generics, one can write general-purpose code that operates on more than one underlying data payload type. By building collections, one can create code packages (such as code for a linked list) that can be used as an extension to the base language by other programmers. And by creating test suites with `JUnit`, one can provide standard modules for testing code both now and after future modifications.

5.1 Using Generics for the Data Payload

In our code fragments in Figures 4.3 and 4.4 the use of the `Record` class was pervasive; this class was the data payload carried by a `DLLNode`. However, in this code we did something that is contrary to our general mantra that details of data should be confined to the code that has a clear need to know about details. Namely, we included in our DLLNode a specific textual reference to the Record class. However, the linked list data structure should concern itself not with the *nature* of the data payload but only with the *existence* of a payload of some definable data type.

Think of the cases in which this is true. The DLLNode code refers in a few places to the instance of the Record class that is the payload for that particular node. It does not, however, actually do anything with the *contents* of the instance of Record. Similarly, the DLL code makes no reference to the contents of a Record; it, too, handles the instance of Record simply as a pass-through entity.

Consider also the inner loop code of the bubblesort in Figure 3.10. There is a compareTo method that is called to determine which of two instances of the data is "smaller," but otherwise the code is moving entire instances of the data class around.

Java provides through its generic feature the ability to write code to manipulate entire instances of a class without specifying what that class is. For example, what one really wants to write as a swap method for the bubblesort (or any other sort, for that matter) is

```
void swap(ArrayList<T> list, int sub1, int sub2)
{
  T temp;
  temp = list.get(sub2);
  list.set(sub2, this.get(sub1));
  list.set(sub1, temp);
}
```

where T is replaced by whatever happens to be the data type/class name of the moment. In fact, this is exactly what we write. The T is taken by the compiler to be a generic data type, and only if someone were to invoke this method from a code fragment like

```
ArrayList<String> myList;
...
swap(myList, i, j);
```

would the compiler then finish the job by creating a swap method to swap instances of String data.

We can rewrite the simplistic code in Figures 4.3 and 4.4 to make use of the generic construct of Java. We have already used generics in this text, probably without knowing that this is what we are doing—the ArrayList is defined with a generic data type that is instanced, for example, when we define an `ArrayList` of `Record` type with the line

```
ArrayList<Record> myList;
```

The basic syntax for using generics is exactly the syntax we have been using for an `Arraylist`. The code that would *define* the use of a generic would look something like `Arraylist<T>`, with the `T` being essentially a symbolic reference to a data type. When we use an `Arraylist`, we have to supply an actual data type as in the preceding `Arraylist` declaration.

Our rewrite of Figures 4.3 and 4.4 to define classes `DLL<T>` and `DLLNode<T>` using generics appears in part in Figures 5.1–5.5.

```
public class DLL<T extends Comparable<T>>
{
  private int size;
  private DLLNode<T> head;
  private DLLNode<T> tail;

  public DLL()
  {
    this.head = new DLLNode<T>();
    this.tail = new DLLNode<T>();
    head.setNext(this.tail);
    tail.setPrev(this.head);
    this.setSize(2);
  }
  private DLLNode<T> getHead()
  {
    return this.head;
  }
  private void setHead(DLLNode<T> value)
  {
    this.head = value;
  }
    CODE FOR THE SIZE VARIABLE HERE ...
  private DLLNode<T> getTail()
  {
    return this.tail;
  }
  private void setTail(DLLNode<T> value)
  {
    this.tail = value;
  }
    MORE CODE ...
}
```

FIGURE 5.1 ■ Code fragment for a doubly-linked list using generics, part 1.

```
private void linkAfter(DLLNode<T> baseNode, DLLNode<T> newNode)
{
  newNode.setNext(baseNode.getNext());
  newNode.setPrev(baseNode);
  baseNode.getNext().setPrev(newNode);
  baseNode.setNext(newNode);
  this.incSize();
}
private void unlink (DLLNode<T> node)
{
  node.getNext().setPrev(node.getPrev());
  node.getPrev().setNext(node.getNext());
  node.setNext(null);
  node.setPrev(null);
  this.decSize();
}
```

FIGURE 5.2 ■ Code fragment for a doubly-linked list using generics, part 2.

We note that the only change needed in the code to use these classes is that the `dll` instance variable that is the linked list must be declared and constructed with `DLL<Record>` in a manner entirely analogous to the `ArrayList<Record>` mentioned previously.

The use of the generic for the type of the data payload permits the linked list class and the node class to include a payload of a type that is unspecified (one might almost say "generic") until the point at which the code is compiled and executed. Since neither the linked list class nor the node class actually *do* anything with the payload other than pass it forward and backward and move it around as an entire unit, these two classes need not know anything about what that payload is. This permits these two classes to be written once and for all (just as the `ArrayList` class was) without having to specify much of anything about the nature of the payload. You can almost view the `T` of the generic as a variable name that is passed to the compiler; unless and until the compiler actually needs to do something that depends on the value of the variable (the three things that we will deal with here are comparison, input, and output), the compiler does not need to have that variable given a value. Things like copying are legal for any variable, so code for copying instances of a variable is legal without knowing what the value of the variable happens to be.

A close comparison of the code in Figures 4.4 and 5.1 shows that essentially nothing has changed except a slight bit of syntax. The major changes, however, come from the fact that the code for `DLL<T>` and `DLLNode<T>`, since they no longer explicitly reference the `Record` class, can now no longer make use of the knowledge of the specific methods that are implemented for the `Record` class; the compiler

```
/*********************************************************************
 * Method to find if a list has a given data item.
 * @param dllData the <code>T</code> to match against.
 * @return the <code>boolean</code> answer to the question.
**/
  public boolean contains(T dllData)
  {
    boolean returnValue = false;
    DLLNode<T> foundNode = null;
    foundNode = this.containsNode(dllData);
    if(null != foundNode)
    {
      returnValue = true;
    }
    return returnValue;
  }

/*********************************************************************
 * Method to remove a node with a given record as data.
 * @param dllData the <code>T</code> to match against.
 * @return the <code>boolean</code> as to whether the record was
 *         found and removed or not.
**/
  public boolean remove(T dllData)
  {
    boolean returnValue = false;
    DLLNode<T> foundNode = null;
    foundNode = this.containsNode(dllData);
    if(null != foundNode)
    {
      this.unlink(foundNode);
      returnValue = true;
    }
    return returnValue;
  }
```

FIGURE 5.3 ■ Code fragment for a doubly-linked list using generics, part 3.

will not accept the use of methods that it cannot guarantee will exist for *every* type that might possibly be passed as T to DLL<T> and DLLNode<T>.

Consider that one of our basic operations is to search the Phonebook for an entry, that is, to use a contains method. Such a method was achieved in our early version of the code by walking through the array and using a compareName method to compare the name variable of a target entry against the name variable of the entries in the list. Almost no data types, however, will possess a compareName method, and yet the compiler, were it to see a reference to nodeData.compareName(*), would

```
/*********************************************************************
 * Method to return the node with a given data item in it, else null.
 * This method eliminates duplicate code in <code>contains</code>
 * and <code>remove</code>.
 * @param dllData the <code>T</code> to match against.
 * @return the <code>DLLNode</code> answer, else null.
**/
  public DLLNode<T> containsNode(T dllData)
  {
    DLLNode<T> returnValue = null;
    DLLNode<T> currentNode = null;

    currentNode = this.getHead();
    currentNode = currentNode.getNext();
    while(currentNode != this.getTail())
    {
      if(0 == currentNode.getNodeData().compareTo(dllData))
      {
        returnValue = currentNode;
        break; // we violate the style rule against 'break'
      }
      currentNode = currentNode.getNext();
    }

    return returnValue;
  }
```

FIGURE 5.4 ■ Code fragment for a doubly-linked list using generics, part 4.

accept this as legal only if it could be assured that all the generic data types `T` used in the linked list classes were to be guaranteed to possess a `compareName` method.

We cannot guarantee to the compiler that all data types used for `T` will possess all possible methods. There is a way, however, to guarantee that some obviously useful or otherwise necessary methods will be supplied. The

```
T extends Comparable
```

in the declaration

```
public class DLL<T extends Comparable>
```

in Figure 5.1 is our promise to the compiler that any data type used for `T` will have implemented the `compareTo` method of the `Comparable` interface.[1] Having

[1] In general, `extends` is a guarantee that the data type will implement *all* the methods required by the interface, but in this case the `Comparable` interface requires only the one method `compareTo`.

```
public class DLLNode<T>
{
  private DLLNode<T> next;
  private DLLNode<T> prev;
  private T nodeData;
  public DLLNode()
  {
    super();
    this.setNext(null);
    this.setPrev(null);
    this.setNodeData(null);
  }
  public DLLNode(T data)
  {
    super();
    this.setNext(null);
    this.setPrev(null);
    this.setNodeData(data);
  }
  public T getNodeData()
  {
    return this.nodeData;
  }
  public void setNodeData(T newData)
  {
    this.nodeData = newData;
  }
  public DLLNode<T> getNext()
  {
    return this.next;
  }
  public void setNext(DLLNode<T> newNext)
  {
    this.next = newNext;
  }
  public DLLNode<T> getPrev()
  {
    return this.prev;
  }
  public void setPrev(DLLNode<T> newPrev)
  {
    this.prev = newPrev;
  }
}
```

FIGURE 5.5 ■ Code fragment for a node in a doubly-linked list using generics.

guaranteed that compareTo will exist, the compiler permits us to write without error the line

```
    if(0 == currentNode.getNodeData().compareTo(dllData))
```

because the call to currentNode.getNodeData() returns an instance of type T, and we have instructed the compiler that T will extend Comparable and thus will have a legitimate compareTo method implemented for it.

The T extends Comparable is a contract we have made with the compiler. The compiler promises to hold up its end of the contract by accepting the use of a compareTo method. We in turn must fulfill our part of the bargain by ensuring that that method exists. To do this, we change the declaration of the Record class to read

```
    public class Record implements Comparable<Record>
```

and we rename the old compareName method compareTo because that's the more general method name that Comparable uses and that the compiler is now expecting to see. The compiler will now be happy, sure in its knowledge that whatever data type is used for T, the comparison we have asked for will be syntactically correct. (Of course, if we write the wrong code for the compareTo method, it could still produce the wrong *results*, but the compiler isn't going to worry about that.) The two classes for DLL<T> and DLLNode<T> can be written without saying anything in detail about what kind of data payload to expect except for the fact that payloads can be compared.

Now that we have guaranteed that we can compare payloads, we can also do a search for a specific payload, and we can therefore implement the contains and remove methods in Figure 5.3. In implementing a contains method, unlike previous methods for this application, we have to consider which values ought to be returned to the calling program. We are shortly going to move from code that is entirely under our control to the implementation of code that must follow the strictures laid down by standard classes in Java, so we will take the opportunity to ensure that what we do now will not have to be changed later. For this reason, we will implement the add, contains, and remove methods to accept a Record (actually an item of type T) as the input parameter and to return a boolean value indicating whether the operation was successful. The use of a boolean both for the answer to the contains question and to indicate that a remove operation actually did find and remove an entry is obvious. Less obvious is the use of a boolean response to an add method. However, in the Java collection classes there are classes for such things as the equivalent of a mathematical set. Adding an entry that is already present does not change the contents of a set, and thus the result of adding a duplicate entry should be false to indicate that no change occurred to the data structure implementing the set. For the sake of consistency with the rest of the Collections framework, we will implement add as a method that returns a boolean. The value true is returned if and only if the underlying structure has changed as a result of the add. In the case of the current class, this will always be true.

The contains and remove methods in Figure 5.3 take as parameters an entire instance of an object of generic type T. This is often not what we want to do; we will frequently want to pass in only a key (like a last name) and have a version of a contains method respond that yes (or no), an item with that String as last name happens to be in the data (or not). We can easily write methods to do this by overriding the original contains method. As a practical matter, though, we then have to consider whether returning just a boolean is the smartest thing to do. The original contains and remove methods are unambiguous; an entire instance of an object is passed in, and the comparison is presumably done on the entire instance. In the case of the ArrayList methods, these two methods return true only if the exact object matches (and not just the values of the instance variables in the object). If we are passing in a last name, it may well happen that we have multiple records with the same last name. We probably don't want to remove any arbitrary record just because the last name matches, and we won't get all the information we need if our contains terminates a search and returns true the first time it hits a match on last name. There will, therefore, be good reasons to write methods that differ from the standard methods for adding, removing, and searching. Perhaps we would want the method to return a subscript if we knew we were dealing with a subscripted list. If we were dealing with something like a linked list, though, without formal subscripts, we might want to have our methods return the entire instance of the object.

5.2 Objects and Equality

We have said it already, but it is worth saying again, because this is the point at which it starts being a big deal: When two instances of an object are compared with the == symbol, the two instances are considered to be equal if they are absolutely identical. This means that the "two" instances must in fact be references to the same location in memory. The code

```
Record one = new Record();
Record two = new Record();
if(one == two)
{
  System.out.println("equal");
}
else
{
  System.out.println("not equal");
}
```

results in the message `not equal` because these are two `Record` instances with the same (default) contents, not the same object, and Java would only return `true` for the `==` if these were the same object. That would happen with the following code fragment because variable `two` would be assigned to be identical to variable `one`.

```
Record one = new Record();
Record two = new Record();
two = one;
if(one == two)
{
  System.out.println("equal");
}
else
{
  System.out.println("not equal");
}
```

The basic `equals` method implemented in the `Object` class works this way—it does the most narrow possible comparison to determine equality, and it requires that the two objects point to the same location in memory before it will return a value of `true`.

Probably in your earlier work in Java this was only an issue when dealing with `String` data. Although the `equals` method for an instance of an `Object` returns `true` only when the two objects are identical, the `equals` method has been overridden for data of `String` type so that the method will return `true` if the two sequences of characters are the same, that is, if the *contents* of the two `String` objects are the same. This allows us to compare two strings of characters, and it is exactly this kind of override that we will have to implement in order to use generics for an arbitrary class. Any class that we declare is a subclass of `Object` and thus inherits by default the `equals` method that is usually too strict for our purposes. We will almost always, especially when using generics, have to override that method with our own appropriate `equals`.

5.3 The Java `LinkedList`

It turns out that Java, straight out of the box, comes replete with classes and interfaces that are part of the *Java Collections Framework*, or JCF. A *collection* is nothing more than a representation for a group of objects known as the *elements* of the collection. There exists, for example, a Java `LinkedList` class that can be used instead of the code that we have presented so far in this text. The `LinkedList` class has methods for **add**, **contains**, and **remove** that function (almost) exactly as do our methods for the same functions. The LinkedList also has a great many other methods that are documented on the Java website. By including

```
import java.util.LinkedList;
```

in our Phonebook.java code, we can remove all reference to our own DLL.java and DLLNode.java classes and replace them with references to DLL from LinkedList. If we do this, the only complication is that we no longer have the hand-coded toString method for the entire linked list. This method exists in the LinkedList class, but its output is different from what we wrote ourselves. Our toString produced formatted output that used the toString that we knew existed (because we had written it) of the data payload. The built-in version of toString for the LinkedList is much cruder because it must be more general purpose; it uses the Object.toString() method that works for all objects. The documentation for Object.toString() says in part "It is recommended that all subclasses override this method," but the creators of Java obviously did not feel it was necessary to *require* that this be overridden.

To create the same output from LinkedList that we had before with our own class, we also need to use the listIterator method that comes with the LinkedList class; this requires us to include in our code

```
import java.util.ListIterator;
```

to ensure that the listIterator method is found by the compiler, and then the Phonebook.java version of toString can be written as in Figure 5.6.

```
/**********************************************************************
 * Method to <code>toString</code> a complete Phonebook.
 * @return the <code>toString</code> rep'n of the entire DLL.
**/
  public String toString()
  {
    String s = "";
    Record rec;
    ListIterator<Record> iter = this.dll.listIterator();
    while(iter.hasNext())
    {
      rec = iter.next();
      s += String.format("%s%n", rec.toString());
    }
    return s;
  }
```

FIGURE 5.6 ■ Code fragment for a toString method.

5.4 Iterators

The code presented in Figure 5.6 uses a Java feature known as an iterator. We have been using an iterator repeatedly in this course for input of data with a `Scanner`, and it is now time to look one level deeper to see what this construct really does for us and how to implement one.

An *iterator* is nothing more than a class that implements two (and sometimes three) methods for accessing the elements of a data structure. We present in Figure 5.7 the code for a `DLLIterator` class that could be implemented separately and that would provide an iterator for the `DLL` code in this chapter.[2] The interface for an `Iterator` class requires that methods `hasNext` and `next` be implemented; these perform the same function as do the methods that we have used for the `Scanner` class for input. The interface does not require that the `remove` method be implemented, so we include code for that but throw the `UnsupportedOperationException` as required.

We have in our `toString` method in Figure 5.6 used not an `Iterator` but a `ListIterator`. The `ListIterator` is intended for use with a collection (like a linked list) that has some notion of "sequential" access as opposed, say, to a collection that implements a mathematical set for which no sequential ordering of the elements is obviously appropriate. A `ListIterator` has more required methods than does an `Iterator`, but the basic concept is the same, and we use both extensively.

An iterator is also present when the Java *foreach* construct is used. Instead of the code in Figure 5.6 that resembles what we use with the `Scanner` class, we could make our linked list "iterable" and write

```
for(Record rec: this.dll)
{
  s = String.format("%s%n", rec.toString());
}
```

In either case, what we have done is to use a construct that allows us to access directly the data payload in the structure without having to provide (or even think about) subscripts.

5.4.1 The Justification for Iterators

It is conceivable at this point that the eyes of many readers are glazing over and readers are wondering whether this concept of an iterator is just something dreamed

[2]In fact, we would probably *not* want to implement this as a separate class, but the reasons for that will appear shortly. For now, it is sufficient that this code will in fact work.

```
import java.util.Iterator;
public class DLLIterator<T extends Comparable<T>> implements Iterator<T>
{
  private DLLNode<T> current;
  private DLL<T> theDLL = null;

  public DLLIterator(DLL<T> dll)
  {
    this.theDLL = dll;
    this.current = this.theDLL.getHead();
  }

  public boolean hasNext()
  {
    boolean returnValue = false;
    if(this.current.getNext() != this.theDLL.getTail())
      returnValue = true;
    return returnValue;
  }

  public T next()
  {
    T returnValue = null;
    this.current = this.current.getNext();
    returnValue = this.current.getNodeData();
    return returnValue;
  }

  public void remove()
  {
    throw new UnsupportedOperationException(
                         "'remove' not supported);
  }
}
```

FIGURE 5.7 ■ Code for an `Iterator` class for a doubly-linked list.

up by compiler writers and language designers because they thought it would be neat to have this kind of feature. Although we may agree with the readers some of the time with regard to compiler and programming language people, this time we have to admit that they got it right. Compare the following two bits of pseudocode, for example.

```
// code fragment one
node = head;
while(node != tail)
{
  Record rec = node.getRecord();
  s = String.format("%s%n", rec.toString());
  node = node.next();
}

// code fragment two
for(Record rec: this.dll)
{
  s = String.format("%s%n", rec.toString());
}
```

The big difference is this: In the first fragment, we knowingly traverse a sequence of *nodes*, fetch the data payload for each node, and then use that payload. (In both fragments all we do is create the `toString` of the payload, but this code could be replaced in both fragments by code that actually used the data.)

In the second fragment, we simply fetch the data. The code at this level in the second fragment is entirely ignorant of the underlying data structure. The data structure has been hidden from this code fragment, and all that is visible is the ability to move from one data element to the "next" data element.

This is[3] the essence of an iterator—instead of fetching the data structure element and then the data in that element, we fetch the data element directly and can ignore the implementation issues below the fetch. The `Iterator` interface requires no more than a `hasNext` and a `next` method (with an optional `remove` method).

Even if no more than the required methods are implemented, this is a powerful technique that eliminates one level of indirect reference (data payloads accessed immediately instead of data payloads accessed within linked list nodes) by focusing directly on the data payload itself. By focusing on the data itself, and removing that one level of indirection, we would hope that programs would become simpler, more likely to be correct, and easier to write.

It would be dishonest, however, to stop here and not point out one more unfortunate fact. What we gain with an iterator, in the ability to reference a data item and then "the next" data item, we must also pay for at the time we need to remove an item or move it around. If we manipulate a sequence of data items with the first code fragment, and what we really want to do is to remove the item from

[3]at least at the level of this course

the linked list, then we have the actual node to be unlinked, which means that we have access to the `next` and `previous` nodes and can do the appropriate unlinking and relinking of nodes. If we have gone beyond the concept of nodes, to deal only with the data items themselves, then in order to unlink and relink, we will need to implement methods that revert to linked list notions. There's never any free lunch.

5.5 Implementing Our Very Own Collection

It is now time to convert our code for a doubly-linked list into the sort of code that would be expected of real Java programmers. We will not push this all the way through to a complete implementation, but we will go far enough to show what could be completed with only a SMOP,[4] and we will leave that SMOP to the student, who will no doubt benefit immensely from the experience.

First, we observe that only two classes need to be changed—the `DLL<T>` class and the `Record` class. The `DLL<T>` class must now begin

```
import java.util.AbstractSequentialList;
import java.util.List;
import java.util.ListIterator;
public class DLL<T extends Comparable<T>>
                    extends AbstractSequentialList<T>
                    implements List<T>
```

and we are now required to implement several methods in order to satisfy the requirements of `AbstractSequentialList` and `List`. We notice that the requirement for the built-in `add` method specifies that elements are to be added at the tail and not the head, so we write an `addBeforeTail` and a `linkBefore` method entirely symmetrical to the two methods we have already written, and we use these instead so as to comply with the rules for the classes and interfaces of the JCF. A brief mention of the hierarchy is in order. The ancestral notion is of a Java `Collection`, which is an interface that prepares the way for dealing with collections of data items (that is, in a way more complicated and providing more underlying support than are present with a simple data array). Three of the subinterfaces that have intuitive meaning are the `List`, `Set`, and `SortedSet`. There is a built-in class `AbstractList` that provides an abstract notion of a "list" data type, and then more specialized than that is the `AbstractSequentialList` class that implements the notion, as the

[4]Small Matter Of Programming; search online for the "Jargon File."

name implies, of a list of data items that are to be thought of in a sequential manner. Indeed, the `LinkedList` class we used in a previous section is itself a subclass of `AbstractSequentialList`.

To complete our nascent abstract sequential list, currently embodied in our linked list code `DLL`, we need only supply in `DLL` a method `listIterator`. This code is simple:

```
public ListIterator<T> listIterator(int index)
{
  ListIterator<T> iter = new DLLListIterator(this);
  return iter;
}
```

although we admit that all we have done is to pass off the work to the implementation of a `DLLListIterator` class. That class is found in Figures 5.8–5.10 as a class *embedded within* the `DLL` class.

In a footnote early in Section 5.4, we said that we would explain soon why we wouldn't really want a separate class; the time is now to explain why we have chosen to embed this class inside the `DLL` class. Our earlier class for an iterator made a copy of the linked list when an instance of the iterator was created. This is not only somewhat clumsy but also highly unsafe; to eliminate this problem, we have embedded the new iterator class inside the `DLL` class so the iterator can have direct access to some of the variables of the `DLL` class. The `unlink` method is used, for example, by the `remove` method. More importantly, we can with the embedded class get direct access to the linked list itself, so the only instance variable inside the iterator class that we need in order to maintain a sense of context with the linked list is the `current` pointer.

Finally, we have chosen in this implementation not to implement the optional methods for `add` and `set` and we have not shown the implementations of `nextIndex` and `previousIndex`. We have, however, implemented the `remove` method together with its subtleties of when it is and is not legal to be issued.

The change to the `Record` class is significant, yet subtle. We note that the `List` possesses methods

```
boolean contains(Object o)
```

and

```
boolean remove(Object o)
```

```
private class DLLListIterator implements ListIterator<T>
{
  private boolean removeInvalid_Add;
  private boolean removeInvalid_NextPrevious;
  private DLLNode<T> cursor;
  private DLLNode<T> lastReturned;
/*********************************************************************
 * Constructor.
**/
  public DLLListIterator(DLL<T> dll)
  {
    this.cursor = dll.getHead().getNext();
    this.lastReturned = null;
    this.removeInvalid_Add = false;  // we don't implement 'add' here
    this.removeInvalid_NextPrevious = true;
  }
/*********************************************************************
 * Method to answer the question of a "next" element.
 * @return the <code>boolean</code> answer to the question.
**/
@Override
  public boolean hasNext()
  {
    boolean returnValue = false;

    if(this.cursor != getTail())
      returnValue = true;

    return returnValue;
  }
/*********************************************************************
 * Method to answer the question of a "previous" element.
 * @return the <code>boolean</code> answer to the question.
**/
@Override
  public boolean hasPrevious()
  {
    boolean returnValue = false;

    if(this.cursor.getPrev() != getHead())
      returnValue = true;

    return returnValue;
  }
```

FIGURE 5.8 ■ The list iterator code, part 1.

```
/**********************************************************************
 * Method to return the "next" element.
 * @return the next data element.
**/
  public T next()
  {
    if(this.hasNext())
    {
      this.removeInvalid_NextPrevious = false;
      this.lastReturned = this.cursor;
      this.cursor = this.cursor.getNext();
    }
    else
    {
      throw new RuntimeException("ERROR: hasNext fails");
    }
    return this.lastReturned.getNodeData();
  }

/**********************************************************************
 * Method to return the "previous" element.
 * @return the previous data element.
**/
  public T previous()
  {
    if(this.hasPrevious())
    {
      this.removeInvalid_NextPrevious = false;
      this.cursor = this.cursor.getPrev();
      this.lastReturned = this.cursor;
    }
    else
    {
      throw new RuntimeException("ERROR: hasPrevious fails");
    }
    return this.lastReturned.getNodeData();
  }
```

FIGURE 5.9 ■ The list iterator code, part 2.

whose functions are exactly as the methods of the same name that we have implemented already. However, it is here that we must take special note once again of the way that Java defines the concept of "equals." Two objects are considered equal, that is,

```
this.equals(that)
```

```
/********************************************************************
 * Method to remove the element returned by the most recent "next"
 * or "previous" operation.  This is invalid if it has already been
 * called once since the last "next" or "previous" operation or if
 * "add" has been called since the last call to "next" or "previous".
 * @throws IllegalStateException if this method is invalid now.
**/
  public void remove()
  {
    String s  = "";
    if(this.removeInvalid_NextPrevious)
    {
      s  = String.format("%s","illegal call to 'remove'");
      s += String.format("%s","--already called since the ");
      s += String.format("%s","last call to 'next' or 'previous'");
      throw new IllegalStateException(s);
    }
    else if(removeInvalid_Add)
    {
      s  = String.format("%s","illegal call to 'remove'");
      s += String.format("%s","--'add' called since the ");
      s += String.format("%s","last call to 'next' or 'previous'");
      throw new IllegalStateException(s);
    }
    else
    {
      // If we got the "lastReturned" from a "next" then we
      // already bumped the "cursor".  But if we got the
      // "lastReturned" from a "previous" then we need to bump
      // the "cursor" explicitly to the next node.
      if(this.cursor == this.lastReturned)
        this.cursor = this.cursor.getNext();
      unlink(this.lastReturned);
      removeInvalid_NextPrevious = true;
    }
  }
```

FIGURE 5.10 ■ The list iterator code, part 3.

returns a `boolean true`, if and only if `this` and `that` *are the same object, unless we override the default code for* `equals`.

The `contains(Object o)` and `remove(Object o)` methods test for exact equality of objects, and these methods always work because everything in Java is a thing of `Object` type. It is almost never the case, though, that this is what we will want in a program. If we had an `ArrayList` of three strings, say `pepperoni`, `mushroom`, and `sausage` for types of pizza, and if we then read input `mushroom` from a user ordering pizza, the `contains` method will return `false` when it does the lookup on `mushroom`. The problem is that the `mushroom` instance of the object stored in the `ArrayList` is a *different* instance of the object from the `mushroom` instance provided by the user, and `contains` looks at the equality of the objects, not equality of the data contents. (For `String` data, that test is done by the `equals` method in the `String` class.)

If we want to test for equality of the data contents, we must override the default for `equals`, which is what we did in our `Record` class because we wanted to consider two records to be "equal" if the last names in the data were the same. In any search using a search key, we will want to do exactly this because we will be searching for the complete record using only the key—if we had the entire record, we wouldn't need to be searching for it!

The second subtle but important change to our `Record` is to ensure that we will in fact override the built-in method for `equals`. Java is very fussy about type-checking, and our earlier code for the `Record` class read

```
public boolean equals(Record that)
```

which is not what the JCF needs. That method has a different signature than the built-in method because the parameter is of type `Record` and not of type `Object`.

In order to use our own version of `equals` to override the default in the requirements for the `List` interface, we must change our code to read as in Figure 5.11.

```
/*********************************************************************
 * Method to override the <code>equals</code> method.
 * We will declare two records to be equal if their data values are
 * equal, not if they are identical objects.
 * NOTE THE PARAMETER TYPE.  THIS IS ESSENTIAL TO HAVING THE METHOD
 * PROPERLY OVERRIDE THE DEFAULT <code>equals</code>.
 * @param that the <code>Object</code> to be compared against.
 * @return boolean answer to the question.
**/
  public boolean equals(Object that)
  {
    boolean retVal;
    retVal = true;
    retVal &= this.getName().equals(((Record) that).getName());
    retVal &= this.getOffice().equals(((Record) that).getOffice());
    retVal &= this.getPhone().equals(((Record) that).getPhone());
    retVal &= (this.getTeaching() == ((Record) that).getTeaching());
    return retVal;
  }
```

FIGURE 5.11 ■ The `equals` node in the `Record` class.

The parameter is passed as an instance of `Object` type, but it then must be cast to an instance of `Record` type in order to invoke the correct `equals` method. Because the signature of the method is now the same as that of the default `equals` method, we will in fact be overriding the default instead of creating a new `equals` method that would be valid only for parameters of type `Record`.

5.6 Testing with JUnit

The last part of converting our code so that it resembles professional code[5] is to implement unit tests with `JUnit`. The Java `JUnit` capability is not inherent in Java, but it is relatively easy to add the `jar` for `JUnit`; for example, when using Eclipse™ it is necessary to add the external `jar` into the compiler's build path for the project.

A `JUnit` unit testing module for the `Record` class is presented in Figures 5.13–5.16. The class is a standard Java class, except that the `jar` for `JUnit` is part of the build path; we import part of that `jar` into our class; and the class extends `TestCase` in the `jar`.

We have a number of methods, each of which has a name that begins with `test` and continues with a method name from `Record`. Each of these methods will be executed after a call to `setUp`, and the `tearDown` method will be executed afterward each time, so these two methods should be written to provide each testing method a clean environment with no context held over from a previous method.

The method calls that we will use from `JUnit` are given in Figure 5.12. The syntax is almost self-explanatory. For the

```
assertEquals(messageString, expectedValue, actualValue)
```

method, for example, the `expectedValue` might be the returned parameter expected from a method called with a certain set of arguments, and the `actualValue` argument would in fact be the call to that method with those arguments. If the values are not equal, then either a default or (in this case) a user-specified `messageString` is printed along with the failure information. As with any other Java class, output to the console is possible if the user is unsure as to the progress of the testing.

In order to test the `Record` class with `JUnit`, we did have to make a few small changes. A `JUnit` test is yet another class in Java and can thus see only the `public` methods and variables. In our earlier code, for example, we had declared

[5]We use the word "resembles" because we are still not going to be done and would not go so far as to suggest that it might actually *be* professional code.

```
assertEquals(expectedValue, actualValue)
assertEquals(messageString, expectedValue, actualValue)
assertFalse(booleanCondition)
assertFalse(messageString, booleanCondition)
assertNotNull(object)
assertNotNull(messageString, object)
assertNotSame(expectedValue, actualValue)
assertNotSame(messageString, expectedValue, actualValue)
assertNull(object)
assertNull(messageString, object)
assertSame(expectedValue, actualValue)
assertSame(messageString, expectedValue, actualValue)
assertTrue(booleanCondition)
assertTrue(messageString, booleanCondition)
failNotEquals(messageString, expectedValue, actualValue)
failNotSame(messageString, expectedValue, actualValue)
```

FIGURE 5.12 ■ Assertion methods in `JUnit`.

the `DUMMYSTRING` and `DUMMYINT` variables to be both `private` and `static`. In order to use them as class variables in the `testConstructor` method in Figure 5.13, we had to change the declaration to `public`. We note that we have not explicitly tested the `getName`, `getOffice`, `getPhone`, and `getTeaching` methods. We have, however, used each of these in testing other methods, so it can be argued that these methods have been tested implicitly.

Further thought together with a look at the code for testing the `equals`, `readRecord`, and `toString` methods shows that testing is an art, not entirely a science. How much testing is enough? What should be tested in which test method? For example, in testing the `equals` method, we have not tested (once again) the fact that our constructor works correctly, but we have included lines to verify that the `setName` method worked as intended. If the test methods are ever intended to be broken apart and used separately, they should test all aspects of the code, but if the test class is intended only to be used as a single class, then we could assume, for example, that the `setName` method has been tested elsewhere and does not need further testing in the `testEquals` method. (This does, of course, lead to the possibility that the test methods will begin by being treated as a block and then some methods will be yanked out for use in another testing class. Such actions cannot be predicted, and such improper later use of code cannot be prevented. The prudent programmer,[6] though, will include in the documentation of a test method a sufficient number of caveats about what is and what is not tested so as to avoid being blamed later for a poor test method.)

[6]Those who are unfamiliar with this sort of expression should look up “prudent mariner” in a nautical context.

```
import junit.framework.*;
import java.util.Scanner;
/**********************************************************************
 *
**/
public class RecordTester extends TestCase
{
  private Record rec1, rec2;
/**********************************************************************
 *
**/
  public RecordTester(String name)
  {
    super(name);
  }
/**********************************************************************
 *
**/
  protected void setUp()
  {
    rec1 = new Record();
    rec2 = new Record();
  }
/**********************************************************************
 *
**/
  protected void tearDown()
  {
    rec1 = null;
    rec2 = null;
  }
/**********************************************************************
 *
**/
  public void testConstructor()
  {
    System.out.println("Test the constructor");
    rec1 = new Record();
    assertEquals(Record.DUMMYSTRING, rec1.getName());
    assertEquals(Record.DUMMYSTRING, rec1.getPhone());
    assertEquals(Record.DUMMYSTRING, rec1.getOffice());
    assertEquals(Record.DUMMYINT, rec1.getTeaching());
  }
```

FIGURE 5.13 ■ A JUnit testing class for Record, part 1.

```
/*********************************************************************
 *
**/
  public void testCompareTo()
  {
    System.out.println("Test compareTo");
    rec1 = new Record();
    rec2 = new Record();
    assertEquals(Record.DUMMYSTRING, rec1.getName());
    assertEquals(Record.DUMMYSTRING, rec2.getName());
    rec1.setName("duncan");
    assertEquals("Failure compareTo set", "duncan", rec1.getName());
    rec2.setName("duncan");
    assertEquals("duncan", rec2.getName());
    assertEquals("Failure compareTo equals", 0, rec1.compareTo(rec2));
    rec2.setName("aaaa");
    assertEquals("aaaa", rec2.getName());
    assertEquals("Failure compareTo greaterthan", 1, rec1.compareTo(rec2));
    rec2.setName("eeee");
    assertEquals("eeee", rec2.getName());
    assertEquals("Failure compareTo lessthan", -1, rec1.compareTo(rec2));
  }
TEST FOR compareName IS SIMILAR
/*********************************************************************
 *
**/
  public void testDefaultInstances()
  {
    assertEquals(Record.DUMMYSTRING, rec1.getName());
  }
TESTS FOR getPhone, getOffice, getTeaching ARE SIMILAR
/*********************************************************************
 *
**/
  public void testSetName()
  {
    assertEquals(Record.DUMMYSTRING, rec1.getName());
    rec1.setName("duncan");
    assertEquals("Name Failure","duncan", rec1.getName());
    assertFalse("messagefail name", rec1.getName().equals("Someone"));
  }
TESTS FOR setPhone, setOffice, setTeaching ARE SIMILAR
```

FIGURE 5.14 ■ A `JUnit` testing class for `Record`, part 2.

```
/*******************************************************************
 *
**/
  public void testEquals()
  {
    System.out.println("Test equals");
    rec1 = new Record();
    rec2 = new Record();
    assertEquals("Failure equals one", true, rec1.equals(rec2));
    rec1 = new Record();
    rec2 = new Record();
    rec1.setName("duncan");
    assertFalse(rec1.equals(rec2));
    assertEquals("duncan", rec1.getName());
    rec2.setName("duncan");
    assertEquals("duncan", rec2.getName());
    assertEquals("Failure equals two", true, rec1.equals(rec2));
    rec1.setOffice("myoffice");
    rec2.setOffice("myoffice");
    rec1.setPhone("myphone");
    rec2.setPhone("myphone");
    rec1.setTeaching(1248);
    rec2.setTeaching(1248);
    assertEquals("Failure equals three", true, rec1.equals(rec2));
  }
/*******************************************************************
 *
**/
  public void testToString()
  {
    String dummy = "";
    Scanner inFile = null;
    System.out.println("Test toString");
    inFile = FileUtils.ScannerOpen("zin");
    rec1 = new Record();
    dummy = inFile.next();
    assertEquals("add", dummy);
    rec1 = Record.readRecord(inFile);
    assertEquals("Herbert", rec1.getName());
    assertEquals("2A41", rec1.getOffice());
    assertEquals("789.0123", rec1.getPhone());
    assertEquals(390, rec1.getTeaching());
    assertEquals("Herbert    2A41  789.0123   390", rec1.toString());
    FileUtils.closeFile(inFile);
  }
```

FIGURE 5.15 ■ A JUnit testing class for Record, part 3.

```
public void testReadRecord()
{
  String dummy = "";
  Scanner inFile = null;

  System.out.println("Test readRecord");

  inFile = FileUtils.ScannerOpen("zin");
  rec1 = new Record();

  dummy = inFile.next();
  assertEquals("add", dummy);
  rec1 = Record.readRecord(inFile);
  assertEquals("Herbert", rec1.getName());
  assertEquals("2A41", rec1.getOffice());
  assertEquals("789.0123", rec1.getPhone());
  assertEquals(390, rec1.getTeaching());

  dummy = inFile.next();
  assertEquals("add", dummy);
  rec1 = Record.readRecord(inFile);
  assertEquals("Lander", rec1.getName());
  assertEquals("2A47", rec1.getOffice());
  assertEquals("789.7890", rec1.getPhone());
  assertEquals(146, rec1.getTeaching());

  dummy = inFile.next();
  assertEquals("add", dummy);
  rec1 = Record.readRecord(inFile);
  assertEquals("Winthrop", rec1.getName());
  assertEquals("3A71", rec1.getOffice());
  assertEquals("789.4667", rec1.getPhone());
  assertEquals(611, rec1.getTeaching());

  FileUtils.closeFile(inFile);
}
```

FIGURE 5.16 ■ A `JUnit` testing class for `Record`, part 4.

■ 5.7 Reflection

Contrary to popular belief and misunderstanding, this text and this course are not about "how to program." Computer science is largely about the management and organization of information. The implementation of decisions about management and organization will be made on a computer using specific programming languages, and thus the implementation will require that one know how to program. Now is a good time to step back and think about the various kinds of information floating around.

This course and its predecessor are not courses on how to program. They are about the design of algorithms, and they use the particular syntax of a programming language (in this case, Java) to force you to be crisp and precise in your thinking and your design. You should by now be familiar with the information about the computational task to be performed and about its implementation in Java. Iterating through a `for` or a `while` loop is a standard process in computer science, with a particular syntax required by Java. Storing data in an array or `ArrayList` and then iterating to swap entries that are in the wrong order, as in the inner loop of the bubblesort, is a standard process in computation, using code such as

```
for(int i = 0; i < list.size()-1; ++i)
{
  for(int j = i+1; j < list.size(); ++j)
  {
    if(list.get(i) > list.get(j))
      swap the i-th and j-th entries
  }
}
```

These are algorithmic processes that require you to manage the actual information of the computation.

What we have covered in this chapter goes one level deeper. The use of the Java generic construct requires you to think about the information used in the *compilation* process. Iterating through the two loops and swapping two entries requires only that we know about the existence of the entries in the list, not that we know anything about the essence of those two entries. It is the test, to see if we need to swap, that requires that we know about the data types to be tested so that we can know how to implement the test.

The double loop with the test as a method call and the swap can be written once for all time using the generic construct to indicate symbolically the data type with which we are dealing. Because the Java compiler will try to compile code that will always work, regardless of what the data type happens to be, you the programmer must manage the information provided to the compiler so that it can function correctly.

It is this metainformation about your computation, the information not about the computation itself but about the compiling process, that you must manage in using the generic construct and the various interfaces provided as part of Java. In order for compilation to take place, the information that is available to the compiler about the program must be internally self-consistent. To use something like the `Record` class in the linked list implemented with generics and to be able to do the lookup of a `contains` method, we must guarantee that we extend `Comparable` and provide some version of a `compareTo` method. Containment requires a test for equality. The `ArrayList` construct, built using generics, has inherited the test for equality of instances of `Object` type all the way at the top of the hierarchy, but that's a test that looks at the actual object, not the data contents. (Remember also that one cannot create an `ArrayList` of `int` variables; it must be an `ArrayList` of `Integer` variables, so that in fact each entry will be an object.)

Designing programs involves a constant tension between the need to keep everything as simple as possible but not too simple and the desire to make everything as general and abstract as possible so as not to have to write nearly identical bits of code. At some point the ability to write general-purpose code (like the loop, test, and swap of the bubblesort) bumps up against the need to handle data in specific ways (like the exact nature of the test), and much of the Java framework has been established to allow you to separate parts of the program based on what information is necessary for proper functioning of each part.

5.8 Summary

Software systems of any reasonable size are not usually coded from scratch any more. One of the justifications for Java as a programming language is the claim that modules can be implemented once, tested carefully, and then repurposed for use in other software projects without modification. The Java generic construct is part of what allows classes to be built, including the Java Collections Framework, that operate on objects whose types are specified at a later time. Facilitating this is the use of iterators that allow one to move from one data payload to another inside a structure without actually knowing what the underlying structure is. Finally, testing of programs using `JUnit` permits a standard test suite to be built and used to ensure that all parts of a method's code have been exercised.

5.9 Exercises

1. Create a `Pair` class using generics that will store an ordered pair of elements of any data type. Pass in the initial values as parameters to a constructor. Create a method `swap` inside `Pair` that will swap the first element and the second element. You will need a driver program to test this adequately.
2. Verify the correctness of the `Pair` code in the previous exercise by writing `JUnit` test functions instead of using just a driver program.

3. Instead of a `Pair` class that has just two values, write a `RotatableList` class using

   ```
   public class RotatableList<E> extends ArrayList<E>
   ```

   to extend `ArrayList` and allow you to write a method `rotate` that rotates the stored values, putting the first value in the last position and moving all the others up one position.
4. Look up the `Class` class in the Java documentation online, and notice that there is a `getClass` method and a `getName` method so that your program can in fact dynamically test the data type of a variable. Use this to implement a `select` method in the `Pair` class that will return the first element if the data type is `Integer`, the second element if the data type is `String`, and `null` for all other data types. Notice that this works because we are still doing nothing that requires us to look at the *values* of the elements.
5. Notice in the previous exercise that you can't really deal with the data stored in `Pair` because the compiler will insist that anything you do will work with all data types. For example, you can't compare the two elements and return the smaller. Similar to what was done with the `Record` class in the text, modify `Pair` to have it read `Pair<T extends Comparable<T>>`. You should now be able to compare elements of `Integer` and `String` type because these both have implemented a `compareTo` method. Now change the driver and try to create a `Pair` of elements of some other type that does not extend `Comparable`, and the code will not compile.
6. Follow up on Exercise 4 to resolve the problem. Similar to what was done with the `Record` class in the text, modify `Pair` to have it read `Pair<T extends Comparable<T>>` and create a wrapper class `MyString` (making sure that this class extends `Comparable`). Your `MyString` class need do nothing but be a wrapper for one instance variable of `String` type, perhaps named `localString`. Comparing pairs of `String` values is possible because the `String` class has a built-in `compareTo` that sorts data alphabetically. Override this with your own `compareTo` method for the `MyString` class so that the string data is compared according to the *length* of the string.
7. This exercise will allow you to build a prototype of classes using generics so you can steal code from yourself later. Start with an interface named `ITinyTest` that reads

   ```
   public interface ITinyTest<T>
   {
     public boolean isTiny();
   }
   ```

Implement both a `MyInteger` and a `MyString` class that are wrappers for the `Integer` and the `String` classes, respectively. Each of these should implement `ITinyTest`, which means that you must include a method `isTiny`. For `String` data, a string is tiny if it is equal to the word "tiny." For `Integer` data, an integer is tiny if it is equal to or less than −999999999. In between your driver main program and the two data classes, write a `MyData` class that uses generics and will allow you to test a generic `MyData` element for tininess. What you get from this is a class that handles instances of data defined as generic, together with an interface that you have defined and that requires looking at the values of the instance variables and underlying data payload classes that implement that interface.

8. Verify that you understand all the linked list code in this chapter by packaging all the linked list pieces together using generics. At the end, you will have a prototype of how to write linked list code as well as how to work with generics.
9. Verify that you understand all the iterator code in this chapter by packaging all the pieces together and testing them with `JUnit`.
10. Instead of using the `ListIterator` to implement your own iterator, use the `Iterator` interface.