SESSION 9

Programming Languages for Objects

Generic Programming

.1.1 Recursive Binary Search

Let's start with an example that you've seen before: the binary search algorithm from Subsection 7.4.1. Binary search is used to find a specified value in a sorted list of items (or, if it does not occur in the list, to determine that fact). The idea is to test the element in the middle of the list. If that element is equal to the specified value, you are done. If the specified value is less than the middle element of the list, then you should search for the value in the first half of the list. Otherwise, you should search for the value in the second half of the list. The method used to search for the value in the first or second half of the list is binary search. That is, you look at the middle element in the half of the list that is still under consideration, and either you've found the value you are looking for, or you have to apply binary search to one half of the remaining elements. And so on! This is a recursive description, and we can write a recursive subroutine to implement it.

Before we can do that, though, there are two considerations that we need to take into account. Each of these illustrates an important general fact about recursive subroutines. First of all, the binary search algorithm begins by looking at the "middle element of the list." But what if the list is empty? If there are no elements in the list, then it is impossible to look at the middle element. In the terminology of Subsection 8.2.1, having a non-empty list is a "precondition" for looking at the middle element, and this is a clue that we have to modify the algorithm to take this precondition into account. What should we do if we find ourselves searching for a specified value in an empty list? The answer is easy: If the list is empty, we can be sure that the value does not occur in the list, so we can give the answer without any further work. An empty list is a base case for the binary search algorithm. A base case for a recursive algorithm is a case that is handled directly, rather than by applying the algorithm recursively. The binary search algorithm actually has another type of base case: If we find the element we are looking for in the middle of the list, we are done. There is no need for further recursion.

The second consideration has to do with the parameters to the subroutine. The problem is phrased in terms of searching for a value in a list. In the original, non-recursive binary search subroutine, the list was given as an array. However, in the recursive approach, we have to be able to apply the subroutine recursively to just a **part** of the original list. Where the original subroutine was designed to search an entire array, the recursive subroutine must be able to search part of an array. The parameters to the subroutine must tell it what part of the array to search. This illustrates a general fact that in order to solve a problem recursively, it is often necessary to generalize the problem slightly.

GENERIC PROGRAMMING refers to writing code that will work for many types of data. We encountered the alternative to generic programming in Subsection 7.2.4, where we looked at dynamic arrays of integers. The source code presented there for working with dynamic arrays of integers works only for data of type int. But the source code for dynamic arrays of double, *String*, *JButton*, or any other type would be almost identical, except for the substitution of one type name for another. It seems silly to write essentially the same code over and over. Java's approach to this problem is parameterized types. As we saw in Section 7.3, the parameterized class *ArrayList* implements dynamic arrays. Since it is parameterized, there are types such as `ArrayList<String>` to represented dynamic arrays of *String*, `ArrayList<JButton>` for dynamic arrays of buttons, and more generally `ArrayList<T>` for any object type *T*. *ArrayList* is just one class, but the source code works for many different types. This is generic programming.

The *ArrayList* class is just one of several standard classes that are used for generic programming in Java. We will spend the next few sections looking at these classes and how they are used, and we'll see that there are also generic methods and generic interfaces. All the classes and interfaces discussed in these sections are defined in the package `java.util`, and you will need an import statement at the beginning of your program to get access to them. (Before you start putting "`import java.util.*`" at the beginning of every program, you should know that some things in `java.util` have names that are the same as things in other packages. For example, both `java.util.List` and `java.awt.List` exist, so it is often better to import the individual classes that you need.)

In the final section of this chapter, we will see that it is possible to define new generic classes, interfaces, and methods. Until then, we will stick to using the generics that are predefined in Java's standard library.

It is no easy task to design a library for generic programming. Java's solution has many nice features but is certainly not the only possible approach. It is almost certainly not the best, and has a few features that in my opinion can only be called bizarre, but in the context of the overall design of Java, it might be close to optimal. To get some perspective on generic programming in general, it might be useful to look very briefly at some other approaches to generic programming.

There are two obvious ways to represent a list: as a dynamic array and as a linked list. We've encountered these already in Section 7.3 and Section 9.2. Both of these options are available in generic form as the collection classes `java.util.ArrayList` and `java.util.LinkedList`. These classes are part of the Java Collection Framework. Each implements the interface *List<T>*, and therefore the interface *Collection<T>*. An object of type

ArrayList<T> represents an ordered sequence of objects of type *T*, stored in an array that will grow in size whenever necessary as new items are added. An object of type *LinkedList<T>* also represents an ordered sequence of objects of type *T*, but the objects are stored in nodes that are linked together with pointers.

Both list classes support the basic list operations that are defined in the interface *List<T>*, and an abstract data type is defined by its operations, not by its representation. So why two classes? Why not a single List class with a single representation? The problem is that there **is** no single representation of lists for which all list operations are efficient. For some operations, linked lists are more efficient than arrays. For others, arrays are more efficient. In a particular application of lists, it's likely that only a few operations will be used frequently. You want to choose the representation for which the frequently used operations will be as efficient as possible.

Broadly speaking, the *LinkedList* class is more efficient in applications where items will often be added or removed at the beginning of the list or in the middle of the list. In an array, these operations require moving a large number of items up or down one position in the array, to make a space for a new item or to fill in the hole left by the removal of an item. In terms of asymptotic analysis (Section 8.5), adding an element at the beginning or in the middle of an array has run time Θ(n), where n is the number of items in the array. In a linked list, nodes can be added or removed at any position by changing a few pointer values, an operation that has run time Θ(1). That is, the operation takes only some constant amount of time, independent of how many items are in the list.

On the other hand, the *ArrayList* class is more efficient when random access to items is required. Random access means accessing the k-th item in the list, for any integer k. Random access is used when you get or change the value stored at a specified position in the list. This is trivial for an array, with run time Θ(1). But for a linked list it means starting at the beginning of the list and moving from node to node along the list for k steps, an operation that has run time Θ(k).

Operations that can be done efficiently for both types of lists include sorting and adding an item at the end of the list.

All lists implement the methods from interface *Collection<T>* that were discussed in Subsection 10.1.4. These methods include `size()`, `isEmpty()`, `add(T)`, `remove(Object)`, and `clear()`. The `add(T)` method adds the object at the end of the list. The `remove(Object)` method involves first finding the object, which uses linear search and is not very efficient for any list since it involves going through the items in the list from beginning to end until the object is found. The interface *List<T>* adds some methods for accessing list items according to their numerical positions in the list. Suppose that `list` is an object of type *List<T>*. Then we have the methods:

- `list.get(index)` -- returns the object of type *T* that is at position `index` in the list, where `index` is an integer. Items are numbered 0, 1, 2, ..., `list.size()-1`. The parameter must be in this range, or an *IndexOutOfBoundsException* is thrown.
- `list.set(index,obj)` -- stores the object `obj` at position number `index` in the list, replacing the object that was there previously. The object `obj` must be of

type *T*. This does not change the number of elements in the list or move any of the other elements.

- `list.add(index,obj)` -- inserts an object `obj` into the list at position number `index`, where `obj` must be of type *T*. The number of items in the list increases by one, and items that come after position `index` move down one position to make room for the new item. The value of `index` must be in the range 0 to `list.size()`, inclusive. If `index` is equal to `list.size()`, then `obj` is added at the end of the list.
- `list.remove(index)` -- removes the object at position number `index`, and returns that object as the return value of the method. Items after this position move up one space in the list to fill the hole, and the size of the list decreases by one. The value of `index` must be in the range 0 to `list.size()-1`
- `list.indexOf(obj)` -- returns an int that gives the position of `obj` in the list, if it occurs. If it does not occur, the return value is `-1`. The object `obj` can be of any type, not just of type *T*. If `obj` occurs more than once in the list, the index of the first occurrence is returned.

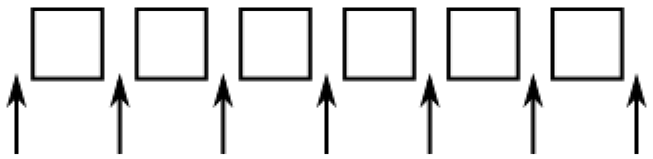
These methods are defined both in class *ArrayList<T>* and in class *LinkedList<T>*, although some of them -- `get` and `set` -- are only efficient for *ArrayLists*. The class *LinkedList<T>* adds a few additional methods, which are not defined for an *ArrayList*. If `linkedlist` is an object of type *LinkedList<T>*, then we have

- `linkedlist.getFirst()` -- returns the object of type *T* that is the first item in the list. The list is not modified. If the list is empty when the method is called, an exception of type *NoSuchElementException* is thrown (the same is true for the next three methods as well).
- `linkedlist.getLast()` -- returns the object of type *T* that is the last item in the list. The list is not modified.
- `linkedlist.removeFirst()` -- removes the first item from the list, and returns that object of type *T* as its return value. The functions `linkedlist.remove()` and `linkedlist.pop()` are also defined, with the same meaning as `removeFirst()`.
- `linkedlist.removeLast()` -- removes the last item from the list, and returns that object of type *T* as its return value.
- `linkedlist.addFirst(obj)` -- adds the `obj`, which must be of type *T*, to the beginning of the list. The function `linkedlist.push(obj)` has the same meaning.
- `linkedlist.addLast(obj)` -- adds the object `obj`, which must be of type *T*, to the end of the list. This is exactly the same as `linkedlist.add(obj)` but is defined to keep the naming consistent.

There is some redundancy here, apparently to make it easy to use a *LinkedList* as if it were a stack or a queue. (See Section 9.3.) For example, we can use a *LinkedList* as a stack by using the methods named `push()` and `pop()`, or as a queue by using `add()` and `remove()` to implement the enqueue and dequeue operations.

If `list` is an object of type *List<T>*, then the method `list.iterator()`, defined in the interface *Collection<T>*, returns an *Iterator* that can be used to traverse the list from beginning to end. However, for *Lists*, there is a special type of *Iterator*, called a *ListIterator*, which offers additional capabilities. *ListIterator<T>* is an interface that extends the interface *Iterator<T>*. The method `list.listIterator()` returns an object of type *ListIterator<T>*.

A *ListIterator* has the usual *Iterator* methods, `hasNext()`, `next()`, and `remove()`, but it also has methods `hasPrevious()`, `previous()`, `add(obj)`, and `set(obj)` that make it possible to move backwards in the list, to add an item at the current position of the iterator, and to replace one of the items in the list. To understand how these work, it's best to think of an iterator as pointing to a position **between** two list elements, or at the beginning or end of the list. In this diagram, the items in a list are represented by squares, and arrows indicate the possible positions of an iterator:



If `iter` is of type *ListIterator<T>*, then `iter.next()` moves the iterator one space to the right along the list and returns the item that the iterator passes as it moves. The method `iter.previous()` moves the iterator one space to the left along the list and returns the item that it passes. The method `iter.remove()` removes an item from the list; the item that is removed is the item that the iterator passed most recently in a call to either `iter.next()` or `iter.previous()`. The method `iter.set(obj)` works similarly; it replaces the item that would be removed by `iter.remove()`. There is also a method `iter.add(obj)` that adds the specified object to the list at the current position of the iterator (where `obj` must be of type *T*). This can be between two existing items or at the beginning of the list or at the end of the list.

(By the way, the lists that are used in class `LinkedList<T>` are doubly linked lists. That is, each node in the list contains two pointers -- one to the next node in the list and one to the previous node. This makes it possible to efficiently implement both the `next()` and `previous()` methods of a `ListIterator`. Also, to make the `addLast()` and `getLast()` methods of a *LinkedList* efficient, the class *LinkedList<T>* includes a "tail pointer" that points to the last node in the list.)

As an example of using a *ListIterator*, suppose that we want to maintain a list of items that is always sorted into increasing order. When adding an item to the list, we can use a *ListIterator* to find the position in the list where the item should be added. Once the position has been found, we use the same list iterator to place the item in that position. The idea is to start at the beginning of the list and to move the iterator forward past all the items that are smaller than the item that is being inserted. At that point, the iterator's `add()` method can be used to insert the item. To be more definite, suppose that `stringList` is a variable of type *List<String>*. Assume that that the strings that are already in the list are stored in ascending order and that `newItem` is a string that we would like to insert into the list. The following code will place `newItem` in the list in its correct position, so that the modified list is still in ascending order:

```
ListIterator<String> iter = stringList.listIterator();

// Move the iterator so that it points to the position where
// newItem should be inserted into the list.  If newItem is
// bigger than all the items in the list, then the while loop
// will end when iter.hasNext() becomes false, that is, when
// the iterator has reached the end of the list.

while (iter.hasNext()) {
   String item = iter.next();
   if (newItem.compareTo(item) <= 0) {
         // newItem should come BEFORE item in the list.
         // Move the iterator back one space so that
         // it points to the correct insertion point,
         // and end the loop.
      iter.previous();
      break;
   }
}

iter.add(newItem);
```

Here, `stringList` might be of type *ArrayList<String>* or of type *LinkedList<String>*. The algorithm that is used to insert `newItem` into the list will be about equally efficient for both types of lists, and it will even work for other classes that implement the interface *List<String>*. You would probably find it easier to design an insertion algorithm that uses array-like indexing with the methods `get(index)` and `add(index,obj)`. However, that algorithm would be horribly inefficient for *LinkedLists* because random access is so inefficient for linked lists. (By the way, the insertion algorithm works when the list is empty. It might be useful for you to think about why this is true.)

10.2.2 Sorting

Sorting a list is a fairly common operation, and there should really be a sorting method in the *List* interface. There is not, presumably because it only makes sense to sort lists of certain types of objects. However, methods for sorting lists are available as `static` methods in the class `java.util.Collections`. This class contains a variety of static utility methods for working with collections. The methods are generic; that is, they will work for collections of objects of various types. (You have already seen similar methods for arrays in the *Arrays* class.) Suppose that `list` is of type *List<T>*. The command

```
Collections.sort(list);
```

can be used to sort the list into ascending order. The items in the list should implement the interface *Comparable<T>* (see Subsection 10.1.6). The method `Collections.sort()` will work, for example, for lists of *String* and for lists of any of the wrapper classes such as *Integer* and *Double*. There is also a sorting method that takes a *Comparator* as its second argument:

```
Collections.sort(list,comparator);
```

In this method, the `comparator` will be used to compare the items in the list. As mentioned in the previous section, a *Comparator* is an object that defines a `compare()` method that can be used to compare two objects. We'll see an example of using a *Comparator* in Section 10.4.

The sorting method that is used by `Collections.sort()` is the so-called "merge sort" algorithm, which has both worst-case and average-case run times that are Θ(n*log(n)) for a list of size n. Although the average run time for MergeSort is a little slower than that of QuickSort, its worst-case performance is much better than QuickSort's. (QuickSort was covered in Subsection 9.1.3.) MergeSort also has a nice property called "stability" that we will encounter at the end of Subsection 10.4.3.

The *Collections* class has at least two other useful methods for modifying lists. `Collections.shuffle(list)` will rearrange the elements of the list into a random order. `Collections.reverse(list)` will reverse the order of the elements, so that the last element is moved to the beginning of the list, the next-to-last element to the second position, and so on.

Since an efficient sorting method is provided for *Lists*, there is no need to write one yourself.

10.2.3 TreeSet and HashSet

A set is a collection of objects in which no object occurs more than once. Sets implement all the methods in the interface *Collection<T>*, but do so in a way that ensures that no element occurs twice in the set. For example, if `set` is an object of type *Set<T>*, then `set.add(obj)` will have no effect on the set if `obj` is already an element of the set. Java has two classes that implement the interface *Set<T>*: `java.util.TreeSet` and `java.util.HashSet`.

In addition to being a *Set*, a *TreeSet* has the property that the elements of the set are arranged into ascending sorted order. An *Iterator* (or a for-each loop) for a `TreeSet` will always visit the elements of the set in ascending order.

A *TreeSet* cannot hold arbitrary objects, since there must be a way to determine the sorted order of the objects it contains. Ordinarily, this means that the objects in a set of type *TreeSet<T>* should implement the interface *Comparable<T>* and that `obj1.compareTo(obj2)` should be defined in a reasonable way for any two objects `obj1` and `obj2` in the set. Alternatively, an object of type *Comparator<T>* can be provided as a parameter to the constructor when the *TreeSet* is created. In that case, the `compare()` method of the *Comparator* will be used to compare objects that are added to the set.

A *TreeSet* does not use the `equals()` method to test whether two objects are the same. Instead, it uses the `compareTo()` (or `compare()`) method. This can be a problem. Recall from Subsection 10.1.6 that `compareTo()` can consider two objects to be the same for the purpose of the comparison even though the objects are not equal. For a *TreeSet*, this means that only **one** of those objects can be in the set. For example, if the *TreeSet* contains mailing addresses and if the `compareTo()` method for addresses just compares their zip codes, then the set can contain only one address in each zip code. Clearly, this is not right! But that only means that you have to be aware of the semantics of *TreeSets*, and you need to make sure that `compareTo()` is defined in a reasonable way for objects that you put into a *TreeSet*. This will be true, by the way, for *Strings*, *Integers*, and many other built-in types, since the `compareTo()` method for these types considers two objects to be the same only if they are actually equal.

In the implementation of a *TreeSet*, the elements are stored in something similar to a binary sort tree. (See Subsection 9.4.2.) However, the data structure that is used is **balanced** in the sense that all the leaves of the tree are at about the same distance from the root of the tree. This ensures that all the basic operations -- inserting, deleting, and searching -- are efficient, with worst-case run time Θ(log(n)), where n is the number of items in the set.

The fact that a *TreeSet* sorts its elements and removes duplicates makes it very useful in some applications. Exercise 7.7 asked you to write a program that would read a file and output an alphabetical list of all the words that occurred in the file, with duplicates removed. The words were to be stored in an *ArrayList*, so it was up to you to make sure that the list was sorted and contained no duplicates. The same task can be programmed much more easily using a *TreeSet* instead of a list. A *TreeSet* automatically eliminates duplicates, and an iterator for the set will automatically visit the items in the set in sorted order. An algorithm for the program, using a *TreeSet*, would be:

```
TreeSet<String> words = new TreeSet<String>();

while there is more data in the input file:
   Let word = the next word from the file
   Convert word to lower case
   words.add(word)    // Adds the word only if not already
present.

for ( String w : words ) // for each String w in words
   Output w  // words are output in sorted order
```

If you would like to see a complete, working program, you can find it in the file *WordListWithTreeSet.java*.

As another example, suppose that `coll` is any `Collection` of *Strings*. (This would also work for any other type for which `compareTo()` is properly defined.) We can use a *TreeSet* to sort the items of `coll` and remove the duplicates simply by saying:

```
TreeSet<String> set = new TreeSet<String>();
set.addAll(coll);
```

The second statement adds all the elements of the collection to the set. Since it's a *Set*, duplicates are ignored. Since it's a *TreeSet*, the elements of the set are sorted. If you would like to have the data in some other type of data structure, it's easy to copy the data from the set. For example, to place the answer in an *ArrayList*, you could say:

```
TreeSet<String> set = new TreeSet<String>();
set.addAll(coll);
ArrayList<String> list = new ArrayList<String>();
list.addAll(set);
```

Now, in fact, every one of Java's collection classes has a constructor that takes a *Collection* as an argument. All the items in that *Collection* are added to the new collection when it is created. So, if `coll` is of type *Collection<String>*, then "`new TreeSet<String>(coll)`" creates a *TreeSet* that contains the same elements as `coll`, but with duplicates removed and in sorted order. This means that we can abbreviate the four lines in the above example to the single command:

```
ArrayList<String> list = new ArrayList<String>( new
TreeSet<String>(coll) );
```

This makes a sorted list of the elements of `coll` with no duplicates. Although the repeated type parameter, "`<String>`", makes it a bit ugly to look at, this is still a nice example of the power of generic programming.

A *HashSet* stores its elements in a hash table, a type of data structure that I will discuss in the next section. The operations of finding, adding, and removing elements are implemented very efficiently in hash tables, even more so than for *TreeSets*. The elements of a *HashSet* are not stored in any particular order, and so do not need to implement the *Comparable* interface. (They do, however, need to define a proper "hash code," as we'll see in the next section.)

The `equals()` method is used to determine whether two objects in a *HashSet* are to be considered the same. An *Iterator* for a *HashSet* will visit its elements in what seems to be a completely arbitrary order, and it's possible for the order to change completely when a new element is added. Use a *HashSet* instead of a *TreeSet* when the elements it contains are not comparable, or when the order is not important, or when the small advantage in efficiency is important.

A note about the mathematics of sets: In mathematical set theory, the items in a set are called members or elements of that set. Important operations include adding an element to a set, removing an element from a set, and testing whether a given entity is an element of a set. Operations that can be performed on two sets include union, intersection, and set difference. All these operations are defined in Java for objects of type *Set*, but with different names. Suppose that `A` and `B` are *Sets*. Then:

- `A.add(x)` **adds** the element `x` to the set `A`.
- `A.remove(x)` **removes** the element `x` from the set `A`.
- `A.contains(x)` **tests** whether `x` is an element of the set `A`.
- `A.addAll(B)` computes the **union** of `A` and `B`.
- `A.retainAll(B)` computes the **intersection** of `A` and `B`.
- `A.removeAll(B)` computes the **set difference**, `A - B`.

There are of course, differences between mathematical sets and sets in Java. Most important, perhaps, sets in Java must be finite, while in mathematics, most of the fun in set theory comes from working with infinity. In mathematics, a set can contain arbitrary elements, while in Java, a set of type *Set<T>* can only contain elements of type *T*. The operation `A.addAll(B)` acts by modifying the value of `A`, while in mathematics the operation `A` union `B` computes a new set, without changing the value of `A` or `B`. See Exercise 10.2 for an example of mathematical set operations in Java.

10.2.4 EnumSet

Enumerated types (or "enums") were introduced in Subsection 2.3.4. Suppose that *E* is an enumerated type. Since *E* is a class, it is possible to create objects of type *TreeSet<E>* and *HashSet<E>*. However, because enums are so simple, trees and hash tables are not the most efficient implementation for sets of enumerated type values. Java provides the class `java.util.EnumSet` as an alternative way to create such sets.

Sets of enumerated type values are created using `static` methods in the class *EnumSet*. For example, if `e1`, `e2`, and `e3` are values belonging to the enumerated type *E*, then the method

```
EnumSet.of( e1, e2, e3 )
```

creates and returns a set of type *EnumSet<E>* that contains exactly the elements `e1`, `e2`, and `e3`. The set implements the interface *Set<E>*, so all the usual set and collection operations are available. The implementation of these operations is very efficient. The implementation uses what is called a *bit vector*. A bit is a quantity that has only two possible values, zero and one. A set of type *EnumSet<E>* is represented by a bit vector that contains one bit for each enum constant in the enumerated type *E*; the bit corresponding to the enum constant `e` is 1 if `e` is a member of the set and is 0 if `e` is not a member of the set. The bit vectors for two sets of type *EnumSet<E>* can be very easily combined to represent such operations as the union and intersection of two sets. The bit vector representation is feasible for *EnumSets*, but not for other sets in Java, because an enumerated type contains only a small finite number of enum constants. (Java actually has a class named *BitSet* that uses bit vectors to represent finite sets of non-negative integers, but this class is not part of the Java Collection Framework and does not implement the *Set* interface.)

The function `EnumSet.of` can be used with any positive number of parameters. All the parameters must be values of the same enumerated type. Null values are not allowed. An *EnumSet* cannot contain the value `null` -- any attempt to add `null` to an *EnumSet* will result in a *NullPointerException*.

There is also a function `EnumSet.range(e1,e2)` that returns an *EnumSet* consisting of the enum constants between `e1` and `e2`, inclusive. The ordering of enum constants is the same as the order in which they are listed in the definition of the enum. In `EnumSet.range(e1,e2)`, `e1` and `e2` must belong to the same enumerated type, and `e1` must be less than or equal to `e2`.

If *E* is an enum, then `EnumSet.allOf(E.class)` is a set that contains all values of type *E*. `EnumSet.noneOf(E.class)` is an empty set, a set of type *EnumSet<E>* that contains no elements at all. Note that in `EnumSet.allOf(E.class)` and `EnumSet.noneOf(E.class)`, the odd-looking parameter represents the enumerated type class itself. If `eset` is a set of type *EnumSet<E>*, then `EnumSet.complementOf(eset)` is a set that contains all the enum constants of *E* that are **not** in `eset`.

As an example, consider a program that keeps schedules of events. The program must keep track of repeating events that happen on specified days of the week. For example, an event might take place only on weekdays, or only on Wednesdays and Fridays. In other words, associated with the event is the **set** of days of the week on which it takes place. This information can be represented using the enumerated type

```
enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
FRIDAY, SATURDAY }
```

The days of the week on which an event takes place would then be a value of type *EnumSet<Day>*. An object of type *RepeatingEvent* would have an instance variable of type *EnumSet<Day>* to hold this information. An event that takes place on Wednesdays and Fridays would have the associated set

```
EnumSet.of( Day.WEDNESDAY, Day.FRIDAY )
```

We could define some common sets of *Days* as

```
EnumSet<Day> weekday = EnumSet.range( Day.MONDAY, Day.FRIDAY );
EnumSet<Day> weekend = EnumSet.complementOf( weekday );
EnumSet<Day> everyday = EnumSet.allOf( Day.class );
```

EnumSets are often used to specify sets of "options" that are to be applied during some type of processing. For example, a program that draws characters in fancy fonts might have various options that can be applied. Let's say that the options are bold, italic, underlined, strikethrough, and boxed. Note that we are assuming that options can be combined in arbitrary ways. For example, you can have italic, boxed, underlined characters. This just means that we need to keep track of a **set** of options. If the options are represented by the enumerated type

```
enum FontOption { BOLD, ITALIC, UNDERLINED, STRIKETHROUGH,
BOXED }
```

then a set of options is represented by a value of type *EnumSet<FontOption>*. Suppose that `options` is a variable of this type that represents the set of options that are currently being applied by the program. Then we can do things like:

- `options = EnumSet.noneOf( FontOption.class )` -- Turn off all options.
- `options = EnumSet.of( FontOption.BOLD )` -- Use bold, with no other options.
- `options.add( FontOption.BOLD )` -- Add bold to any options that are already on.
- `options.remove( FontOption.UNDERLINED )` -- Turn underlining off (if it's on).

This is a nice, safe way to work with sets of options. Applications like this are one of the major reasons that enumerated types were introduced.

10.2.5 Priority Queues

A priority queue is an ADT that represents a collection of items, where each item has an assigned "priority" that allows any two items to be compared. Operations on a priority queue include *add*, which adds an item to the collection, and *remove*, which removes and returns an item from the collection that has the minimum priority among items currently in the collection. (Maximum priority would also be possible, but in Java's version, the *remove* operation removes a minimum priority item.)

A simple implementation of priority queue could be achieved by using a linked list to store the items in the queue in order of increasing priority. In that case, *remove* would simply remove and return the first item in the list. However, *add* would have to insert the new item into its correct position in the list, an operation with average run time $\Theta(n)$, where n is the number of items in the list. In fact, priority queues can be implemented so that both *add* and *remove* have run time $\Theta(\log(n))$, which is much more efficient. (The efficient implementation uses something called a "heap," which is not to be confused with the heap where objects are created. I will not discuss the implementation here.)

The parameterized class *PriorityQueue<T>* implements a priority queue of objects of type *T*. This class implements the interface *Collection<T>*. So, if `pq` is a *PriorityQueue*, then it has all the methods defined in that interface. But the essential priority queue operations are

- `pg.add(obj)` -- adds `obj` to the priority queue, where `obj` must be an object of type *T*

- `pg.remove()` -- removes and returns an item of minimal priority. The return value is an object of type *T*. Throws an exception if the queue is empty.
- `pg.isEmpty()` -- tests whether the priority queue is empty.

You've probably noticed that I haven't yet mentioned how the priority of items in the priority queue are determined. The situation is much like sorting: We need to be able to compare any two items in the queue. As with sorting, there are two solutions. If the items implement the *Comparable* interface, then they can be compared using the `compareTo()` method from that interface. Alternatively, a *Comparator* object can be provided as a parameter to the *PriorityQueue* constructor. In that case, the Comparator's `compare` method will be used to compare items.

Classes such as *String*, *Integer*, and *Date* that implement *Comparable* can be used in a priority queue. For example, a *PriorityQueue<String>* can be used to sort strings into lexicographic order: Just *add* all the strings to the priority queue, then *remove* them one-by-one. Since items are removed from a queue in order of priority, they will be removed in lexicographic order. Earlier, I showed how to use a *TreeSet* to sort and remove duplicates from a collection. A *PriorityQueue* can be used in a similar way to sort a collection without removing duplicates. For example, if `coll` is of type *Collection<String>*, then the following code segment will print all the items from `coll` in order, including duplicates:

```
PriorityQueue<String> pq = new PriorityQueue<String>();
pq.addAll( coll );
while ( ! pq.isEmpty() ) {
    System.out.println( pq.remove() );
}
```

(Note, by the way, that we can't use an iterator or a for-each loop to print the items in this example, since interators and for-each loops do not traverse a priority queue in ascending order.)

The sample program *WordListWithPriorityQueue.java* makes a sorted list of words from a file without removing duplicates, using a priority queue to hold the words. It is a minor modification of *WordListWithTreeSet.java*.

Although priority queues can be used for sorting, they have other natural applications. For example, consider the problem of scheduling "jobs" to be executed on a computer, where each job is assigned a priority and jobs with lower priority should always be executed before jobs with higher priority. Jobs can be placed into a priority queue as they are created. When the computer removes jobs from the queue for execution, they will be removed in order of increasing priority.

10.1.1 Generic Programming in Smalltalk

Smalltalk was one of the very first object-oriented programming languages. It is still used today, although its use is not very common. It never achieved anything like the popularity of Java or C++, but it is the source of many ideas used in these languages. In Smalltalk, essentially all programming is generic, because of two basic properties of the language.

First of all, variables in Smalltalk are typeless. A data value has a type, such as integer or string, but variables do not have types. Any variable can hold data of any type. Parameters are also typeless, so a subroutine can be applied to parameter values of any type. Similarly, a data structure can hold data values of any type. For example, once you've defined a binary tree data structure in SmallTalk, you can use it for binary trees of integers or strings or dates or data of any other type. There is simply no need to write new code for each data type.

Secondly, all data values are objects, and all operations on objects are defined by methods in a class. This is true even for types that are "primitive" in Java, such as integers. When the "+" operator is used to add two integers, the operation is performed by calling a method in the integer class. When you define a new class, you can define a "+" operator, and you will then be able to add objects belonging to that class by saying "`a + b`" just as if you were adding numbers. Now, suppose that you write a subroutine that uses the "+" operator to add up the items in a list. The subroutine can be applied to a list of integers, but it can also be applied, automatically, to any other data type for which "+" is defined. Similarly, a subroutine that uses the "<**"** operator to sort a list can be applied to lists containing any type of data for which "<" is defined. There is no need to write a different sorting subroutine for each type of data.

Put these two features together and you have a language where data structures and algorithms will work for any type of data for which they make sense, that is, for which the appropriate operations are defined. This is real generic programming. This might sound pretty good, and you might be asking yourself why all programming languages don't work this way. This type of freedom makes it easier to write programs, but unfortunately it makes it harder to write programs that are correct and robust (see Chapter 8). Once you have a data structure that can contain data of any type, it becomes hard to ensure that it only holds the type of data that you want it to hold. If you have a subroutine that can sort any type of data, it's hard to ensure that it will only be applied to data for which the "<" operator is defined. More particularly, there is no way for a **compiler** to ensure these things. The problem will only show up at run time when an attempt is made to apply some operation to a data type for which it is not defined, and the program will crash.

10.1.2 Generic Programming in C++

Unlike Smalltalk, C++ is a very strongly typed language, even more so than Java. Every variable has a type, and can only hold data values of that type. This means that the kind of generic programming that is used in Smalltalk is impossible in C++. Nevertheless, C++ has a powerful and flexible system of generic programming. It is made possible by a language feature known as templates. In C++, instead of writing a different sorting subroutine for each type of data, you can write a single subroutine template. The template is not a subroutine; it's more like a factory for making subroutines. We can look at an example, since the syntax of C++ is very similar to Java's:

```
template<class ItemType>
void sort( ItemType A[], int count ) {
      // Sort items in the array, A, into increasing order.
      // The items in positions 0, 1, 2, ..., (count-1) are
sorted.
      // The algorithm that is used here is selection sort.
   for (int i = count-1; i > 0; i--) {
      int position_of_max = 0;
      for (int j = 1; j <= count ; j++)
         if ( A[j] > A[position_of_max] )
            position_of_max = j;
      ItemType temp = A[count];
      A[count] = A[position_of_max];
      A[position_of_max] = temp;
   }
}
```

This piece of code defines a subroutine template. If you remove the first line, "template<class ItemType>", and substitute the word "int" for the word "ItemType" in the rest of the template, you get a subroutine for sorting arrays of ints. (Even though it says "class ItemType", you can actually substitute any type for ItemType, including the primitive types.) If you substitute "string" for "ItemType", you get a subroutine for sorting arrays of strings. This is pretty much what the compiler does with the template. If your program says "`sort(list,10)`" where list is an array of ints, the compiler uses the template to generate a subroutine for sorting arrays of ints. If you say "`sort(cards,10)`" where cards is an array of objects of type *Card*, then the compiler generates a subroutine for sorting arrays of *Cards*. At least, it tries to. The template uses the ">" operator to compare values. If this operator is defined for values of type *Card*, then the compiler will successfully use the template to generate a subroutine for sorting cards. If ">" is not defined for *Cards*, then the compiler will fail -- but this will happen at compile time, not, as in Smalltalk, at run time where it would make the program crash. (By the way, in C++, it is possible to write definitions of operators like > for any type, so that it is possible that > might work for values of type *Card*.)

In addition to subroutine templates, C++ also has templates for making classes. If you write a template for a binary tree class, you can use it to generate classes for binary trees of ints, binary trees of strings, binary trees of dates, and so on -- all from one template. Modern C++ comes with a large number of pre-written templates called the Standard Template Library or STL. The STL is quite complex. Many people would say that it's much too complex. But it is also one of the most interesting features of C++.

10.1.3 Generic Programming in Java

Java's generic programming features have gone through several stages of development. Early versions of Java did not have parameterized types, but they did have classes to represent common data structures. Those classes were designed to work with *Objects*; that is, they could hold objects of any type, and there was no way to restrict the types of objects that could be stored in a given data structure. For example, *ArrayList* was not originally a parameterized type, so that any ArrayList could hold any type of object. This means that if `list` was an *ArrayList*, then `list.get(i)` would return a value of type *Object*. If the programmer was actually using the list to store *Strings*, the value returned by `list.get(i)` would have to be type-cast to treat it as a string:

```
            String item = (String)list.get(i);
```

This is still a kind of generic programming, since one class can work for any kind of object, but it was closer in spirit to Smalltalk than it was to C++, since there is no way to do type checks at compile time. Unfortunately, as in Smalltalk, the result is a category of errors that show up only at run time, rather than at compile time. If a programmer assumes that all the items in a data structure are strings and tries to process those items as strings, a run-time error will occur if other types of data have inadvertently been added to the data structure. In Java, the error will most likely occur when the program retrieves an *Object* from the data structure and tries to type-cast it to type *String*. If the object is not actually of type *String*, the illegal type-cast will throw an error of type *ClassCastException*.

Java 5.0 introduced parameterized types, which made it possible to create generic data structures that can be type-checked at compile time rather than at run time. For example, if `list` is of type `ArrayList<String>`, then the compiler will only allow objects of type *String* to be added to `list`. Furthermore, the return type of `list.get(i)` is *String*, so type-casting is not necessary. Java's parameterized classes are similar to template classes in C++ (although the implementation is very different), and their introduction moves Java's generic programming model closer to C++ and farther from Smalltalk. In this chapter, I will use the parameterized types exclusively, but you should remember that their use is not mandatory. It is still legal to use a parameterized class as a non-parameterized type, such as a plain *ArrayList*. In that case, any type of object can be stored in the data structure.

Note that there is a significant difference between parameterized classes in Java and template classes in C++. A template class in C++ is not really a class at all -- it's a kind of factory for generating classes. Every time the template is used with a new type, a new compiled class is created. With a Java parameterized class, there is only one compiled class file. For example, there is only one compiled class file, `ArrayList.class`, for the parameterized class *ArrayList*. The parameterized types *ArrayList<String>* and *ArrayList<Integer>* both use the same compiled class file, as does the plain *ArrayList* type. The type parameter -- *String* or *Integer* -- just tells the compiler to limit the type of object that can be stored in the data structure. The type parameter has no effect at run time and is not even known at run time. The type information is said to be "erased" at run time. This type erasure introduces a certain amount of weirdness. For example, you can't test "`if (list instanceof ArrayList<String>)`" because the `instanceof` operator is evaluated at run time, and at run time only the plain *ArrayList* exists. Similarly, you can't type-cast to the type `ArrayList<String>`. Even worse, you can't create an array that has base type `ArrayList<String>` by using the `new` operator, as in "`new ArrayList<String>[N]`". This is because the `new` operator is evaluated at run time, and at run time there is no such thing as "`ArrayList<String>`"; only the non-parameterized type *ArrayList* exists at run time. (However, although you can't have an array of `ArrayList<String>`, you **can** have an ArrayList of `ArrayList<String>` -- with the type written as `ArrayList<ArrayList<String>>` -- which is just as good or better.)

Fortunately, most programmers don't have to deal with such problems, since they turn up only in fairly advanced programming. Most people who use parameterized types will not encounter the problems, and they will get the benefits of type-safe generic programming with little difficulty.

10.1.4 The Java Collection Framework

As I've said, Java comes with a number of parameterized classes that implement common data structures. This collection of data structure classes is referred to as the Java Collection Framework, or JFC. We will spend the next few sections learning about the JFC.

The generic data structures in the Java Collection Framework can be divided into two categories: collections and maps. A collection is more or less what it sounds like: a collection of objects. A map associates objects in one set with objects in another set in the way that a dictionary associates definitions with words or a phone book associates phone numbers with names. A map is similar to what I called an "association list" in Subsection 7.4.2. In Java, collections and maps are represented by the parameterized interfaces *Collection<T>* and *Map<T,S>*. Here, "T" and "S" stand for any type except for the primitive types. *Map<T,S>* is an example of a parameterized type that ahs two type parameters, *T* and *S*; we will not deal further with this possibility until we look at maps more closely in Section 10.3. In this section and the next, we look at collections only.

There are two types of collections: lists and sets. A list is a collection in which the objects are arranged in a linear sequence. A list has a first item, a second item, and so on. For any item in the list, except the last, there is an item that directly follows it. For collections that are "sets," the defining property is that no object can occur more than once in a set; the elements of a set are not necessarily thought of as being in any particular order. The ideas of lists and sets are represented as parameterized interfaces *List<T>* and *Set<T>*. These are sub-interfaces of *Collection<T>*. That is, any object that implements the interface *List<T>* or *Set<T>* automatically implements *Collection<T>* as well. The interface *Collection<T>* specifies general operations that can be applied to any collection at all. *List<T>* and *Set<T>* add additional operations that are appropriate for lists and sets respectively.

Of course, any actual object that is a collection, list, or set must belong to a concrete class that implements the corresponding interface. For example, the class *ArrayList<T>* implements the interface *List<T>* and therefore also implements *Collection<T>*. This means that all the methods that are defined in the list and collection interfaces can be used with an ArrayList. We will look at various classes that implement the list and set interfaces in the next section. But before we do that, we'll look briefly at some of the general operations that are available for all collections.

The interface *Collection<T>* specifies methods for performing some basic operations on any collection of objects. Since "collection" is a very general concept, operations that can be applied to all collections are also very general. They are generic operations in the sense that they can be applied to various types of collections containing various types of objects. Suppose that `coll` is an object that implements the interface *Collection<T>* (for some specific non-primitive type *T*). Then the following operations, which are specified in the interface *Collection<T>*, are defined for `coll`:

- `coll.size()` -- returns an int that gives the number of objects in the collection.
- `coll.isEmpty()` -- returns a boolean value which is `true` if the size of the collection is 0.
- `coll.clear()` -- removes all objects from the collection.
- `coll.add(tobject)` -- adds `tobject` to the collection. The parameter must be of type *T*; if not, a syntax error occurs at compile time. (Remember that if *T* is a class, this includes objects belonging to a subclass of *T*, and if *T* is an interface, it includes any object that implements *T*.) The `add()` method returns a boolean value which tells you whether the operation actually modified the collection. For example, adding an object to a `Set` has no effect if that object was already in the set.
- `coll.contains(object)` -- returns a boolean value that is true if `object` is in the collection. Note that `object` is **not** required to be of type *T*, since it makes sense to check whether `object` is in the collection, no matter what type `object` has. (For testing equality, `null` is considered to be equal to itself. The criterion for testing non-null objects for equality can differ from one kind of collection to another; see Subsection 10.1.6, below.)
- `coll.remove(object)` -- removes `object` from the collection, if it occurs in the collection, and returns a boolean value that tells you whether the object was found. Again, `object` is not required to be of type *T*. The test for equality is the same test that is used by `contains()`.
- `coll.containsAll(coll2)` -- returns a boolean value that is true if every `object` in `coll2` is also in `coll`. The parameter can be any collection.
- `coll.addAll(coll2)` -- adds all the objects in `coll2` to `coll`. The parameter, `coll2`, can be any collection of type *Collection<T>*. However, it can also be more general. For example, if *T* is a class and *S* is a sub-class of *T*, then `coll2` can be of type *Collection<S>*. This makes sense because any object of type *S* is automatically of type *T* and so can legally be added to `coll`.
- `coll.removeAll(coll2)` -- removes every `object` from `coll` that also occurs in the collection `coll2`. `coll2` can be any collection.
- `coll.retainAll(coll2)` -- removes every `object` from `coll` that **does not occur** in the collection `coll2`. It "retains" only the objects that do occur in `coll2`. `coll2` can be any collection.
- `coll.toArray()` -- returns an array of type `Object[]` that contains all the items in the collection. Note that the return type is `Object[]`, not `T[]`! However,

there is another version of this method that takes an array of type `T[]` as a parameter: the method `coll.toArray(tarray)` returns an array of type `T[]` containing all the items in the collection. If the array parameter `tarray` is large enough to hold the entire collection, then the items are stored in `tarray` and `tarray` is also the return value of the collection. If `tarray` is not large enough, then a new array is created to hold the items; in that case `tarray` serves only to specify the type of the array. For example, `coll.toArray(new String[0])` can be used if `coll` is a collection of *Strings* and will return a new array of type `String[]`.

Since these methods are part of the *Collection<T>* interface, they must be defined for every object that implements that interface. There is a problem with this, however. For example, the size of some collections cannot be changed after they are created. Methods that add or remove objects don't make sense for these collections. While it is still legal to call the methods, an exception will be thrown when the call is evaluated at run time. The type of the exception is *UnsupportedOperationException*. Furthermore, since *Collection<T>* is only an interface, not a concrete class, the actual implementation of the method is left to the classes that implement the interface. This means that the semantics of the methods, as described above, are not guaranteed to be valid for all collection objects; they are valid, however, for classes in the Java Collection Framework.

There is also the question of efficiency. Even when an operation is defined for several types of collections, it might not be equally efficient in all cases. Even a method as simple as `size()` can vary greatly in efficiency. For some collections, computing the `size()` might involve counting the items in the collection. The number of steps in this process is equal to the number of items. Other collections might have instance variables to keep track of the size, so evaluating `size()` just means returning the value of a variable. In this case, the computation takes only one step, no matter how many items there are. When working with collections, it's good to have some idea of how efficient operations are and to choose a collection for which the operations that you need can be implemented most efficiently. We'll see specific examples of this in the next two sections.

10.1.5 Iterators and for-each Loops

The interface *Collection<T>* defines a few basic generic algorithms, but suppose you want to write your own generic algorithms. Suppose, for example, you want to do something as simple as printing out every item in a collection. To do this in a generic way, you need some way of going through an arbitrary collection, accessing each item in turn. We have seen how to do this for specific data structures: For an array, you can use a `for` loop to iterate through all the array indices. For a linked list, you can use a while loop in which you advance a pointer along the list. For a binary tree, you can use a recursive subroutine to do an inorder traversal. Collections can be represented in any of these forms and many others besides. With such a variety of traversal mechanisms, how can we even hope to come up with a single generic method that will work for collections that are stored in wildly different forms? This problem is solved by iterators. An iterator is an object that can be used to traverse a collection. Different types of collections have iterators that are implemented in different ways, but all iterators are **used** in the same way. An algorithm that uses an iterator to traverse a collection is generic, because the same technique can be applied to any type of collection. Iterators can seem rather strange to someone who is encountering generic programming for the first time, but you should understand that they solve a difficult problem in an elegant way.

The interface *Collection<T>* defines a method that can be used to obtain an iterator for any collection. If `coll` is a collection, then `coll.iterator()` returns an iterator that can be used to traverse the collection. You should think of the iterator as a kind of generalized pointer that starts at the beginning of the collection and can move along the collection from one item to the next. Iterators are defined by a parameterized interface named *Iterator<T>*. If `coll` implements the interface *Collection<T>* for some specific type *T*, then `coll.iterator()` returns an iterator of type *Iterator<T>*, with the same type *T* as its type parameter. The interface *Iterator<T>* defines just three methods. If `iter` refers to an object that implements *Iterator<T>*, then we have:

- `iter.next()` -- returns the next item, and advances the iterator. The return value is of type *T*. This method lets you look at one of the items in the collection. Note that there is no way to look at an item without advancing the iterator past that item. If this method is called when no items remain, it will throw a *NoSuchElementException*.
- `iter.hasNext()` -- returns a boolean value telling you whether there are more items to be processed. In general, you should test this before calling `iter.next()`.
- `iter.remove()` -- if you call this after calling `iter.next()`, it will remove the item that you just saw from the collection. Note that this method has **no parameter**. It removes the item that was most recently returned by `iter.next()`. This might produce an *UnsupportedOperationException*, if the collection does not support removal of items.

Using iterators, we can write code for printing all the items in **any** collection. Suppose, for example, that `coll` is of type *Collection<String>*. In that case, the value returned by `coll.iterator()` is of type *Iterator<String>*, and we can say:

```
Iterator<String> iter;          // Declare the iterator
variable.
iter = coll.iterator();         // Get an iterator for the
collection.
while ( iter.hasNext() ) {
   String item = iter.next();   // Get the next item.
   System.out.println(item);
}
```

The same general form will work for other types of processing. For example, the following code will remove all `null` values from any collection of type *Collection<JButton>* (as long as that collection supports removal of values):

```
Iterator<JButton> iter = coll.iterator():
while ( iter.hasNext() ) {
    JButton item = iter.next();
    if (item == null)
       iter.remove();
}
```

(Note, by the way, that when *Collection<T>*, *Iterator<T>*, or any other parameterized type is used in actual code, they are always used with actual types such as *String* or *JButton* in place of the "formal type parameter" *T*. An iterator of type *Iterator<String>* is used to iterate through a collection of *Strings*; an iterator of type *Iterator<JButton>* is used to iterate through a collection of *JButtons*; and so on.)

An iterator is often used to apply the same operation to all the elements in a collection. In many cases, it's possible to avoid the use of iterators for this purpose by using a for-each loop. The for-each loop was discussed in Subsection 7.1.1 for use with arrays and in Subsection 7.3.3 for use with ArrayLists. But in fact, a for-each loop can be used to iterate through any collection. For a collection `coll` of type *Collection<T>*, a for-each loop takes the form:

```
for ( T x : coll ) { // "for each object x, of type T, in coll"
   //  process x
}
```

Here, `x` is the loop control variable. Each object in `coll` will be assigned to `x` in turn, and the body of the loop will be executed for each object. Since objects in `coll` are of type *T*, `x` is declared to be of type *T*. For example, if `namelist` is of type *Collection<String>*, we can print out all the names in the collection with:

```
for ( String name : namelist ) {
   System.out.println( name );
}
```

This for-each loop could, of course, be written as a `while` loop using an iterator, but the for-each loop is much easier to follow.

10.1.6 Equality and Comparison

There are several methods in the *Collection* interface that test objects for equality. For example, the methods `coll.contains(object)` and `coll.remove(object)` look for an item in the collection that is equal to `object`. However, equality is not such a simple matter. The obvious technique for testing equality -- using the == operator -- does not usually give a reasonable answer when applied to objects. The == operator tests whether two objects are identical in the sense that they share the same location in memory. Usually, however, we want to consider two objects to be equal if they represent the same value, which is a very different thing. Two values of type *String* should be considered equal if they contain the same sequence of characters. The question of whether those characters are stored in the same location in memory is irrelevant. Two values of type *Date* should be considered equal if they represent the same time.

The *Object* class defines the boolean-valued method `equals(Object)` for testing whether one object is equal to another. This method is used by many, but not by all, collection classes for deciding whether two objects are to be considered the same. In the *Object* class, `obj1.equals(obj2)` is defined to be the same as `obj1 == obj2`. However, for many sub-classes of `Object`, this definition is not reasonable, and it should be overridden. The *String* class, for example, overrides `equals()` so that for a *String* `str`, `str.equals(obj)` if `obj` is also a *String* and `obj` contains the same sequence of characters as `str`.

If you write your own class, you might want to define an `equals()` method in that class to get the correct behavior when objects are tested for equality. For example, a *Card* class that will work correctly when used in collections could be defined as:

```
public class Card {  // Class to represent playing cards.

   private int suit;  // Number from 0 to 3 that codes for the
suit --
                      // spades, diamonds, clubs or hearts.
   private int value; // Number from 1 to 13 that represents
the value.

   public boolean equals(Object obj) {
       try {
          Card other = (Card)obj;  // Type-cast obj to a Card.
          if (suit == other.suit && value == other.value) {
                // The other card has the same suit and value
as
                // this card, so they should be considered
equal.
             return true;
          }
          else
             return false;
       }
       catch (Exception e) {
              // This will catch the NullPointerException that
occurs if obj
              // is null and the ClassCastException that occurs
if obj is
              // not of type Card.  In these cases, obj is not
equal to
              // this Card, so return false.
            return false;
       }
    }

    .
```

```
        . // other methods and constructors
        .
    }
```

Without the `equals()` method in this class, methods such as `contains()` and `remove()` from the interface *Collection<Card>* will not work as expected.

A similar concern arises when items in a collection are sorted. Sorting refers to arranging a sequence of items in ascending order, according to some criterion. The problem is that there is no natural notion of ascending order for arbitrary objects. Before objects can be sorted, some method must be defined for comparing them. Objects that are meant to be compared should implement the interface `java.lang.Comparable`. In fact, *Comparable* is defined as a parameterized interface, *Comparable<T>*, which represents the ability to be compared to an object of type *T*. The interface *Comparable<T>* defines one method:

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

The value returned by `obj1.compareTo(obj2)` should be negative if and only if `obj1` comes before `obj2`, when the objects are arranged in ascending order. It should be positive if and only if `obj1` comes after `obj2`. A return value of zero means that the objects are considered to be the same for the purposes of this comparison. This does not necessarily mean that the objects are equal in the sense that `obj1.equals(obj2)` is true. For example, if the objects are of type *Address*, representing mailing addresses, it might be useful to sort the objects by zip code. Two *Addresses* are considered the same for the purposes of the sort if they have the same zip code -- but clearly that would not mean that they are the same address. (Some classes in the JFC use `compareTo()` rather than `equals()` to test objects for equality.)

The *String* class implements the interface *Comparable<String>* and defines `compareTo` in a reasonable way. In this case, the return value of `compareTo` is zero if and only if the two strings that are being compared are equal. If you define your own class and want to be able to sort objects belonging to that class, you should do the same. For example:

```
/**
 * Represents a full name consisting of a first name and a last
name.
 */
public class FullName implements Comparable<FullName> {

   private String firstName, lastName;  // Non-null first and
last names.

   public FullName(String first, String last) {  //
Constructor.
      if (first == null || last == null)
         throw new IllegalArgumentException("Names must be non-
null.");
      firstName = first;
      lastName = last;
   }
   public boolean equals(Object obj) {
      try {
         FullName other = (FullName)obj;  // Type-cast obj to
type FullName
         return firstName.equals(other.firstName)
                                      &&
lastName.equals(other.lastName);
      }
      catch (Exception e) {
         return false;  // if obj is null or is not of type
FullName
      }
   }

   public int compareTo( FullName other ) {
      if ( lastName.compareTo(other.lastName) < 0 ) {
             // If lastName comes before the last name of
             // the other object, then this FullName comes
             // before the other FullName.  Return a negative
             // value to indicate this.
         return -1;
      }
      else if ( lastName.compareTo(other.lastName) > 0 ) {
             // If lastName comes after the last name of
             // the other object, then this FullName comes
             // after the other FullName.  Return a positive
             // value to indicate this.
         return 1;
      }
      else {
             // Last names are the same, so base the comparison
on
             // the first names, using compareTo from class
String.
         return firstName.compareTo(other.firstName);
      }
   }

   .
   . // other methods
   .
}
```

(I find it a little odd that the class here is declared as "`class FullName implements Comparable<FullName>`", with "FullName" repeated as a type parameter in the name of the interface. However, it does make sense. It means that we are going to compare objects that belong to the class *FullName* to other objects **of the same type**. Even though this is the only reasonable thing to do, that fact is not obvious to the Java compiler -- and the type parameter in *Comparable<FullName>* is there for the compiler.)

There is another way to allow for comparison of objects in Java, and that is to provide a separate object that is capable of making the comparison. The object must implement the interface *Comparator<T>*, where *T* is the type of the objects that are to be compared. The interface *Comparator<T>* defines the method:

```
public int compare( T obj1, T obj2 )
```

This method compares two objects of type *T* and returns a value that is negative, or positive, or zero, depending on whether `obj1` comes before `obj2`, or comes after `obj2`, or is considered to be the same as `obj2` for the purposes of this comparison. Comparators are useful for comparing objects that do not implement the *Comparable* interface and for defining several different orderings on the same collection of objects.

In the next two sections, we'll see how *Comparable* and *Comparator* are used in the context of collections and maps.

10.1.7 Generics and Wrapper Classes

As noted in Section 7.3 the case of *ArrayList*, Java's generic programming does not apply to the primitive types, since generic data structures can only hold objects, while values of primitive type are not objects. However, the "wrapper classes" that were introduced in Subsection 7.3.2 make it possible to get around this restriction to a great extent.

Recall that each primitive type has an associated wrapper class: class *Integer* for type int, class *Boolean* for type boolean, class *Character* for type char, and so on.

An object of type *Integer* contains a value of type int. The object serves as a "wrapper" for the primitive type value, which allows it to be used in contexts where objects are required, such as in generic data structures. For example, a list of *Integers* can be stored in a variable of type *ArrayList<Integer>*, and interfaces such as *Collection<Integer>* and *Set<Integer>* are defined. Furthermore, class *Integer* defines `equals()`, `compareTo()`, and `toString()` methods that do what you would expect (that is, that compare and write out the corresponding primitive type values in the usual way). Similar remarks apply for all the wrapper classes.

Recall also that Java does automatic conversions between a primitive type and the corresponding wrapper type. (These conversions, which are called autoboxing and unboxing, were also introduced in Subsection 7.3.3.) This means that once you have created a generic data structure to hold objects belonging to one of the wrapper classes, you can use the data structure pretty much as if it actually contained primitive type values. For example, if `numbers` is a variable of type *Collection<Integer>*, it is legal to call `numbers.add(17)` or `numbers.remove(42)`. You can't literally add the primitive type value 17 to `numbers`, but Java will automatically convert the 17 to the corresponding wrapper object, `new Integer(17)`, and the wrapper object will be added to the collection. (The creation of the object does add some time and memory overhead to the operation, and you should keep that in mind in situations where efficiency is important. An array of int is more efficient than an *ArrayList<Integer>*.)

Maps

AN ARRAY OF N ELEMENTS can be thought of as a way of associating some item with each of the integers `0`, `1`, ..., `N-1`. If `i` is one of these integers, it's possible to get the item associated with `i`, and it's possible to put a new item in the `i`-th position. These "get" and "put" operations define what it means to be an array.

A map is a kind of generalized array. Like an array, a map is defined by "get" and "put" operations. But in a map, these operations are defined not for integers `0`, `1`, ..., `N-1`, but for arbitrary objects of some specified type *T*. Associated to these objects of type *T* are objects of some possibly different type *S*.

In fact, some programming languages use the term associative array instead of "map" and use the same notation for associative arrays as for regular arrays. In those languages, for example, you might see the notation `A["fred"]` used to indicate the item associated to the string "fred" in an associative array `A`. Java does not use array notation for maps, unfortunately, but the idea is the same: A map is like an array, but the indices for a map are objects, not integers. In a map, an object that serves as an "index" is called a key. The object that is associated with a key is called a value. Note that a key can have at most one associated value, but the same value can be associated to several different keys. A map can be considered to be a set of "associations," where each association is a key/value pair.

10.3.1 The Map Interface

In Java, maps are defined by the interface `java.util.Map`, which includes `put` and `get` methods as well as other general methods for working with maps. The map interface, *Map<K,V>*, is parameterized by **two** types. The first type parameter, *K*, specifies the type of objects that are possible keys in the map; the second type parameter, *V*, specifies the type of objects that are possible values in the map. For example, a map of type *Map<Date,JButton>* would associate values of type *JButton* to keys of type *Date*. For a map of type *Map<String,String>*, both the keys and the values are of type *String*.

Suppose that `map` is a variable of type *Map<K,V>* for some specific types *K* and *V*. Then the following are some of the methods that are defined for `map`:

- `map.get(key)` -- returns the object of type *V* that is associated by the map to the `key`. If the map does not associate any value with `key`, then the return value is `null`. Note that it's also possible for the return value to be `null` when the map explicitly associates the value `null` with the key. Referring to "`map.get(key)`" is similar to referring to "`A[key]`" for an array `A`. (But note that there is nothing like an *IndexOutOfBoundsException* for maps.)

- `map.put(key,value)` -- Associates the specified `value` with the specified `key`, where `key` must be of type *K* and `value` must be of type *V*. If the map already associated some other value with the key, then the new value replaces the old one. This is similar to the command "`A[key] = value`" for an array.
- `map.putAll(map2)` -- if `map2` is another map of type *Map<K,V>*, this copies all the associations from `map2` into map.
- `map.remove(key)` -- if `map` associates a value to the specified `key`, that association is removed from the map.
- `map.containsKey(key)` -- returns a boolean value that is `true` if the map associates some value to the specified `key`.
- `map.containsValue(value)` -- returns a boolean value that is `true` if the map associates the specified `value` to some key.
- `map.size()` -- returns an int that gives the number of key/value associations in the map.
- `map.isEmpty()` -- returns a boolean value that is `true` if the map is empty, that is if it contains no associations.
- `map.clear()` -- removes all associations from the map, leaving it empty.

The `put` and `get` methods are certainly the most commonly used of the methods in the *Map* interface. In many applications, these are the only methods that are needed, and in such cases a map is really no more difficult to use than a standard array.

Java includes two classes that implement the interface *Map<K,V>*: *TreeMap<K,V>* and *HashMap<K,V>*. In a *TreeMap*, the key/value associations are stored in a sorted tree, in which they are sorted according to their `keys`. For this to work, it must be possible to compare the keys to one another. This means either that the keys must implement the interface *Comparable<K>*, or that a *Comparator* must be provided for comparing keys. (The *Comparator* can be provided as a parameter to the *TreeMap* constructor.) Note that in a *TreeMap*, as in a *TreeSet*, the `compareTo()` (or `compare()`) method is used to decide whether two keys are to be considered the same. This can have undesirable consequences if the `compareTo()` method does not agree with the usual notion of equality, and you should keep this in mind when using *TreeMaps*.

A *HashMap* does not store associations in any particular order, so the keys that can be used in a *HashMap* do not have to be comparable. However, the key class should have reasonable definitions for the `equals()` method and for a `hashCode()` method that is discussed later in this section; most of Java's standard classes define these methods correctly. Most operations are a little faster on *HashMaps* than they are on *TreeMaps*. In general, you should use a *HashMap* unless you have some particular need for the ordering property of a *TreeMap*. In particular, if you are only using the `put` and `get` operations, you can safely use a *HashMap*.

Let's consider an example where maps would be useful. In Subsection 7.4.2, I presented a simple *PhoneDirectory* class that associated phone numbers with names. That class defined operations `addEntry(name,number)` and `getNumber(name)`, where both `name` and `number` are given as *Strings*. In fact, the phone directory is acting just like a map, with the `addEntry` method playing the role of the `put` operation and `getNumber` playing the role of `get`. In a real programming application, there would be no need to define a new class; we could simply use a map of type *Map<String,String>*. A directory would be defined as

```
Map<String,String> directory = new Map<String,String>();
```

and then `directory.put(name,number)` would record a phone number in the directory and `directory.get(name)` would retrieve the phone number associated with a given name.

10.3.2 Views, SubSets, and SubMaps

A *Map* is not a *Collection*, and maps do not implement all the operations defined on collections. In particular, there are no iterators for maps. Sometimes, though, it's useful to be able to iterate through all the associations in a map. Java makes this possible in a roundabout but clever way. If `map` is a variable of type *Map<K,V>*, then the method

```
map.keySet()
```

returns the set of all objects that occur as keys for associations in the map. The value returned by this method is an object that implements the interface *Set<K>*. The elements of this set are the map's keys. The obvious way to implement the `keySet()` method would be to create a new set object, add all the keys from the map, and return that set. But that's not how it's done. The value returned by `map.keySet()` is not an independent object. It is what is called a view of the actual objects that are stored in the map. This "view" of the map implements the *Set<K>* interface, but it does it in such a way that the methods defined in the interface refer directly to keys in the map. For example, if you remove a key from the view, that key -- along with its associated value -- is actually removed from the map. It's not legal to add an object to the view, since it doesn't make sense to add a key to a map without specifying the value that should be associated to the key. Since `map.keySet()` does not create a new set, it's very efficient, even for very large maps.

One of the things that you can do with a *Set* is get an *Iterator* for it and use the iterator to visit each of the elements of the set in turn. We can use an iterator for the key set of a map to traverse the map. For example, if `map` is of type *Map<String,Double>*, we could write:

```
Set<String> keys = map.keySet();     // The set of keys in the
map.
Iterator<String> keyIter = keys.iterator();
System.out.println("The map contains the following
associations:");
while (keyIter.hasNext()) {
   String key = keyIter.next();  // Get the next key.
   Double value = map.get(key);  // Get the value for that key.
   System.out.println( "   (" + key + "," + value + ")" );
}
```

Or we could do the same thing more easily, avoiding the explicit use of an iterator, with a for-each loop:

```
System.out.println("The map contains the following
associations:");
for ( String key : map.keySet() ) {  // "for each key in the
map's key set"
    Double value = map.get(key);
    System.out.println( "   (" + key + "," + value + ")" );
}
```

If the map is a *TreeMap*, then the key set of the map is a sorted set, and the iterator will visit the keys in ascending order. For a *HashMap*, the keys are visited in an arbitrary, unpredictable order.

The *Map* interface defines two other views. If `map` is a variable of type *Map<K,V>*, then the method:

```
map.values()
```

returns an object of type *Collection<V>* that contains all the values from the associations that are stored in the map. The return value is a *Collection* rather than a *Set* because it can contain duplicate elements (since a map can associate the same value to any number of keys). The method:

```
map.entrySet()
```

returns a set that contains all the associations from the map. The elements in the set are objects of type *Map.Entry<K,V>*. *Map.Entry<K,V>* is defined as a static nested interface inside the interface *Map<K,V>*, so its full name contains a period. However, the name can be used in the same way as any other type name. (The return type of the method `map.entrySet()` is written as *Set<Map.Entry<K,V>>*. The type parameter in this case is itself a parameterized type. Although this might look confusing, it's just Java's way of saying that the elements of the set are of type *Map.Entry<K,V>*.) The information in the set returned by `map.entrySet()` is actually no different from the information in the map itself, but the set provides a different view of this information, with different operations. Each `Map.Entry` object contains one key/value pair, and defines methods `getKey()` and `getValue()` for retrieving the key and the value. There is also a method, `setValue(value)`, for setting the value; calling this method for a `Map.Entry` object will modify the map itself, just as if the map's `put` method were called. As an example, we can use the entry set of a map to print all the key/value pairs in the map. This is more efficient than using the key set to print the same information, as I did in the above example, since we don't have to use the `get()` method to look up the value associated with each key. Suppose again that `map` is of type *Map<String,Double>*. Then we can write:

```
Set<Map.Entry<String,Double>> entries = map.entrySet();
Iterator<Map.Entry<String,Double>> entryIter =
entries.iterator();
System.out.println("The map contains the following
associations:");
```

```
while (entryIter.hasNext()) {
   Map.Entry<String,Double> entry = entryIter.next();
   String key = entry.getKey();  // Get the key from the entry.
   Double value = entry.getValue();  // Get the value.
   System.out.println( "    (" + key + "," + value + ")" );
}
```

or, using a for-each loop:

```
System.out.println("The map contains the following
associations:");
for ( Map.Entry<String,Double> entry : map.entrySet() ) {
   System.out.println( "    (" + entry.getKey() + "," +
entry.getValue() + ")" );
}
```

Maps are not the only place in Java's generic programming framework where views are used. For example, the interface *List<T>* defines a sublist as a view of a part of a list. If `list` implements the interface *List<T>*, then the method

```
list.subList( fromIndex, toIndex )
```

where `fromIndex` and `toIndex` are integers, returns a view of the part of the list consisting of the list elements in positions between `fromIndex` and `toIndex` (including `fromIndex` but excluding `toIndex`). This view lets you operate on the sublist using any of the operations defined for lists, but the sublist is not an independent list. Changes made to the sublist are actually made to the original list.

Similarly, it is possible to obtain views that represent certain subsets of a sorted set. If `set` is of type *TreeSet<T>*, then `set.subSet(fromElement,toElement)` returns a *Set<T>* that contains all the elements of `set` that are between `fromElement` and `toElement` (including `fromElement` and excluding `toElement`). The parameters `fromElement` and `toElement` must be objects of type *T*. For example, if `words` is a set of type *TreeSet<String>* in which all the elements are strings of lower case letters, then `words.subSet("m","n")` contains all the elements of `words` that begin with the letter 'm'. This subset is a view of part of the original set. That is, creating the subset does not involve copying elements. And changes made to the subset, such as adding or removing elements, are actually made to the original set. The view `set.headSet(toElement)` consists of all elements from the set which are strictly less than `toElement`, and `set.tailSet(fromElement)` is a view that contains all elements from the set that are greater than or equal to `fromElement`.

The class *TreeMap<K,V>* defines three submap views. A submap is similar to a subset. A submap is a *Map* that contains a subset of the keys from the original *Map*, along with their associated values. If `map` is a variable of type *TreeMap<K,V>*, and if `fromKey` and `toKey` are of type *T*, then `map.subMap(fromKey,toKey)` returns a view that contains all key/value pairs from `map` whose keys are between `fromKey` and `toKey` (including `fromKey` and excluding `toKey`). There are also views `map.headMap(toKey)` and `map.tailMap(fromKey)` which are defined analogously to `headSet` and `tailSet`. Suppose, for example, that `blackBook` is a map of type *TreeMap<String,String>* in which the keys are names and the values are phone numbers. We can print out all the entries from `blackBook` where the name begins with "M" as follows:

```
Map<String,String> ems = blackBook.subMap("M","N");
     // This submap contains entries for which the key is
greater
     // than or equal to "M" and strictly less than "N".

if (ems.isEmpty()) {
   System.out.println("No entries beginning with M.");
}
else {
   System.out.println("Entries beginning with M:");
   for ( Map.Entry<String,String> entry : ems.entrySet() )
      System.out.println( "   " + entry.getKey() + ": " +
entry.getValue() );
}
```

Subsets and submaps are probably best thought of as generalized search operations that make it possible to find all the items in a range of values, rather than just to find a single value. Suppose, for example that a database of scheduled events is stored in a map of type *TreeMap<Date,Event>* in which the keys are the times of the events, and suppose you want a listing of all events that are scheduled for some time on July 4, 2014. Just make a submap containing all keys in the range from 12:00 AM, July 4, 2014 to 12:00 AM, July 5, 2014, and output all the entries from that submap. This type of search, which is known as a subrange query is quite common.

10.3.3 Hash Tables and Hash Codes

HashSets and *HashMaps* are implemented using a data structure known as a hash table. You don't need to understand hash tables to use *HashSets* or *HashMaps*, but any computer programmer should be familiar with hash tables and how they work.

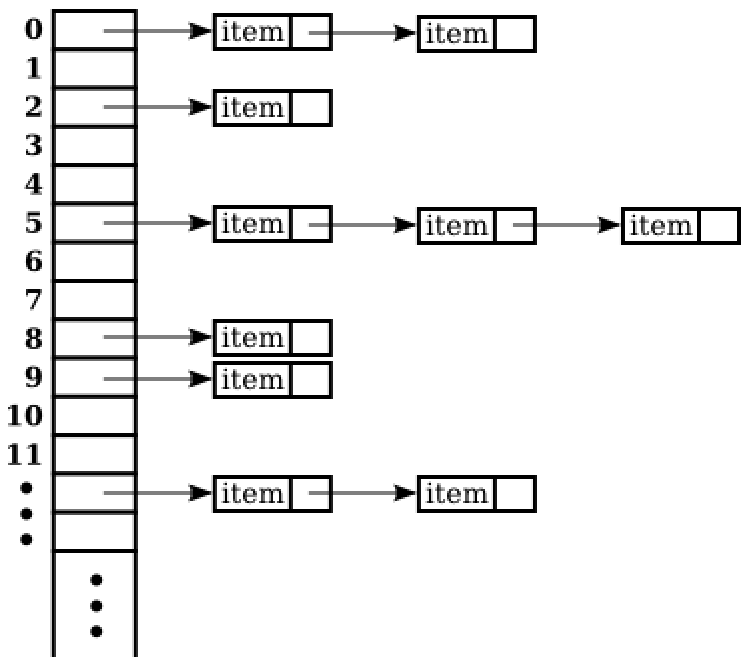
Hash tables are an elegant solution to the search problem. A hash table, like a *HashMap*, stores key/value pairs. Given a key, you have to search the table for the corresponding key/value pair. When a hash table is used to implement a set, there are no values, and the only question is whether or not the key occurs in the set. You still have to search for the key to check whether it is there or not.

In most search algorithms, in order to find the item you are interested in, you have to look through a bunch of other items that don't interest you. To find something in an unsorted list, you have to go though the items one-by-one until you come to the one you are looking for. In a binary sort tree, you have to start at the root and move down the tree until you find the item you want. When you search for a key/value pair in a hash table, you can go directly to the location that contains the item you want. You don't have to look through any other items. (This is not quite true, but it's close.) The location of the key/value pair is computed from the key: You just look at the key, and then you go directly to the location where it is stored.

How can this work? If the keys were integers in the range 0 to 99, we could store the key/value pairs in an array, `A`, of 100 elements. The key/value pair with key `K` would be stored in `A[K]`. The key takes us directly to the location of the key/value pair. The problem is that there are usually far too many different possible keys for us to be able to use an array with one location for each possible key. For example, if the key can be any value of type int, then we would need an array with over four billion locations -- quite a waste of space if we are only going to store, say, a few thousand items! If the key can be a string of any length, then the number of possible keys is infinite, and using an array with one location for each possible key is simply impossible.

Nevertheless, hash tables store their data in an array, and the array index where a key is stored is based on the key. The index is not equal to the key, but it is computed from the key. The array index for a key is called the hash code for that key. A function that computes a hash code, given a key, is called a hash function. To find a key in a hash table, you just have to compute the hash code of the key and go directly to the array location given by that hash code. If the hash code is 17, look in array location number 17.

Now, since there are fewer array locations than there are possible keys, it's possible that we might try to store two or more keys in the same array location. This is called a collision. A collision is not an error. We can't reject a key just because another key happened to have the same hash code. A hash table must be able to handle collisions in some reasonable way. In the type of hash table that is used in Java, each array location actually holds a linked list of key/value pairs (possibly an empty list). When two items have the same hash code, they are in the same linked list. The structure of the hash table looks something like this:



In this diagram, there are two items with hash code 0, no items with hash code 1, one item with hash code 2, and so on. In a properly designed hash table, most of the linked lists are of length zero or one, and the average length of the lists is less than one. Although the hash code of a key doesn't necessarily take you directly to that key, there are probably no more than one or two other items that you have to look through before finding the key you want. For this to work properly, the number of items in the hash table should be somewhat less than the number of locations in the array. In Java's implementation, whenever the number of items exceeds 75% of the array size, the array is replaced by a larger one and all the items in the old array are inserted into the new one. (This is why adding one new item will sometimes cause the ordering of all the items in the hash table to change completely.)

There is still the question of where hash codes come from. Every object in Java has a hash code. The *Object* class defines the method `hashCode()`, which returns a value of type int. When an object, `obj`, is stored in a hash table that has `N` locations, a hash code in the range `0` to `N-1` is needed. This hash code is computed as `Math.abs(obj.hashCode()) % N`, the remainder when the absolute value of `obj.hashCode()` is divided by `N`. (The `Math.abs` is necessary because `obj.hashCode()` can be a negative integer, and we need a non-negative number to use as an array index.)

For hashing to work properly, two objects that are equal according to the `equals()` method must have the same hash code. In the *Object* class, this condition is satisfied because both `equals()` and `hashCode()` are based on the address of the memory location where the object is stored. However, as noted in Subsection 10.1.6, many classes redefine the `equals()` method. If a class redefines the `equals()` method, and if objects of that class will be used as keys in hash tables, then the class must also redefine the `hashCode()` method. For example, in the *String* class, the `equals()` method is redefined so that two objects of type *String* are considered to be equal if they contain the same sequence of characters. The `hashCode()` method is also redefined in the *String* class, so that the hash code of a string is computed from the characters in that string rather than from its location in memory. For Java's standard classes, you can expect `equals()` and `hashCode()` to be correctly defined. However, you might need to define these methods in classes that you write yourself.

Writing a good hash function is something of an art. In order to work well, the hash function must spread the possible keys fairly evenly over the hash table. Otherwise, the items in a table can be concentrated in a subset of the available locations, and the linked lists at those locations can grow to large size; that would destroy the efficiency that is the major reason for hash tables to exist in the first place

10.4.1 Symbol Tables

We begin with a straightforward but important application of maps. When a compiler reads the source code of a program, it encounters definitions of variables, subroutines, and classes. The names of these things can be used later in the program. The compiler has to remember the definition of each name, so that it can recognize the name and apply the definition when the name is encountered later in the program. This is a natural application for a *Map*. The name can be used as a key in the map. The value associated to the key is the definition of the name, encoded somehow as an object. A map that is used in this way is called a symbol table.

In a compiler, the values in a symbol table can be quite complicated, since the compiler has to deal with names for various sorts of things, and it needs a different type of information for each different type of name. We will keep things simple by looking at a symbol table in another context. Suppose that we want a program that can evaluate expressions entered by the user, and suppose that the expressions can contain variables, in addition to operators, numbers, and parentheses. For this to make sense, we need some way of assigning values to variables. When a variable is used in an expression, we need to retrieve the variable's value. A symbol table can be used to store the data that we need. The keys for the symbol table are variable names. The value associated with a key is the value of that variable, which is of type double. The symbol table will be an object of type *Map<String,Double>*. (Remember that primitive types such as double can't be used as type parameters; a wrapper class such as *Double* must be used instead. See Subsection 10.1.7.)

To demonstrate the idea, we'll use a rather simple-minded program in which the user types commands such as:

```
let x = 3 + 12
print 2 + 2
print 10*x +17
let rate = 0.06
print 1000*(1+rate)
```

The program is an interpreter for a very simple language. The only two commands that the program understands are "print" and "let". When a "print" command is executed, the computer evaluates the expression and displays the value. If the expression contains a variable, the computer has to look up the value of that variable in the symbol table. A "let" command is used to give a value to a variable. The computer has to store the value of the variable in the symbol table. (Note: The "variables" I am talking about here are not variables in the Java program. The Java program is executing a sort of program typed in by the user. I am talking about variables in the user's program. The user gets to make up variable names, so there is no way for the Java program to know in advance what the variables will be.)

In Subsection 9.5.2, we saw how to write a program, *SimpleParser2.java*, that can evaluate expressions that do not contain variables. Here, I will discuss another example program, *SimpleInterpreter.java*, that is based on the older program. I will only talk about the parts that are relevant to the symbol table.

The program uses a *HashMap* as the symbol table. A *TreeMap* could also be used, but since the program does not need to access the variables in alphabetical order, we don't need to have the keys stored in sorted order. The symbol table in the program is represented by a variable named `symbolTable` of type *HashMap<String,Double>*. At the beginning of the program, the symbol table object is created with the command:

```
symbolTable = new HashMap<String,Double>();
```

This creates a map that initially contains no key/value associations. To execute a "let" command, the program uses the symbol table's `put()` method to associate a value with the variable name. Suppose that the name of the variable is given by a *String*, `varName`, and the value of the variable is stored in a variable `val` of type double. The following command would then set the value associated with the variable in the symbol table:

```
symbolTable.put( varName, val );
```

In the program *SimpleInterpreter.java*, you'll find this in the method named `doLetCommand()`. The actual value that is stored in the symbol table is an object of type *Double*. We can use the double value `val` in the call to `put` because Java does an automatic conversion of type double to *Double* when necessary. The double value is "wrapped" in an object of type *Double*, so that, in effect, the above statement is equivalent to

```
symbolTable.put( varName, new Double(val) );
```

Just for fun, I decided to pre-define two variables named "pi" and "e" whose values are the usual mathematical constants π and *e*. In Java, the values of these constants are given by `Math.PI` and `Math.E`. To make these variables available to the user of the program, they are added to the symbol table with the commands:

Writing Generic Classes and Methods

SO FAR IN THIS CHAPTER, you have learned about using the generic classes and methods that are part of the Java Collection Framework. Now, it's time to learn how to write new generic classes and methods from scratch. Generic programming produces highly general and reusable code -- it's very useful for people who write reusable software libraries to know how to do generic programming, since it enables them to write code that can be used in many different situations. Not every programmer needs to write reusable software libraries, but every programmer should know at least a little about how to do it. In fact, just to read the Javadoc documentation for Java's standard generic classes, you need to know some of the syntax that is introduced in this section.

I will not cover every detail of generic programming in Java in this section, but the material presented here should be sufficient to cover the most common cases.

10.5.1 Simple Generic Classes

Let's start with an example that illustrates the motivation for generic programming. In Subsection 10.2.1, I remarked that it would be easy to use a *LinkedList* to implement a queue. (Queues were introduced in Subsection 9.3.2.) To ensure that the only operations that are performed on the list are the queue operations `enqueue`, `dequeue`, and `isEmpty`, we can create a new class that contains the linked list as a private instance variable. To implement queues of strings, for example, we can define the class:

```
class QueueOfStrings {
   private LinkedList<String> items = new LinkedList<String>();
   public void enqueue(String item) {
      items.addLast(item);
   }
   public String dequeue() {
      return items.removeFirst();
   }
   public boolean isEmpty() {
      return (items.size() == 0);
   }
}
```

This is a fine and useful class. But, if this is how we write queue classes, and if we want queues of *Integers* or *Doubles* or *JButtons* or any other type, then we will have to write a different class for each type. The code for all of these classes will be almost identical, which seems like a lot of redundant programming. To avoid the redundancy, we can write a **generic** *Queue* class that can be used to define queues of any type of object.

The syntax for writing the generic class is straightforward: We replace the specific type *String* with a type parameter such as *T*, and we add the type parameter to the name of the class:

```
class Queue<T> {
   private LinkedList<T> items = new LinkedList<T>();
   public void enqueue(T item) {
      items.addLast(item);
   }
   public T dequeue() {
      return items.removeFirst();
   }
   public boolean isEmpty() {
      return (items.size() == 0);
   }
}
```

Note that within the class, the type parameter *T* is used just like any regular type name. It's used to declare the return type for `dequeue`, as the type of the formal parameter `item` in `enqueue`, and even as the actual type parameter in *LinkedList<T>*. Given this class definition, we can use parameterized types such as *Queue<String>* and *Queue<Integer>* and *Queue<JButton>*. That is, the *Queue* class is used in exactly the same way as built-in generic classes like *LinkedList* and *HashSet*.

Note that you don't have to use "T" as the name of the type parameter in the definition of the generic class. Type parameters are like formal parameters in subroutines. You can make up any name you like in the **definition** of the class. The name in the definition will be replaced by an actual type name when the class is used to declare variables or create objects. If you prefer to use a more meaningful name for the type parameter, you might define the *Queue* class as:

```
class Queue<ItemType> {
   private LinkedList<ItemType> items = new LinkedList<ItemType>();
   public void enqueue(ItemType item) {
      items.addLast(item);
   }
   public ItemType dequeue() {
      return items.removeFirst();
   }
   public boolean isEmpty() {
      return (items.size() == 0);
   }
}
```

Changing the name from "`T`" to "`ItemType`" has absolutely no effect on the meaning of the class definition or on the way that *Queue* is used.

Generic interfaces can be defined in a similar way. It's also easy to define generic classes and interfaces that have two or more type parameters, as is done with the standard interface *Map<K,V>*. A typical example is the definition of a "Pair" that contains two objects, possibly of different types. A simple version of such a class can be defined as:

```
class Pair<T,S> {
   public T first;
   public S second;
   public Pair( T a, S b ) {  // Constructor.
      first = a;
      second = b;
   }
}
```

This class can be used to declare variables and create objects such as:

```
Pair<String,Color> colorName = new Pair<String,Color>("Red",
Color.RED);
Pair<Double,Double> coordinates = new
Pair<Double,Double>(17.3,42.8);
```

Note that in the definition of the constructor in this class, the name "`Pair`" does **not** have type parameters. You might have expected "`Pair<T,S>`". However, the name of the class is "`Pair`", not "`Pair<T,S>`", and within the definition of the class, "`T`" and "`S`" are used as if they are the names of specific, actual types. Note in any case that type parameters are **never** added to the names of methods or constructors, only to the names of classes and interfaces.

10.5.2 Simple Generic Methods

In addition to generic classes, Java also has generic methods. An example is the method `Collections.sort()`, which can sort collections of objects of any type. To see how to write generic methods, let's start with a non-generic method for counting the number of times that a given string occurs in an array of strings:

```
/**
 * Returns the number of times that itemToCount occurs in list.
Items in the
 * list are tested for equality using itemToCount.equals(), except
in the
 * special case where itemToCount is null.
 */
public static int countOccurrences(String[] list, String
itemToCount) {
   int count = 0;
   if (itemToCount == null) {
      for ( String listItem : list )
         if (listItem == null)
            count++;
   }
   else {
      for ( String listItem : list )
         if (itemToCount.equals(listItem))
            count++;
   }
   return count;
}
```

Once again, we have some code that works for type *String*, and we can imagine writing almost identical code to work with other types of objects. By writing a generic method, we get to write a single method definition that will work for objects of any type. We need to replace the specific type *String* in the definition of the method with the name of a type parameter, such as *T*. However, if that's the only change we make, the compiler will think that "T" is the name of an actual type, and it will mark it as an undeclared identifier. We need some way of telling the compiler that "T" is a type parameter. That's what the "`<T>`" does in the definition of the generic class "`class Queue<T> { ...`". For a generic method, the "`<T>`" goes just before the name of the return type of the method:

```
public static <T> int countOccurrences(T[] list, T itemToCount) {
   int count = 0;
   if (itemToCount == null) {
      for ( T listItem : list )
         if (listItem == null)
            count++;
   }
   else {
      for ( T listItem : list )
         if (itemToCount.equals(listItem))
            count++;
   }
   return count;
}
```

The "`<T>`" marks the method as being generic and specifies the name of the type parameter that will be used in the definition. Of course, the name of the type parameter doesn't have to be "T"; it can be anything. (The "`<T>`" looks a little strange in that position, I know, but it had to go somewhere and that's just where the designers of Java decided to put it.)

Given the generic method definition, we can apply it to objects of any type. If `wordList` is a variable of type `String[]` and `word` is a variable of type *String*, then

```
int ct = countOccurrences( wordList, word );
```

will count the number of times that `word` occurs in `wordList`. If `palette` is a variable of type `Color[]` and `color` is a variable of type *Color*, then

```
int ct = countOccurrences( palette, color );
```

will count the number of times that `color` occurs in `palette`. If `numbers` is a variable of type `Integer[]`, then

```
int ct = countOccurrences( numbers, 17 );
```

will count the number of times that `17` occurs in `numbers`. This last example uses autoboxing; the 17 is automatically converted to a value of type *Integer*. Note that, since generic programming in Java applies only to objects, we **cannot** use `countOccurrences` to count the number of occurrences of 17 in an array of type `int[]`.

A generic method can have one or more type parameters, such as the "T" in `countOccurrences`. Note that when a generic method is used, as in the function call "`countOccurrences(wordlist, word)`", there is no explicit mention of the type that is substituted for the type parameter. The compiler deduces the type from the types of the actual parameters in the method call. Since `wordlist` is of type `String[]`, the compiler can tell that in "`countOccurrences(wordlist, word)`", the type that replaces *T* is *String*. This contrasts with the use of generic classes, as in "`new Queue<String>()`", where the type parameter is specified explicitly.

The `countOccurrences` method operates on an array. We could also write a similar method to count occurrences of an object in any collection:

```
public static <T> int countOccurrences(Collection<T> collection, T
itemToCount) {
   int count = 0;
   if (itemToCount == null) {
      for ( T item : collection )
         if (item == null)
            count++;
   }
   else {
      for ( T item : collection )
         if (itemToCount.equals(item))
            count++;
   }
```

```
      return count;
   }
```

Since *Collection<T>* is itself a generic type, this method is very general. It can operate on an *ArrayList* of *Integers*, a *TreeSet* of *Strings*, a *LinkedList* of *JButtons*, ...

10.5.3 Type Wildcards

There is a limitation on the sort of generic classes and methods that we have looked at so far: The type parameter in our examples, usually named *T*, can be any type at all. This is OK in many cases, but it means that the only things that you can do with *T* are things that can be done with **every** type, and the only things that you can do with objects of type *T* are things that you can do with **every** object. With the techniques that we have covered so far, you can't, for example, write a generic method that compares objects with the `compareTo()` method, since that method is not defined for all objects. The `compareTo()` method is defined in the *Comparable* interface. What we need is a way of specifying that a generic class or method only applies to objects of type *Comparable* and not to arbitrary objects. With that restriction, we should be free to use `compareTo()` in the definition of the generic class or method.

There are two different but related syntaxes for putting restrictions on the types that are used in generic programming. One of these is bounded type parameters, which are used as formal type parameters in generic class and method definitions; a bounded type parameter would be used in place of the simple type parameter *T* in "`class GenericClass<T> ...`" or in "`public static< T> void genericMethod(...`". The second syntax is wildcard types, which are used as type parameters in the declarations of variables and of formal parameters in method definitions; a wildcard type could be used in place of the type parameter *String* in the declaration statement "`List<String> list;`" or in the formal parameter list "`void concat(Collection<String> c)`". We will look at wildcard types first, and we will return to the topic of bounded types later in this section.

Let's start with a simple example in which a wildcard type is useful. Suppose that *Shape* is a class that defines a method `public void draw()`, and suppose that *Shape* has subclasses such as *Rect* and *Oval*. Suppose that we want a method that can draw all the shapes in a collection of *Shapes*. We might try:

```
   public static void drawAll(Collection<Shape> shapes) {
      for ( Shape s : shapes )
         s.draw();
   }
```

This method works fine if we apply it to a variable of type *Collection<Shape>*, or *ArrayList<Shape>*, or any other collection class with type parameter *Shape*. Suppose, however, that you have a list of *Rects* stored in a variable named `rectangles` of type *Collection<Rect>*. Since *Rects* are *Shapes*, you might expect to be able to call `drawAll(rectangles).`

Unfortunately, this will not work; a collection of *Rects* is **not** considered to be a collection of *Shapes*! The variable `rectangles` cannot be assigned to the formal parameter `shapes`. The solution is to replace the type parameter "`Shape`" in the declaration of `shapes` with the wildcard type "`? extends Shape`":

```
public static void drawAll(Collection<? extends Shape> shapes) {
   for ( Shape s : shapes )
      s.draw();
}
```

The wildcard type "`? extends Shape`" means roughly "any type that is either equal to *Shape* or that is a subclass of *Shape*". When the parameter `shapes` is declared to be of type *Collection<? extends Shape>*, it becomes possible to call the `drawAll` method with an actual parameter of type *Collection<Rect>* since *Rect* is a subclass of *Shape* and therefore matches the wildcard. We could also pass actual parameters to `drawAll` of type *ArrayList<Rect>* or *Set<Oval>* or *List<Oval>*. And we can still pass variables of type *Collection<Shape>* or *ArrayList<Shape>*, since the class *Shape* itself matches "`? extends Shape`". We have greatly increased the usefulness of the method by using the wildcard type.

(Although it is not essential, you might be interested in knowing *why* Java does not allow a collection of *Rects* to be used as a collection of *Shapes*, even though every *Rect* is considered to be a *Shape*. Consider the rather silly but legal method that adds an oval to a list of shapes:

```
static void addOval(List<Shape> shapes, Oval oval) {
   shapes.add(oval);
}
```

Suppose that `rectangles` is of type *List<Rect>*. It's illegal to call `addOval(rectangles,oval)`, because of the rule that a list of *Rects* is not a list of *Shapes*. If we dropped that rule, then `addOval(rectangles,oval)` would be legal, and it would add an *Oval* to a list of *Rects*. This would be bad: Since *Oval* is not a subclass of *Rect*, an *Oval* is **not** a *Rect*, and a list of *Rects* should never be able to contain an *Oval*. The method call `addOval(rectangles,oval)` does not make sense and **should** be illegal, so the rule that a collection of *Rects* is not a collection of *Shapes* is a good rule.)

As another example, consider the method `addAll()` from the interface *Collection<T>*. In my description of this method in Subsection 10.1.4, I say that for a collection, `coll`, of type *Collection<T>*, `coll.addAll(coll2)` "adds all the objects in `coll2` to `coll`. The parameter, `coll2`, can be any collection of type *Collection<T>*. However, it can also be more general. For example, if *T* is a class and *S* is a sub-class of *T*, then `coll2` can be of type *Collection<S>*. This makes sense because any object of type *S* is automatically of type *T* and so can legally be added to `coll`." If you think for a moment, you'll see that what I'm describing here, a little awkwardly, is a use of wildcard types: We don't want to require `coll2` to be a collection of objects of type *T*; we want to allow collections of any subclass of *T*. To be more specific, let's look at how a similar `addAll()` method could be added to the generic *Queue* class that was defined earlier in this section:

```
class Queue<T> {
   private LinkedList<T> items = new LinkedList<T>();
   public void enqueue(T item) {
      items.addLast(item);
   }
   public T dequeue() {
      return items.removeFirst();
   }
   public boolean isEmpty() {
      return (items.size() == 0);
   }
   public void addAll(Collection<? extends T> collection) {
         // Add all the items from the collection to the end of the
queue
      for ( T item : collection )
         enqueue(item);
   }
}
```

Here, *T* is a type parameter in the generic class definition. We are combining wildcard types with generic classes. Inside the generic class definition, "`T`" is used as if it is a specific, though unknown, type. The wildcard type "`? extends T`" means some type that is equal to or extends that specific type. When we create a queue of type *Queue<Shape>*, "`T`" refers to "Shape", and the wildcard type "`? extends T`" in the class definition means "`? extends Shape`", meaning that the `addAll` method of the queue can be applied to collections of *Rects* and *Ovals* as well as to collections of *Shapes*.

The for-each loop in the definition of `addAll` iterates through the `collection` using a variable, `item`, of type *T*. Now, `collection` can be of type *Collection<S>*, where *S* is a subclass of *T*. Since `item` is of type *T*, not *S*, do we have a problem here? No, no problem. As long as *S* is a subclass of *T*, a value of type *S* can be assigned to a variable of type *T*. The restriction on the wildcard type makes everything work nicely.

The `addAll` method adds all the items from a collection to the queue. Suppose that we wanted to do the opposite: Add all the items that are currently on the queue to a given collection. An instance method defined as

```
public void addAllTo(Collection<T> collection)
```

would only work for collections whose base type is exactly the same as *T*. This is too restrictive. We need some sort of wildcard. However, "`? extends T`" won't work. Suppose we try it:

```
public void addAllTo(Collection<? extends T> collection) {
      // Remove all items currently on the queue and add them to
collection
   while ( ! isEmpty() ) {
      T item = dequeue();  // Remove an item from the queue.
      collection.add( item );  // Add it to the collection.
ILLEGAL!!
   }
}
```

The problem is that we can't add an `item` of type *T* to a collection that might only be able to hold items belonging to some subclass, *S*, of *T*. The containment is going in the wrong direction: An `item` of type *T* is not necessarily of type *S*. For example, if we have a queue of type *Queue<Shape>*, it doesn't make sense to add items from the queue to a collection of type *Collection<Rect>*, since not every *Shape* is a *Rect*. On the other hand, if we have a *Queue<Rect>*, it would make sense to add items from that queue to a *Collection<Shape>* or indeed to any collection *Collection<S>* where *S* is a **super**class of *Rect*.

To express this type of relationship, we need a new kind of type wildcard: "`? super T`". This wildcard means, roughly, "either *T* itself or any class that is a superclass of *T*." For example, *Collection<? super Rect>* would match the types *Collection<Shape>*, *ArrayList<Object>*, and *Set<Rect>*. This is what we need for our `addAllTo` method. With this change, our complete generic queue class becomes:

```
class Queue<T> {
   private LinkedList<T> items = new LinkedList<T>();
   public void enqueue(T item) {
      items.addLast(item);
   }
   public T dequeue() {
      return items.removeFirst();
   }
   public boolean isEmpty() {
      return (items.size() == 0);
   }
   public void addAll(Collection<? extends T> collection) {
         // Add all the items from the collection to the end of the
queue
      for ( T item : collection )
         enqueue(item);
   }
   public void addAllTo(Collection<? super T> collection) {
         // Remove all items currently on the queue and add them to
collection
      while ( ! isEmpty() ) {
         T item = dequeue();  // Remove an item from the queue.
         collection.add( item );  // Add it to the collection.
      }
   }
}
```

In a wildcard type such as "`? extends T`", *T* can be an `interface` instead of a class. Note that the term "`extends`" (not "`implements`") is used in the wildcard type, even if *T* is an interface. For example, we will see that *Runnable* is an `interface` that defines the method `public void run()`. (Runnable objects are usually associated with threads; see Chapter 12.) Here is a method that runs all the objects in a collection of *Runnables* by executing the `run()` method from each runnable object:

```
public static runAll( Collection<? extends Runnable> runnables ) {
   for ( Runnable runnable : runnables ) {
      runnable.run();
```

```
    }
}
```

Wildcard types are used **only** as type parameters in parameterized types, such as *Collection<? extends Runnable>*. The place where a wildcard type is most likely to occur, by far, is in a formal parameter list, where the wildcard type is used in the declaration of the type of a formal parameter. However, they can also be used in a few other places. For example, they can be used in the type specification in a variable declaration statement.

One final remark: The wildcard type "`<?>`" is equivalent to "`<? extends Object>`". That is, it matches any possible type. For example, the `removeAll()` method in the generic interface *Collections<T>* is declared as

```
public boolean removeAll( Collection<?> c ) { ...
```

This just means that the `removeAll` method can be applied to any collection of any type of object.

10.5.4 Bounded Types

Wildcard types don't solve all of our problems. They allow us to generalize method definitions so that they can work with collections of objects of various types, rather than just a single type. However, they do not allow us to restrict the types that are allowed as type parameters in a generic class or method definition. Bounded types exist for this purpose.

We start with a small, not very realistic example. Suppose that you would like to create groups of GUI components using a generic class named *ComponentGroup*. For example, the parameterized type *ComponentGroup<JButton>* would represent a group of *JButtons*, while *ComponentGroup<JPanel>* would represent a group of *JPanels*. The class will include methods that can be called to apply certain operations to all components in the group at once. For example, there will be an instance method of the form

```
public void repaintAll() {
    .
    .  // Call the repaint() method of every component in the group.
    .
}
```

The problem is that the `repaint()` method is defined in a *JComponent* object, but not for objects of arbitrary type. It wouldn't make sense to allow types such as *ComponentGroup<String>* or *ComponentGroup<Integer>*, since *Strings* and *Integers* don't have `repaint()` methods. We need some way to restrict the type parameter *T* in *ComponentGroup<T>* so that only *JComponent* and subclasses of *JComponent* are allowed as actual type parameters. We can do this by using the bounded type "`T extends JComponent`" instead of a plain "`T`" in the definition of the class:

```
public class ComponentGroup<T extends JComponent> {
   private ArrayList<T> components; // For storing the components
in this group.
   public void repaintAll() {
      for ( JComponent c : components )
         if (c != null)
            c.repaint();
   }
   public void setAllEnabled( boolean enable ) {
      for ( JComponent c : components )
         if (c != null)
            c.setEnabled(enable);
      }
   }
   public void add( T c ) {  // Add a value c, of type T, to the
group.
      components.add(c);
   }
   .
   .  // Additional methods and constructors.
   .
}
```

The restriction "`extends JComponent`" on `T` makes it illegal to create the parameterized types *ComponentGroup<String>* and *ComponentGroup<Integer>*, since the actual type parameter that replaces "`T`" is required to be either *JComponent* itself or a subclass of *JComponent*. With this restriction, we know -- and, more important, the compiler knows -- that the objects in the group are of type *JComponent* and the operations `c.repaint()` and `c.setEnabled()` are defined for any `c` in the group.

In general, a bounded type parameter "`T extends SomeType`" means roughly "a type, *T*, that is either equal to *SomeType* or is a subclass of *SomeType*", and the upshot is that any object of type *T* is also of type *SomeType*, and any operation that is defined for objects of type *SomeType* is defined for objects of type *T*. The type *SomeType* doesn't have to be the name of a class. It can be any name that represents an actual object type. For example, it can be an `interface` or even a parameterized type.

Bounded types and wildcard types are clearly related. They are, however, used in very different ways. A bounded type can be used only as a formal type parameter in the definition of a generic method, class, or interface. A wildcard type is used most often to declare the type of a formal parameter in a method and cannot be used as a formal type parameter. One other difference, by the way, is that, in contrast to wildcard types, bounded type parameters can only use "`extends`", never "`super`".

Bounded type parameters can be used when declaring generic methods. For example, as an alternative to the generic *ComponentGroup* class, one could write a free-standing generic `static` method that can repaint any collection of *JComponents* as follows:

```
public static <T extends JComponent> void repaintAll(Collection<T>
comps) {
   for ( JComponent c : comps )
      if (c != null)
         c.repaint();
}
```

Using "`<T extends JComponent>`" as the formal type parameter means that the method can only be called for collections whose base type is *JComponent* or some subclass of *JComponent*. Thus, it is legal to call `repaintAll(coll)` where `coll` is of type *List<JPanel>* but not where `coll` is of type *Set<String>*.

Note that we don't really need a generic type parameter in this case. We can write an equivalent method using a wildcard type:

```
public static void repaintAll(Collection<? extends JComponent>
comps) {
   for ( JComponent c : comps )
      if (c != null)
         c.repaint();
}
```

In this situation, the version that uses the wildcard type is to be preferred, since the implementation is simpler. However, there are some situations where a generic method with a bounded type parameter cannot be rewritten using a wildcard type. Note that a generic type parameter gives a name, such as *T*, to the unknown type, while a wildcard type does not give a name to the unknown type. The name makes it possible to refer to the unknown type in the body of the method that is being defined. If a generic method definition uses the generic type name more than once or uses it outside the formal parameter list of the method, then the generic type cannot be replaced with a wildcard type.

Let's look at a generic method in which a bounded type parameter is essential. In Subsection 10.2.1, I presented a code segment for inserting a string into a sorted list of strings, in such a way that the modified list is still in sorted order. Here is the same code, but this time in the form of a method definition (and without the comments):

```
static void sortedInsert(List<String> sortedList, String newItem) {
   ListIterator<String> iter = sortedList.listIterator();
   while (iter.hasNext()) {
      String item = iter.next();
      if (newItem.compareTo(item) <= 0) {
         iter.previous();
         break;
      }
   }
   iter.add(newItem);
}
```

This method works fine for lists of strings, but it would be nice to have a generic method that can be applied to lists of other types of objects. The problem, of course, is that the code assumes that the `compareTo()` method is defined for objects in the list, so the method can only work for lists of objects that implement the *Comparable* interface. We can't simply use a wildcard type to enforce this restriction. Suppose we try to do it, by replacing *List<String>* with *List<? extends Comparable>*:

```
static void sortedInsert(List<? extends Comparable> sortedList,
???? newItem) {
   ListIterator<????> iter = stringList.listIterator();
   ...
```

We immediately run into a problem, because we have no name for the unknown type represented by the wildcard. We **need** a name for that type because the type of `newItem` and of `iter` should be the same as the type of the items in the list. The problem is solved if we write a generic method with a bounded type parameter, since then we have a name for the unknown type, and we can write a valid generic method:

```
static <T extends Comparable> void sortedInsert(List<T> sortedList,
T newItem) {
   ListIterator<T> iter = sortedList.listIterator();
   while (iter.hasNext()) {
      T item = iter.next();
      if (newItem.compareTo(item) <= 0) {
         iter.previous();
         break;
      }
   }
   iter.add(newItem);
}
```

There is still one technicality to cover in this example. *Comparable* is itself a parameterized type, but I have used it here without a type parameter. This is legal but the compiler might give you a warning about using a "raw type." In fact, the objects in the list should implement the parameterized interface *Comparable<T>*, since they are being compared to items of type *T*. This just means that instead of using *Comparable* as the type bound, we should use *Comparable<T>*:

```
static <T extends Comparable<T>> void sortedInsert(List<T>
sortedList, ...
```

With this example, I will leave the topic of generic types and generic programming. In this chapter, I have occasionally used terms such as "strange" and "weird" to talk about generic programming in Java. I will confess that I have some affection for the more simple-minded generic programming style of Smalltalk. Nevertheless, I recognize the power and increased robustness of generics in Java. I hope that I have convinced you that using the Java Collection Framework is reasonably natural and straightforward, and that using it can save you a lot of time and effort compared to repeatedly recoding the same data structures and algorithms from scratch. Things become more technical when you start writing new generic classes and methods of your own, and the syntax is (as I've said) a little strange. But with some practice, you'll get used to the syntax and will find that it's not that difficult after all.