documentation that comes in the JGL package is quite good and should be adequate to get you started.

The new collections

To me, collection classes are one of the most powerful tools for raw programming. You might have gathered that I'm somewhat disappointed in the collections provided in Java through version 1.1. As a result, it's a tremendous pleasure to see that collections were given proper attention in Java 1.2, and thoroughly redesigned (by Joshua Bloch at Sun). I consider the new collections to be one of the two major features in Java 1.2 (the other is the Swing library, covered in Chapter 13) because they significantly increase your programming muscle and help bring Java in line with more mature programming systems.

Some of the redesign makes things tighter and more sensible. For example, many names are shorter, cleaner, and easier to understand, as well as to type. Some names are changed to conform to accepted terminology: a particular favorite of mine is "iterator" instead of "enumeration."

The redesign also fills out the functionality of the collections library. You can now have the behavior of linked lists, queues, and dequeues (double-ended queues, pronounced "decks").

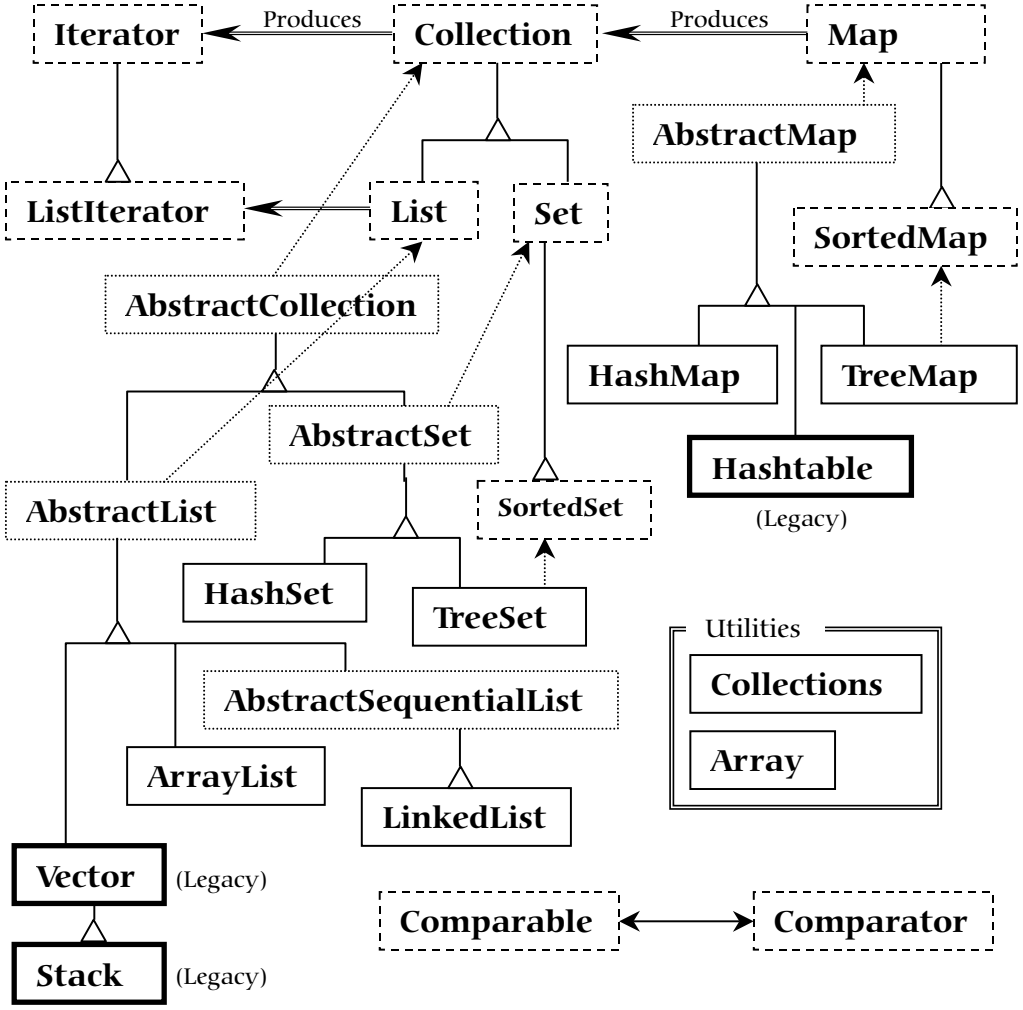
The design of a collections library is difficult (true of most library design problems). In C++, the STL covered the bases with many different classes. This was better than what was available prior to the STL (nothing), but it didn't translate well into Java. The result was a rather confusing morass of classes. On the other extreme, I've seen a collections library that consists of a single class, "collection," which acts like a **Vector** and a **Hashtable** at the same time. The designers of the new collections library wanted to strike a balance: the full functionality that you expect from a mature collections library, but easier to learn and use than the STL and other similar collections libraries. The result can seem a bit odd in places. Unlike some of the decisions made in the early Java libraries, these oddities were not accidents, but carefully considered decisions based on tradeoffs in complexity. It might take you a little while to get comfortable with some aspects of the library, but I think you'll find yourself rapidly acquiring and using these new tools.

The new collections library takes the issue of "holding your objects" and divides it into two distinct concepts:

1. **Collection**: a group of individual elements, often with some rule applied to them. A **List** must hold the elements in a particular sequence, and a **Set** cannot have any duplicate elements. (A *bag*, which is not implemented in the new collections library since **List**s provide you with that functionality, has no such rules.)

2. **Map**: a group of key-value object pairs (what you've seen up until now as a **Hashtable**). At first glance, this might seem like it ought to be a **Collection** of pairs, but when you try to implement it that way the design gets awkward, so it's clearer to make it a separate concept. On the other hand, it's convenient to look at portions of a **Map** by creating a **Collection** to represent that portion. Thus, a **Map** can return a **Set** of its keys, a **List** of its values, or a **List** of its pairs. **Map**s, like arrays, can easily be expanded to multiple dimensions without adding new concepts: you simply make a **Map** whose values are **Map**s (and the values of *those* **Map**s can be **Map**s, etc.).

Collections and **Map**s may be implemented in many different ways, according to your programming needs. It's helpful to look at a diagram of the new collections:



This diagram can be a bit overwhelming at first, but throughout the rest of this section you'll see that there are really only three collection components: **Map**, **List**, and **Set**, and only two or three implementations of each one (with, typically, a preferred version). When you see this, the new collections should not seem so daunting.

The dashed boxes represent **interface**s, the dotted boxes represent **abstract** classes, and the solid boxes are regular (concrete) classes. The dashed arrows indicate that a particular class is implementing an **interface** (or in the case of an **abstract** class, partially implementing that **interface**). The double-line arrows show that a class can produce objects of the class the arrow is pointing to. For example, any **Collection** can

produce an **Iterator**, while a **List** can produce a **ListIterator** (as well as an ordinary **Iterator**, since **List** is inherited from **Collection**).

The interfaces that are concerned with holding objects are **Collection**, **List**, **Set**, and **Map**. Typically, you'll write the bulk of your code to talk to these interfaces, and the only place where you'll specify the precise type you're using is at the point of creation. So you can create a **List** like this:

```
List x = new LinkedList();
```

Of course, you can also decide to make **x** a **LinkedList** (instead of a generic **List**) and carry the precise type information around with **x**. The beauty (and the intent) of using the **interface** is that if you decide you want to change your implementation, all you need to do is change it at the point of creation, like this:

```
List x = new ArrayList();
```

The rest of your code can remain untouched.

In the class hierarchy, you can see a number of classes whose names begin with "**Abstract**," and these can seem a bit confusing at first. They are simply tools that partially implement a particular interface. If you were making your own **Set**, for example, you wouldn't start with the **Set** interface and implement all the methods, instead you'd inherit from **AbstractSet** and do the minimal necessary work to make your new class. However, the new collections library contains enough functionality to satisfy your needs virtually all the time. So for our purposes, you can ignore any class that begins with "**Abstract**."

Therefore, when you look at the diagram, you're really concerned with only those **interface**s at the top of the diagram and the concrete classes (those with solid boxes around them). You'll typically make an object of a concrete class, upcast it to the corresponding **interface**, and then use the **interface** throughout the rest of your code. Here's a simple example, which fills a **Collection** with **String** objects and then prints each element in the **Collection**:

```
//: SimpleCollection.java
// A simple example using the new Collections
package c08.newcollections;
import java.util.*;

public class SimpleCollection {
  public static void main(String[] args) {
    Collection c = new ArrayList();
```

```
    for(int i = 0; i < 10; i++)
      c.add(Integer.toString(i));
    Iterator it = c.iterator();
    while(it.hasNext())
      System.out.println(it.next());
  }
} ///:~
```

All the code examples for the new collections libraries will be placed in the subdirectory **newcollections**, so you'll be reminded that these work only with Java 1.2. As a result, you must invoke the program by saying:

```
java c08.newcollections.SimpleCollection
```

with a similar syntax for the rest of the programs in the package.

You can see that the new collections are part of the **java.util** library, so you don't need to add any extra **import** statements to use them.

The first line in **main( )** creates an **ArrayList** object and then upcasts it to a **Collection**. Since this example uses only the **Collection** methods, any object of a class inherited from **Collection** would work, but **ArrayList** is the typical workhorse **Collection** and takes the place of **Vector**.

The **add( )** method, as its name suggests, puts a new element in the **Collection**. However, the documentation carefully states that **add( )** "ensures that this Collection contains the specified element." This is to allow for the meaning of **Set**, which adds the element only if it isn't already there. With an **ArrayList**, or any sort of **List**, **add( )** always means "put it in."

All **Collection**s can produce an **Iterator** via their **iterator( )** method. An **Iterator** is just like an **Enumeration**, which it replaces, except:

1. It uses a name (iterator) that is historically understood and accepted in the OOP community.

2. It uses shorter method names than **Enumeration**: **hasNext( )** instead of **hasMoreElements( )**, and **next( )** instead of **nextElement( )**.

3. It adds a new method, **remove( )**, which removes the last element produced by the **Iterator**. So you can call **remove( )** only once for every time you call **next( )**.

In **SimpleCollection.java**, you can see that an **Iterator** is created and used to traverse the **Collection**, printing each element.

Using Collections

The following table shows everything you can do with a **Collection**, and thus, everything you can do with a **Set** or a **List.** (**List** also has additional functionality.) **Map**s are not inherited from **Collection**, and will be treated separately.

Boolean add(Object)	*Ensures that the Collection contains the argument. Returns false if it doesn't add the argument.
Boolean addAll(Collection)	*Adds all the elements in the argument. Returns true if any elements were added.
void clear( )	*Removes all the elements in the Collection.
Boolean contains(Object)	True if the Collection contains the argument.
Boolean containsAll(Collection)	True if the Collection contains all the elements in the argument.
Boolean isEmpty( )	True if the Collection has no elements.
Iterator iterator( )	Returns an Iterator that you can use to move through the elements in the Collection.
Boolean remove(Object)	*If the argument is in the Collection, one instance of that element is removed. Returns true if a removal occurred.
Boolean removeAll(Collection)	*Removes all the elements that are contained in the argument. Returns true if any removals occurred.
Boolean retainAll(Collection)	*Retains only elements that are contained in the argument (an "intersection" from set theory). Returns true if any changes occurred.
int size( )	Returns the number of elements in the Collection.
Object[] toArray( )	Returns an array containing all the elements in the Collection.
Object[] toArray(Object[] a)	Returns an array containing all the elements in the Collection, whose type is that of the array **a** rather than plain **Object** (you must cast the array to the right type).
	*This is an "optional" method, which means it might not be implemented by a particular Collection. If not, that method throws an **UnsupportedOperationException**.

The following example demonstrates all of these methods. Again, these work with anything that inherits from **Collection**; an **ArrayList** is used as a kind of "least-common denominator":

```
//: Collection1.java
// Things you can do with all Collections
package c08.newcollections;
import java.util.*;

public class Collection1 {
  // Fill with 'size' elements, start
  // counting at 'start':
  public static Collection
  fill(Collection c, int start, int size) {
    for(int i = start; i < start + size; i++)
      c.add(Integer.toString(i));
    return c;
  }
  // Default to a "start" of 0:
  public static Collection
  fill(Collection c, int size) {
    return fill(c, 0, size);
  }
  // Default to 10 elements:
  public static Collection fill(Collection c) {
    return fill(c, 0, 10);
  }
  // Create & upcast to Collection:
  public static Collection newCollection() {
    return fill(new ArrayList());
    // ArrayList is used for simplicity, but it's
    // only seen as a generic Collection
    // everywhere else in the program.
  }
  // Fill a Collection with a range of values:
  public static Collection
  newCollection(int start, int size) {
    return fill(new ArrayList(), start, size);
  }
  // Moving through a List with an iterator:
  public static void print(Collection c) {
    for(Iterator x = c.iterator(); x.hasNext();)
      System.out.print(x.next() + " ");
    System.out.println();
  }
```

```
  public static void main(String[] args) {
    Collection c = newCollection();
    c.add("ten");
    c.add("eleven");
    print(c);
    // Make an array from the List:
    Object[] array = c.toArray();
    // Make a String array from the List:
    String[] str =
      (String[])c.toArray(new String[1]);
    // Find max and min elements; this means
    // different things depending on the way
    // the Comparable interface is implemented:
    System.out.println("Collections.max(c) = " +
      Collections.max(c));
    System.out.println("Collections.min(c) = " +
      Collections.min(c));
    // Add a Collection to another Collection
    c.addAll(newCollection());
    print(c);
    c.remove("3"); // Removes the first one
    print(c);
    c.remove("3"); // Removes the second one
    print(c);
    // Remove all components that are in the
    // argument collection:
    c.removeAll(newCollection());
    print(c);
    c.addAll(newCollection());
    print(c);
    // Is an element in this Collection?
    System.out.println(
      "c.contains(\"4\") = " + c.contains("4"));
    // Is a Collection in this Collection?
    System.out.println(
      "c.containsAll(newCollection()) = " +
      c.containsAll(newCollection()));
    Collection c2 = newCollection(5, 3);
    // Keep all the elements that are in both
    // c and c2 (an intersection of sets):
    c.retainAll(c2);
    print(c);
    // Throw away all the elements in c that
    // also appear in c2:
```

```
    c.removeAll(c2);
    System.out.println("c.isEmpty() = " +
      c.isEmpty());
    c = newCollection();
    print(c);
    c.clear(); // Remove all elements
    System.out.println("after c.clear():");
    print(c);
  }
} ///:~
```

The first methods provide a way to fill any **Collection** with test data, in this case just **int**s converted to **String**s. The second method will be used frequently throughout the rest of this chapter.

The two versions of **newCollection( )** create **ArrayList**s containing different sets of data and return them as **Collection** objects, so it's clear that nothing other than the **Collection** interface is being used.

The **print( )** method will also be used throughout the rest of this section. Since it moves through a **Collection** using an **Iterator**, which any **Collection** can produce, it will work with **List**s and **Set**s and any **Collection** that a **Map** produces.

main( ) uses simple exercises to show all of the methods in **Collection**.

The following sections compare the various implementations of **List**, **Set,** and **Map** and indicate in each case (with an asterisk) which one should be your default choice. You'll notice that the legacy classes **Vector**, **Stack**, and **Hashtable** are *not* included because in all cases there are preferred classes within the new collections.

Using Lists

List (interface)	Order is the most important feature of a **List**; it promises to maintain elements in a particular sequence. **List** adds a number of methods to **Collection** that allow insertion and removal of elements in the middle of a **List.** (This is recommended only for a **LinkedList.**) A **List** will produce a **ListIterator**, and using this you can traverse the **List** in both directions, as well as insert and remove elements in the middle of the list (again, recommended only for a **LinkedList**).

ArrayList*	A **List** backed by an array. Use instead of **Vector** as a general-purpose object holder. Allows rapid random access to elements, but is slow when inserting and removing elements from the middle of a list. **ListIterator** should be used only for back-and-forth traversal of an **ArrayList**, but not for inserting and removing elements, which is expensive compared to **LinkedList**.
LinkedList	Provides optimal sequential access, with inexpensive insertions and deletions from the middle of the list. Relatively slow for random access. (Use **ArrayList** instead.) Also has **addFirst( )**, **addLast( )**, **getFirst( )**, **getLast( )**, **removeFirst( )**, and **removeLast( )** (which are not defined in any interfaces or base classes) to allow it to be used as a stack, a queue, and a dequeue.

The methods in the following example each cover a different group of activities: things that every list can do (**basicTest( )**), moving around with an **Iterator** (**iterMotion( )**) versus changing things with an **Iterator** (**iterManipulation( )**), seeing the effects of **List** manipulation (**testVisual( )**), and operations available only to **LinkedList**s.

```
//: List1.java
// Things you can do with Lists
package c08.newcollections;
import java.util.*;

public class List1 {
  // Wrap Collection1.fill() for convenience:
  public static List fill(List a) {
    return (List)Collection1.fill(a);
  }
  // You can use an Iterator, just as with a
  // Collection, but you can also use random
  // access with get():
  public static void print(List a) {
    for(int i = 0; i < a.size(); i++)
      System.out.print(a.get(i) + " ");
    System.out.println();
  }
  static boolean b;
  static Object o;
  static int i;
```

```
  static Iterator it;
  static ListIterator lit;
  public static void basicTest(List a) {
    a.add(1, "x"); // Add at location 1
    a.add("x"); // Add at end
    // Add a collection:
    a.addAll(fill(new ArrayList()));
    // Add a collection starting at location 3:
    a.addAll(3, fill(new ArrayList()));
    b = a.contains("1"); // Is it in there?
    // Is the entire collection in there?
    b = a.containsAll(fill(new ArrayList()));
    // Lists allow random access, which is cheap
    // for ArrayList, expensive for LinkedList:
    o = a.get(1); // Get object at location 1
    i = a.indexOf("1"); // Tell index of object
    // indexOf, starting search at location 2:
    i = a.indexOf("1", 2);
    b = a.isEmpty(); // Any elements inside?
    it = a.iterator(); // Ordinary Iterator
    lit = a.listIterator(); // ListIterator
    lit = a.listIterator(3); // Start at loc 3
    i = a.lastIndexOf("1"); // Last match
    i = a.lastIndexOf("1", 2); // ...after loc 2
    a.remove(1); // Remove location 1
    a.remove("3"); // Remove this object
    a.set(1, "y"); // Set location 1 to "y"
    // Keep everything that's in the argument
    // (the intersection of the two sets):
    a.retainAll(fill(new ArrayList()));
    // Remove elements in this range:
    a.removeRange(0, 2);
    // Remove everything that's in the argument:
    a.removeAll(fill(new ArrayList()));
    i = a.size(); // How big is it?
    a.clear(); // Remove all elements
  }
  public static void iterMotion(List a) {
    ListIterator it = a.listIterator();
    b = it.hasNext();
    b = it.hasPrevious();
    o = it.next();
    i = it.nextIndex();
    o = it.previous();
```

```
    i = it.previousIndex();
  }
  public static void iterManipulation(List a) {
    ListIterator it = a.listIterator();
    it.add("47");
    // Must move to an element after add():
    it.next();
    // Remove the element that was just produced:
    it.remove();
    // Must move to an element after remove():
    it.next();
    // Change the element that was just produced:
    it.set("47");
  }
  public static void testVisual(List a) {
    print(a);
    List b = new ArrayList();
    fill(b);
    System.out.print("b = ");
    print(b);
    a.addAll(b);
    a.addAll(fill(new ArrayList()));
    print(a);
    // Shrink the list by removing all the
    // elements beyond the first 1/2 of the list
    System.out.println(a.size());
    System.out.println(a.size()/2);
    a.removeRange(a.size()/2, a.size()/2 + 2);
    print(a);
    // Insert, remove, and replace elements
    // using a ListIterator:
    ListIterator x = a.listIterator(a.size()/2);
    x.add("one");
    print(a);
    System.out.println(x.next());
    x.remove();
    System.out.println(x.next());
    x.set("47");
    print(a);
    // Traverse the list backwards:
    x = a.listIterator(a.size());
    while(x.hasPrevious())
      System.out.print(x.previous() + " ");
    System.out.println();
```

```
    System.out.println("testVisual finished");
  }
  // There are some things that only
  // LinkedLists can do:
  public static void testLinkedList() {
    LinkedList ll = new LinkedList();
    Collection1.fill(ll, 5);
    print(ll);
    // Treat it like a stack, pushing:
    ll.addFirst("one");
    ll.addFirst("two");
    print(ll);
    // Like "peeking" at the top of a stack:
    System.out.println(ll.getFirst());
    // Like popping a stack:
    System.out.println(ll.removeFirst());
    System.out.println(ll.removeFirst());
    // Treat it like a queue, pulling elements
    // off the tail end:
    System.out.println(ll.removeLast());
    // With the above operations, it's a dequeue!
    print(ll);
  }
  public static void main(String args[]) {
    // Make and fill a new list each time:
    basicTest(fill(new LinkedList()));
    basicTest(fill(new ArrayList()));
    iterMotion(fill(new LinkedList()));
    iterMotion(fill(new ArrayList()));
    iterManipulation(fill(new LinkedList()));
    iterManipulation(fill(new ArrayList()));
    testVisual(fill(new LinkedList()));
    testLinkedList();
  }
} ///:~
```

In **basicTest( )** and **iterMotion( )** the calls are simply made to show the proper syntax, and while the return value is captured, it is not used. In some cases, the return value isn't captured since it isn't typically used. You should look up the full usage of each of these methods in your online documentation before you use them.

Using Sets

Set has exactly the same interface as **Collection**, so there isn't any extra functionality as there is with the two different **List**s. Instead, the **Set** is exactly a **Collection**, it just has different behavior. (This is the ideal use of inheritance and polymorphism: to express different behavior.) A **Set** allows only one instance of each object value to exist (what constitutes the "value" of an object is more complex, as you shall see).

Set (interface)	Each element that you add to the **Set** must be unique; otherwise the **Set** doesn't add the duplicate element. Objects added to a **Set** must define **equals( )** to establish object uniqueness. **Set** has exactly the same interface as **Collection**. The **Set** interface does not guarantee it will maintain its elements in any particular order.
HashSet*	For **Set**s where fast lookup time is important. Objects must also define **hashCode( )**.
TreeSet	An ordered **Set** backed by a red-black tree. This way, you can extract an ordered sequence from a **Set**.

The following example does *not* show everything you can do with a **Set**, since the interface is the same as **Collection** and so was exercised in the previous example. Instead, this demonstrates the behavior that makes a **Set** unique:

```
//: Set1.java
// Things you can do with Sets
package c08.newcollections;
import java.util.*;

public class Set1 {
  public static void testVisual(Set a) {
    Collection1.fill(a);
    Collection1.fill(a);
    Collection1.fill(a);
    Collection1.print(a); // No duplicates!
    // Add another set to this one:
    a.addAll(a);
    a.add("one");
    a.add("one");
    a.add("one");
    Collection1.print(a);
    // Look something up:
    System.out.println("a.contains(\"one\"): " +
```

```
        a.contains("one"));
    }
    public static void main(String[] args) {
      testVisual(new HashSet());
      testVisual(new TreeSet());
    }
} ///:~
```

Duplicate values are added to the **Set**, but when it is printed you'll see the **Set** has accepted only one instance of each value.

When you run this program you'll notice that the order maintained by the **HashSet** is different from **TreeSet**, since each has a different way of storing elements so they can be located later. (**TreeSet** keeps them sorted, while **HashSet** uses a hashing function, which is designed specifically for rapid lookups.) When creating your own types, be aware that a **Set** needs a way to maintain a storage order, just as with the "groundhog" examples shown earlier in this chapter. To implement comparability with the new collections, however, you must implement the **Comparable** interface and define the **compareTo( )** method (this will be described more fully later). Here's an example:

```
//: Set2.java
// Putting your own type in a Set
package c08.newcollections;
import java.util.*;

class MyType implements Comparable {
  private int i;
  public MyType(int n) { i = n; }
  public boolean equals(Object o) {
    return
      (o instanceof MyType)
      && (i == ((MyType)o).i);
  }
  public int hashCode() { return i; }
  public String toString() { return i + " "; }
  public int compareTo(Object o) {
    int i2 = ((MyType) o).i;
    return (i2 < i ? -1 : (i2 == i ? 0 : 1));
  }
}

public class Set2 {
  public static Set fill(Set a, int size) {
```

```
    for(int i = 0; i < size; i++)
      a.add(new MyType(i));
    return a;
  }
  public static Set fill(Set a) {
    return fill(a, 10);
  }
  public static void test(Set a) {
    fill(a);
    fill(a); // Try to add duplicates
    fill(a);
    a.addAll(fill(new TreeSet()));
    System.out.println(a);
  }
  public static void main(String[] args) {
    test(new HashSet());
    test(new TreeSet());
  }
} ///:~
```

The definitions for **equals( )** and **hashCode( )** follow the form given in the "groundhog" examples. You must define an **equals( )** in both cases, but the **hashCode( )** is absolutely necessary only if the class will be placed in a **HashSet** (which is likely, since that should generally be your first choice as a **Set** implementation). However, as a programming style you should always override **hashCode( )** when you override **equals( )**.

In the **compareTo( )**, note that I did *not* use the "simple and obvious" form **return i-i2**. Although this is a common programming error, it would only work properly if **i** and **i2** were **unsigned int**s (if Java *had* an "unsigned" keyword, which it does not). It breaks for Java's signed **int**, which are not big enough to represent the difference of two signed **int**s. If **i** is a large positive integer and **j** is a large negative integer, **i-j** will overflow and return a negative value, which will not work.

Using Maps

Map (interface)	Maintains key-value associations (pairs), so you can look up a value using a key.
HashMap*	Implementation based on a hash table. (Use this instead of **Hashtable.**) Provides constant-time performance for inserting and locating pairs. Performance can be adjusted via constructors that allow you to set the *capacity* and *load factor* of the hash table.

TreeMap	Implementation based on a red-black tree. When you view the keys or the pairs, they will be in sorted order (determined by **Comparable** or **Comparator**, discussed later). The point of a **TreeMap** is that you get the results in sorted order. **TreeMap** is the only **Map** with the **subMap( )** method, which allows you to return a portion of the tree.

The following example contains two sets of test data and a **fill( )** method that allows you to fill any map with any two-dimensional array of **Object**s. These tools will be used in other **Map** examples, as well.

```
//: Map1.java
// Things you can do with Maps
package c08.newcollections;
import java.util.*;

public class Map1 {
  public final static String[][] testData1 = {
    { "Happy", "Cheerful disposition" },
    { "Sleepy", "Prefers dark, quiet places" },
    { "Grumpy", "Needs to work on attitude" },
    { "Doc", "Fantasizes about advanced degree"},
    { "Dopey", "'A' for effort" },
    { "Sneezy", "Struggles with allergies" },
    { "Bashful", "Needs self-esteem workshop"},
  };
  public final static String[][] testData2 = {
    { "Belligerent", "Disruptive influence" },
    { "Lazy", "Motivational problems" },
    { "Comatose", "Excellent behavior" }
  };
  public static Map fill(Map m, Object[][] o) {
    for(int i = 0; i < o.length; i++)
      m.put(o[i][0], o[i][1]);
    return m;
  }
  // Producing a Set of the keys:
  public static void printKeys(Map m) {
    System.out.print("Size = " + m.size() +", ");
    System.out.print("Keys: ");
    Collection1.print(m.keySet());
  }
  // Producing a Collection of the values:
  public static void printValues(Map m) {
```

```
    System.out.print("Values: ");
    Collection1.print(m.values());
  }
  // Iterating through Map.Entry objects (pairs):
  public static void print(Map m) {
    Collection entries = m.entries();
    Iterator it = entries.iterator();
    while(it.hasNext()) {
      Map.Entry e = (Map.Entry)it.next();
      System.out.println("Key = " + e.getKey() +
        ", Value = " + e.getValue());
    }
  }
  public static void test(Map m) {
    fill(m, testData1);
    // Map has 'Set' behavior for keys:
    fill(m, testData1);
    printKeys(m);
    printValues(m);
    print(m);
    String key = testData1[4][0];
    String value = testData1[4][1];
    System.out.println("m.containsKey(\"" + key +
      "\"): " + m.containsKey(key));
    System.out.println("m.get(\"" + key + "\"): "
      + m.get(key));
    System.out.println("m.containsValue(\""
      + value + "\"): " +
      m.containsValue(value));
    Map m2 = fill(new TreeMap(), testData2);
    m.putAll(m2);
    printKeys(m);
    m.remove(testData2[0][0]);
    printKeys(m);
    m.clear();
    System.out.println("m.isEmpty(): "
      + m.isEmpty());
    fill(m, testData1);
    // Operations on the Set change the Map:
    m.keySet().removeAll(m.keySet());
    System.out.println("m.isEmpty(): "
      + m.isEmpty());
  }
  public static void main(String args[]) {
```

```
    System.out.println("Testing HashMap");
    test(new HashMap());
    System.out.println("Testing TreeMap");
    test(new TreeMap());
  }
} ///:~
```

The **printKeys( )**, **printValues( )**, and **print( )** methods are not only useful utilities, they also demonstrate the production of **Collection** views of a **Map**. The **keySet( )** method produces a **Set** backed by the keys in the **Map**; here, it is treated as only a **Collection**. Similar treatment is given to **values( )**, which produces a **List** containing all the values in the **Map.** (Note that keys must be unique, while values can contain duplicates.) Since these **Collection**s are backed by the **Map**, any changes in a **Collection** will be reflected in the associated **Map**.

The **print( )** method grabs the **Iterator** produced by **entries** and uses it to print both the key and value for each pair. The rest of the program provides simple examples of each **Map** operation, and tests each type of **Map**.

When creating your own class to use as a key in a **Map**, you must deal with the same issues discussed previously for **Set**s.

Choosing an implementation

From the diagram on page 363 you can see that there are really only three collection components: **Map**, **List**, and **Set**, and only two or three implementations of each interface. If you need to use the functionality offered by a particular **interface**, how do you decide which particular implementation to use?

To understand the answer, you must be aware that each different implementation has its own features, strengths, and weaknesses. For example, you can see in the diagram that the "feature" of **Hashtable**, **Vector**, and **Stack** is that they are legacy classes, so that existing code doesn't break. On the other hand, it's best if you don't use those for new (Java 1.2) code.

The distinction between the other collections often comes down to what they are "backed by;" that is, the data structures that physically implement your desired **interface**. This means that, for example, **ArrayList, LinkedList**, and **Vector** (which is roughly equivalent to **ArrayList**) all implement the **List** interface so your program will produce the same results regardless of the one you use. However, **ArrayList** (and

Vector) is backed by an array, while the **LinkedList** is implemented in the usual way for a doubly-linked list, as individual objects each containing data along with handles to the previous and next elements in the list. Because of this, if you want to do many insertions and removals in the middle of a list a **LinkedList** is the appropriate choice. (**LinkedList** also has additional functionality that is established in **AbstractSequentialList**.) If not, an **ArrayList** is probably faster.

As another example, a **Set** can be implemented as either an **TreeSet** or a **HashSet**. A **TreeSet** is backed by a **TreeMap** and is designed to produce a constantly-sorted set. However, if you're going to have larger quantities in your **Set**, the performance of **TreeSet** insertions will get slow. When you're writing a program that needs a **Set**, you should choose **HashSet** by default, and change to **TreeSet** when it's more important to have a constantly-sorted set.

Choosing between Lists

The most convincing way to see the differences between the implementations of **List** is with a performance test. The following code establishes an inner base class to use as a test framework, then creates an anonymous inner class for each different test. Each of these inner classes is called by the **test( )** method. This approach allows you to easily add and remove new kinds of tests.

```
//: ListPerformance.java
// Demonstrates performance differences in Lists
package c08.newcollections;
import java.util.*;

public class ListPerformance {
  private static final int REPS = 100;
  private abstract static class Tester {
    String name;
    int size; // Test quantity
    Tester(String name, int size) {
      this.name = name;
      this.size = size;
    }
    abstract void test(List a);
  }
  private static Tester[] tests = {
    new Tester("get", 300) {
      void test(List a) {
        for(int i = 0; i < REPS; i++) {
```

```
          for(int j = 0; j < a.size(); j++)
            a.get(j);
        }
      }
    },
    new Tester("iteration", 300) {
      void test(List a) {
        for(int i = 0; i < REPS; i++) {
          Iterator it = a.iterator();
          while(it.hasNext())
            it.next();
        }
      }
    },
    new Tester("insert", 1000) {
      void test(List a) {
        int half = a.size()/2;
        String s = "test";
        ListIterator it = a.listIterator(half);
        for(int i = 0; i < size * 10; i++)
          it.add(s);
      }
    },
    new Tester("remove", 5000) {
      void test(List a) {
        ListIterator it = a.listIterator(3);
        while(it.hasNext()) {
          it.next();
          it.remove();
        }
      }
    },
  };
  public static void test(List a) {
    // A trick to print out the class name:
    System.out.println("Testing " +
      a.getClass().getName());
    for(int i = 0; i < tests.length; i++) {
      Collection1.fill(a, tests[i].size);
      System.out.print(tests[i].name);
      long t1 = System.currentTimeMillis();
      tests[i].test(a);
      long t2 = System.currentTimeMillis();
      System.out.println(": " + (t2 - t1));
```

```
    }
  }
  public static void main(String[] args) {
    test(new ArrayList());
    test(new LinkedList());
  }
} ///:~
```

The inner class **Tester** is **abstract**, to provide a base class for the specific tests. It contains a **String** to be printed when the test starts, a **size** parameter to be used by the test for quantity of elements or repetitions of tests, a constructor to initialize the fields, and an **abstract** method **test( )** that does the work. All the different types of tests are collected in one place, the array **tests**, which is initialized with different anonymous inner classes that inherit from **Tester**. To add or remove tests, simply add or remove an inner class definition from the array, and everything else happens automatically.

The **List** that's handed to **test( )** is first filled with elements, then each test in the **tests** array is timed. The results will vary from machine to machine; they are intended to give only an order of magnitude comparison between the performance of the different collections. Here is a summary of one run:

Type	Get	Iteration	Insert	Remove
ArrayList	110	490	3790	8730
LinkedList	1980	220	110	110

You can see that random accesses (**get( )**) are cheap for **ArrayList**s and expensive for **LinkedList**s. (Oddly, iteration is *faster* for a **LinkedList** than an **ArrayList**, which is counter-intuitive.) On the other hand, insertions and removals from the middle of a list are dramatically cheaper for a **LinkedList** than for an **ArrayList**. The best approach is probably to choose an **ArrayList** as your default and to change to a **LinkedList** if you discover performance problems because of many insertions and removals from the middle of the list.

Choosing between Sets

You can choose between an **TreeSet** and a **HashSet**, depending on the size of the **Set** (if you need to produce an ordered sequence from a **Set**, use **TreeSet**). The following test program gives an indication of this tradeoff:

```
//: SetPerformance.java
package c08.newcollections;
```

```
import java.util.*;

public class SetPerformance {
  private static final int REPS = 200;
  private abstract static class Tester {
    String name;
    Tester(String name) { this.name = name; }
    abstract void test(Set s, int size);
  }
  private static Tester[] tests = {
    new Tester("add") {
      void test(Set s, int size) {
        for(int i = 0; i < REPS; i++) {
          s.clear();
          Collection1.fill(s, size);
        }
      }
    },
    new Tester("contains") {
      void test(Set s, int size) {
        for(int i = 0; i < REPS; i++)
          for(int j = 0; j < size; j++)
            s.contains(Integer.toString(j));
      }
    },
    new Tester("iteration") {
      void test(Set s, int size) {
        for(int i = 0; i < REPS * 10; i++) {
          Iterator it = s.iterator();
          while(it.hasNext())
            it.next();
        }
      }
    },
  };
  public static void test(Set s, int size) {
    // A trick to print out the class name:
    System.out.println("Testing " +
      s.getClass().getName() + " size " + size);
    Collection1.fill(s, size);
    for(int i = 0; i < tests.length; i++) {
      System.out.print(tests[i].name);
      long t1 = System.currentTimeMillis();
      tests[i].test(s, size);
```

```
      long t2 = System.currentTimeMillis();
      System.out.println(": " +
        ((double)(t2 - t1)/(double)size));
    }
  }
  public static void main(String[] args) {
    // Small:
    test(new TreeSet(), 10);
    test(new HashSet(), 10);
    // Medium:
    test(new TreeSet(), 100);
    test(new HashSet(), 100);
    // Large:
    test(new HashSet(), 1000);
    test(new TreeSet(), 1000);
  }
} ///:~
```

The following table shows the results of one run (using Beta3 software on one particular platform; you should run the test yourself as well):

Type	Test size	Add	Contains	Iteration
	10	22.0	11.0	16.0
TreeSet	100	22.5	13.2	12.1
	1000	31.1	18.7	11.8
	10	5.0	6.0	27.0
HashSet	100	6.6	6.6	10.9
	1000	7.4	6.6	9.5

HashSet is generally superior to **TreeSet** for all operations, and the performance is effectively independent of size.

Choosing between Maps

When choosing between implementations of **Map**, the size of the **Map** is what most strongly affects performance, and the following test program gives an indication of this tradeoff:

```
//: MapPerformance.java
// Demonstrates performance differences in Maps
package c08.newcollections;
import java.util.*;

public class MapPerformance {
  private static final int REPS = 200;
```

```
  public static Map fill(Map m, int size) {
    for(int i = 0; i < size; i++) {
      String x = Integer.toString(i);
      m.put(x, x);
    }
    return m;
  }
  private abstract static class Tester {
    String name;
    Tester(String name) { this.name = name; }
    abstract void test(Map m, int size);
  }
  private static Tester[] tests = {
    new Tester("put") {
      void test(Map m, int size) {
        for(int i = 0; i < REPS; i++) {
          m.clear();
          fill(m, size);
        }
      }
    },
    new Tester("get") {
      void test(Map m, int size) {
        for(int i = 0; i < REPS; i++)
          for(int j = 0; j < size; j++)
            m.get(Integer.toString(j));
      }
    },
    new Tester("iteration") {
      void test(Map m, int size) {
        for(int i = 0; i < REPS * 10; i++) {
          Iterator it = m.entries().iterator();
          while(it.hasNext())
            it.next();
        }
      }
    },
  };
  public static void test(Map m, int size) {
    // A trick to print out the class name:
    System.out.println("Testing " +
      m.getClass().getName() + " size " + size);
    fill(m, size);
    for(int i = 0; i < tests.length; i++) {
```

```
        System.out.print(tests[i].name);
        long t1 = System.currentTimeMillis();
        tests[i].test(m, size);
        long t2 = System.currentTimeMillis();
        System.out.println(": " +
          ((double)(t2 - t1)/(double)size));
      }
    }
    public static void main(String[] args) {
      // Small:
      test(new Hashtable(), 10);
      test(new HashMap(), 10);
      test(new TreeMap(), 10);
      // Medium:
      test(new Hashtable(), 100);
      test(new HashMap(), 100);
      test(new TreeMap(), 100);
      // Large:
      test(new HashMap(), 1000);
      test(new Hashtable(), 1000);
      test(new TreeMap(), 1000);
    }
  } ///:~
```

Because the size of the map is the issue, you'll see that the timing tests divide the time by the size to normalize each measurement. Here is one set of results. (Yours will probably be different.)

Type	Test size	Put	Get	Iteration
Hashtable	10	11.0	5.0	44.0
	100	7.7	7.7	16.5
	1000	8.0	8.0	14.4
TreeMap	10	16.0	11.0	22.0
	100	25.8	15.4	13.2
	1000	33.8	20.9	13.6
HashMap	10	11.0	6.0	33.0
	100	8.2	7.7	13.7
	1000	8.0	7.8	11.9

As you might expect, **Hashtable** performance is roughly equivalent to **HashMap** (you can also see that **HashMap** is generally a bit faster. Remember that **HashMap** is intended to replace **Hashtable**). The **TreeMap** is generally slower than the **HashMap**, so why would you use it? So you could use it not as a **Map**, but as a way to create an ordered

list. The behavior of a tree is such that it's always in order and doesn't have to be specially sorted. (The *way* it is ordered will be discussed later.) Once you fill a **TreeMap**, you can call **keySet( )** to get a **Set** view of the keys, then **toArray( )** to produce an array of those keys. You can then use the **static** method **Arrays.binarySearch( )** (discussed later) to rapidly find objects in your sorted array. Of course, you would probably only do this if, for some reason, the behavior of a **HashMap** was unacceptable, since **HashMap** is designed to rapidly find things. In the end, when you're using a **Map** your first choice should be **HashMap**, and only if you need a constantly-sorted **Map** will you need **TreeMap**.

There is another performance issue that the above table does not address, and that is speed of creation. The following program tests creation speed for different types of **Map**:

```
//: MapCreation.java
// Demonstrates time differences in Map creation
package c08.newcollections;
import java.util.*;

public class MapCreation {
  public static void main(String[] args) {
    final long REPS = 100000;
    long t1 = System.currentTimeMillis();
    System.out.print("Hashtable");
    for(long i = 0; i < REPS; i++)
      new Hashtable();
    long t2 = System.currentTimeMillis();
    System.out.println(": " + (t2 - t1));
    t1 = System.currentTimeMillis();
    System.out.print("TreeMap");
    for(long i = 0; i < REPS; i++)
      new TreeMap();
    t2 = System.currentTimeMillis();
    System.out.println(": " + (t2 - t1));
    t1 = System.currentTimeMillis();
    System.out.print("HashMap");
    for(long i = 0; i < REPS; i++)
      new HashMap();
    t2 = System.currentTimeMillis();
    System.out.println(": " + (t2 - t1));
  }
} ///:~
```

At the time this program was written, the creation speed of **TreeMap** was dramatically faster than the other two types. This, along with the acceptable and consistent **put( )** performance of **TreeMap**, suggests a possible strategy if you're creating many **Map**s, and only later in your program doing many lookups: Create and fill **TreeMap**s, and when you start looking things up, convert the important **TreeMap**s into **HashMap**s using the **HashMap(Map)** constructor. Again, you should only worry about this sort of thing after it's been proven that you have a performance bottleneck. ("First make it work, then make it fast – if you must.")

Unsupported operations

It's possible to turn an array into a **List** with the **static Arrays.toList( )** method:

```
//: Unsupported.java
// Sometimes methods defined in the Collection
// interfaces don't work!
package c08.newcollections;
import java.util.*;

public class Unsupported {
  private static String[] s = {
    "one", "two", "three", "four", "five",
    "six", "seven", "eight", "nine", "ten",
  };
  static List a = Arrays.toList(s);
  static List a2 = Arrays.toList(
    new String[] { s[3], s[4], s[5] });
  public static void main(String[] args) {
    Collection1.print(a); // Iteration
    System.out.println(
      "a.contains(" + s[0] + ") = " +
      a.contains(s[0]));
    System.out.println(
      "a.containsAll(a2) = " +
      a.containsAll(a2));
    System.out.println("a.isEmpty() = " +
      a.isEmpty());
    System.out.println(
      "a.indexOf(" + s[5] + ") = " +
      a.indexOf(s[5]));
    // Traverse backwards:
```

```
    ListIterator lit = a.listIterator(a.size());
    while(lit.hasPrevious())
      System.out.print(lit.previous());
    System.out.println();
    // Set the elements to different values:
    for(int i = 0; i < a.size(); i++)
      a.set(i, "47");
    Collection1.print(a);
    // Compiles, but won't run:
    lit.add("X"); // Unsupported operation
    a.clear(); // Unsupported
    a.add("eleven"); // Unsupported
    a.addAll(a2); // Unsupported
    a.retainAll(a2); // Unsupported
    a.remove(s[0]); // Unsupported
    a.removeAll(a2); // Unsupported
  }
} ///:~
```

You'll discover that only a portion of the **Collection** and **List** interfaces are actually implemented. The rest of the methods cause the unwelcome appearance of something called an **UnsupportedOperationException**. You'll learn all about exceptions in the next chapter, but the short story is that the **Collection interface**, as well as some of the other **interface**s in the new collections library, contain "optional" methods, which might or might not be "supported" in the concrete class that **implements** that **interface**. Calling an unsupported method causes an **UnsupportedOperationException** to indicate a programming error.

"What?!?" you say, incredulous. "The whole point of **interface**s and base classes is that they promise these methods will do something meaningful! This breaks that promise – it says that not only will calling some methods *not* perform a meaningful behavior, they will stop the program! Type safety was just thrown out the window!" It's not quite that bad. With a **Collection**, **List**, **Set**, or **Map**, the compiler still restricts you to calling only the methods in that **interface**, so it's not like Smalltalk (in which you can call any method for any object, and find out only when you run the program whether your call does anything). In addition, most methods that take a **Collection** as an argument only read from that **Collection** –all the "read" methods of **Collection** are *not* optional.

This approach prevents an explosion of interfaces in the design. Other designs for collection libraries always seem to end up with a confusing plethora of interfaces to describe each of the variations on the main theme and are thus difficult to learn. It's not even possible to capture all

of the special cases in **interface**s, because someone can always invent a new **interface**. The "unsupported operation" approach achieves an important goal of the new collections library: it is simple to learn and use. For this approach to work, however:

1. The **UnsupportedOperationException** must be a rare event. That is, for most classes all operations should work, and only in special cases should an operation be unsupported. This is true in the new collections library, since the classes you'll use 99 percent of the time – **ArrayList**, **LinkedList**, **HashSet**, and **HashMap**, as well as the other concrete implementations – support all of the operations. The design does provide a "back door" if you want to create a new **Collection** without providing meaningful definitions for all the methods in the **Collection interface**, and yet still fit it into the existing library.

2. When an operation *is* unsupported, there should be reasonable likelihood that an **UnsupportedOperationException** will appear at implementation time, rather than after you've shipped the product to the customer. After all, it indicates a programming error: you've used a class incorrectly. This point is less certain, and is where the experimental nature of this design comes into play. Only over time will we find out how well it works.

In the example above, **Arrays.toList( )** produces a **List** that is backed by a fixed-size array. Therefore it makes sense that the only supported operations are the ones that don't change the size of the array. If, on the other hand, a new **interface** were required to express this different kind of behavior (called, perhaps, "**FixedSizeList**"), it would throw open the door to complexity and soon you wouldn't know where to start when trying to use the library.

The documentation for a method that takes a **Collection**, **List**, **Set**, or **Map** as an argument should specify which of the optional methods must be implemented. For example, sorting requires the **set**( ) and **Iterator.set**( ) methods but not **add**( ) and **remove**( ).

Sorting and searching

Java 1.2 adds utilities to perform sorting and searching for arrays or **List**s. These utilities are **static** methods of two new classes: **Arrays** for sorting and searching arrays, and **Collections** for sorting and searching **List**s.

Arrays

The **Arrays** class has an overloaded **sort( )** and **binarySearch( )** for arrays of all the primitive types, as well as for **String** and **Object**. Here's an example that shows sorting and searching an array of **byte** (all the other primitives look the same) and an array of **String**:

```
//: Array1.java
// Testing the sorting & searching in Arrays
package c08.newcollections;
import java.util.*;

public class Array1 {
  static Random r = new Random();
  static String ssource =
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ" +
    "abcdefghijklmnopqrstuvwxyz";
  static char[] src = ssource.toCharArray();
  // Create a random String
  public static String randString(int length) {
    char[] buf = new char[length];
    int rnd;
    for(int i = 0; i < length; i++) {
      rnd = Math.abs(r.nextInt()) % src.length;
      buf[i] = src[rnd];
    }
    return new String(buf);
  }
  // Create a random array of Strings:
  public static
  String[] randStrings(int length, int size) {
    String[] s = new String[size];
    for(int i = 0; i < size; i++)
      s[i] = randString(length);
    return s;
  }
  public static void print(byte[] b) {
    for(int i = 0; i < b.length; i++)
      System.out.print(b[i] + " ");
    System.out.println();
  }
  public static void print(String[] s) {
    for(int i = 0; i < s.length; i++)
      System.out.print(s[i] + " ");
```

```
    System.out.println();
  }
  public static void main(String[] args) {
    byte[] b = new byte[15];
    r.nextBytes(b); // Fill with random bytes
    print(b);
    Arrays.sort(b);
    print(b);
    int loc = Arrays.binarySearch(b, b[10]);
    System.out.println("Location of " + b[10] +
      " = " + loc);
    // Test String sort & search:
    String[] s = randStrings(4, 10);
    print(s);
    Arrays.sort(s);
    print(s);
    loc = Arrays.binarySearch(s, s[4]);
    System.out.println("Location of " + s[4] +
      " = " + loc);
  }
} ///:~
```

The first part of the class contains utilities to generate random **String** objects using an array of characters from which random letters can be selected. **randString( )** returns a string of any length, and **randStrings( )** creates an array of random **String**s, given the length of each **String** and the desired size of the array. The two **print( )** methods simplify the display of the sample arrays. In **main( )**, **Random.nextBytes( )** fills the array argument with randomly-selected **byte**s. (There are no corresponding **Random** methods to create arrays of the other primitive data types.) Once you have an array, you can see that it's only a single method call to perform a **sort( )** or **binarySearch( )**. There's an important warning concerning **binarySearch( )**: If you do not call **sort( )** before you perform a **binarySearch( )**, unpredictable behavior can occur, including infinite loops.

Sorting and searching with **String**s looks the same, but when you run the program you'll notice something interesting: the sorting is lexicographic, so uppercase letters precede lowercase letters in the character set. Thus, all the capital letters are at the beginning of the list, followed by the lowercase letters, so 'Z' precedes 'a'. It turns out that even telephone books are typically sorted this way.

Comparable and Comparator

What if this isn't what you want? For example, the index in this book would not be too useful if you had to look in two places for everything that begins with 'A' or 'a'.

When you want to sort an array of **Object**, there's a problem. What determines the ordering of two **Object**s? Unfortunately, the original Java designers didn't consider this an important problem, or it would have been defined in the root class **Object**. As a result, ordering must be imposed on **Object**s from the outside, and the new collections library provides a standard way to do this (which is almost as good as defining it in **Object**).

There is a **sort( )** for arrays of **Object** (and **String**, of course, is an **Object**) that takes a second argument: an object that implements the **Comparator** interface (part of the new collections library) and performs comparisons with its single **compare( )** method. This method takes the two objects to be compared as its arguments and returns a negative integer if the first argument is less than the second, zero if they're equal, and a positive integer if the first argument is greater than the second. With this knowledge, the **String** portion of the example above can be re-implemented to perform an alphabetic sort:

```
//: AlphaComp.java
// Using Comparator to perform an alphabetic sort
package c08.newcollections;
import java.util.*;

public class AlphaComp implements Comparator {
  public int compare(Object o1, Object o2) {
    // Assume it's used only for Strings...
    String s1 = ((String)o1).toLowerCase();
    String s2 = ((String)o2).toLowerCase();
    return s1.compareTo(s2);
  }
  public static void main(String[] args) {
    String[] s = Array1.randStrings(4, 10);
    Array1.print(s);
    AlphaComp ac = new AlphaComp();
    Arrays.sort(s, ac);
    Array1.print(s);
    // Must use the Comparator to search, also:
    int loc = Arrays.binarySearch(s, s[3], ac);
    System.out.println("Location of " + s[3] +
```

```
      " = " + loc);
  }
} ///:~
```

By casting to **String**, the **compare( )** method implicitly tests to ensure that it is used only with **String** objects – the run-time system will catch any discrepancies. After forcing both **String**s to lower case, the **String.compareTo( )** method produces the desired results.

When you use your own **Comparator** to perform a **sort( )**, you must use that same **Comparator** when using **binarySearch( )**.

The **Arrays** class has another **sort( )** method that takes a single argument: an array of **Object**, but with no **Comparator**. This **sort( )** method must also have some way to compare two **Object**s. It uses the *natural comparison method* that is imparted to a class by implementing the **Comparable interface**. This **interface** has a single method, **compareTo( )**, which compares the object to its argument and returns negative, zero, or positive depending on whether it is less than, equal to, or greater than the argument. A simple example demonstrates this:

```
//: CompClass.java
// A class that implements Comparable
package c08.newcollections;
import java.util.*;

public class CompClass implements Comparable {
  private int i;
  public CompClass(int ii) { i = ii; }
  public int compareTo(Object o) {
    // Implicitly tests for correct type:
    int argi = ((CompClass)o).i;
    if(i == argi) return 0;
    if(i < argi) return -1;
    return 1;
  }
  public static void print(Object[] a) {
    for(int i = 0; i < a.length; i++)
      System.out.print(a[i] + " ");
    System.out.println();
  }
  public String toString() { return i + ""; }
  public static void main(String[] args) {
    CompClass[] a = new CompClass[20];
    for(int i = 0; i < a.length; i++)
```

```
      a[i] = new CompClass(
        (int)(Math.random() *100));
    print(a);
    Arrays.sort(a);
    print(a);
    int loc = Arrays.binarySearch(a, a[3]);
    System.out.println("Location of " + a[3] +
     " = " + loc);
  }
} ///:~
```

Of course, your **compareTo( )** method can be as complex as necessary.

Lists

A **List** can be sorted and searched in the same fashion as an array. The **static** methods to sort and search a **List** are contained in the class **Collections**, but they have similar signatures as the ones in **Arrays**: **sort(List)** to sort a **List** of objects that implement **Comparable**, **binarySearch(List, Object)** to find an object in the list, **sort(List, Comparator)** to sort a **List** using a **Comparator**, and **binarySearch(List, Object, Comparator)** to find an object in that list.[1] This example uses the previously-defined **CompClass** and **AlphaComp** to demonstrate the sorting tools in **Collections**:

```
//: ListSort.java
// Sorting and searching Lists with 'Collections'
package c08.newcollections;
import java.util.*;

public class ListSort {
  public static void main(String[] args) {
    final int SZ = 20;
    // Using "natural comparison method":
    List a = new ArrayList();
    for(int i = 0; i < SZ; i++)
      a.add(new CompClass(
        (int)(Math.random() *100)));
    Collection1.print(a);
    Collections.sort(a);
```

[1] At the time of this writing, **Collections.sort( )** has been modified to use a *stable sort algorithm* (one that does not reorder equal elements).

```
    Collection1.print(a);
    Object find = a.get(SZ/2);
    int loc = Collections.binarySearch(a, find);
    System.out.println("Location of " + find +
     " = " + loc);
    // Using a Comparator:
    List b = new ArrayList();
    for(int i = 0; i < SZ; i++)
      b.add(Array1.randString(4));
    Collection1.print(b);
    AlphaComp ac = new AlphaComp();
    Collections.sort(b, ac);
    Collection1.print(b);
    find = b.get(SZ/2);
    // Must use the Comparator to search, also:
    loc = Collections.binarySearch(b, find, ac);
    System.out.println("Location of " + find +
     " = " + loc);
  }
} ///:~
```

The use of these methods is identical to the ones in **Arrays**, but you're using a **List** instead of an array.

The **TreeMap** must also order its objects according to **Comparable** or **Comparator**.

Utilities

There are a number of other useful utilities in the **Collections** class:

enumeration(Collection)	Produces an old-style **Enumeration** for the argument.
max(Collection) **min(Collection)**	Produces the maximum or minimum element in the argument using the natural comparison method of the objects in the **Collection**.
max(Collection, Comparator) **min(Collection, Comparator)**	Produces the maximum or minimum element in the **Collection** using the **Comparator**.
nCopies(int n, Object o)	Returns an immutable **List** of size **n** whose handles all point to **o**.
subList(List, int min, int max)	Returns a new **List** backed by the

	specified argument **List** that is a window into that argument with indexes starting at **min** and stopping just before **max**.

Note that **min( )** and **max( )** work with **Collection** objects, not with **List**s, so you don't need to worry about whether the **Collection** should be sorted or not. (As mentioned earlier, you *do* need to **sort( )** a **List** or an array before performing a **binarySearch( )**.)

Making a Collection or Map unmodifiable

Often it is convenient to create a read-only version of a **Collection** or **Map**. The **Collections** class allows you to do this by passing the original container into a method that hands back a read-only version. There are four variations on this method, one each for **Collection** (if you don't want to treat a **Collection** as a more specific type), **List**, **Set**, and **Map**. This example shows the proper way to build read-only versions of each:

```
//: ReadOnly.java
// Using the Collections.unmodifiable methods
package c08.newcollections;
import java.util.*;

public class ReadOnly {
  public static void main(String[] args) {
    Collection c = new ArrayList();
    Collection1.fill(c); // Insert useful data
    c = Collections.unmodifiableCollection(c);
    Collection1.print(c); // Reading is OK
    //! c.add("one"); // Can't change it

    List a = new ArrayList();
    Collection1.fill(a);
    a = Collections.unmodifiableList(a);
    ListIterator lit = a.listIterator();
    System.out.println(lit.next()); // Reading OK
    //! lit.add("one"); // Can't change it

    Set s = new HashSet();
    Collection1.fill(s);
    s = Collections.unmodifiableSet(s);
    Collection1.print(s); // Reading OK
    //! s.add("one"); // Can't change it
```

```
    Map m = new HashMap();
    Map1.fill(m, Map1.testData1);
    m = Collections.unmodifiableMap(m);
    Map1.print(m); // Reading OK
    //! m.put("Ralph", "Howdy!");
  }
} ///:~
```

In each case, you must fill the container with meaningful data *before* you make it read-only. Once it is loaded, the best approach is to replace the existing handle with the handle that is produced by the "unmodifiable" call. That way, you don't run the risk of accidentally changing the contents once you've made it unmodifiable. On the other hand, this tool also allows you to keep a modifiable container as **private** within a class and to return a read-only handle to that container from a method call. So you can change it from within the class but everyone else can only read it.

Calling the "unmodifiable" method for a particular type does not cause compile-time checking, but once the transformation has occurred, any calls to methods that modify the contents of a particular container will produce an **UnsupportedOperationException**.

Synchronizing a Collection or Map

The **synchronized** keyword is an important part of the subject of *multithreading*, a more complicated topic that will not be introduced until Chapter 14. Here, I shall note only that the **Collections** class contains a way to automatically synchronize an entire container. The syntax is similar to the "unmodifiable" methods:

```
//: Synchronization.java
// Using the Collections.synchronized methods
package c08.newcollections;
import java.util.*;

public class Synchronization {
  public static void main(String[] args) {
    Collection c =
      Collections.synchronizedCollection(
        new ArrayList());
    List list = Collections.synchronizedList(
      new ArrayList());
    Set s = Collections.synchronizedSet(
      new HashSet());
```

```
    Map m = Collections.synchronizedMap(
      new HashMap());
  }
} ///:~
```

In this case, you immediately pass the new container through the appropriate "synchronized" method; that way there's no chance of accidentally exposing the unsynchronized version.

The new collections also have a mechanism to prevent more than one process from modifying the contents of a container. The problem occurs if you're iterating through a container and some other process steps in and inserts, removes, or changes an object in that container. Maybe you've already passed that object, maybe it's ahead of you, maybe the size of the container shrinks after you call **size( )** – there are many scenarios for disaster. The new collections library incorporates a *fail fast* mechanism that looks for any changes to the container other than the ones your process is personally responsible for. If it detects that someone else is modifying the container, it immediately produces a **ConcurrentModificationException**. This is the "fail-fast" aspect – it doesn't try to detect a problem later on using a more complex algorithm.

Summary

To review the collections provided in the standard Java (1.0 and 1.1) library (**BitSet** is not included here since it's more of a special-purpose class):

1. An array associates numerical indices to objects. It holds objects of a known type so you don't have to cast the result when you're looking up an object. It can be multidimensional, and it can hold primitives. However, its size cannot be changed once you create it.

2. A **Vector** also associates numerical indices to objects – you can think of arrays and **Vector**s as random-access collections. The **Vector** automatically resizes itself as you add more elements. But a **Vector** can hold only **Object** handles, so it won't hold primitives and you must always cast the result when you pull an **Object** handle out of a collection.

3. A **Hashtable** is a type of **Dictionary**, which is a way to associate, not numbers, but *objects* with other objects. A **Hashtable** also supports random access to objects, in fact, its whole design is focused around rapid access.

4. A **Stack** is a last-in, first-out (LIFO) queue.

If you're familiar with data structures, you might wonder why there's not a larger set of collections. From a functionality standpoint, do you really *need* a larger set of collections? With a **Hashtable**, you can put things in and find them quickly, and with an **Enumeration**, you can iterate through the sequence and perform an operation on every element in the sequence. That's a powerful tool, and maybe it should be enough.

But a **Hashtable** has no concept of order. **Vector**s and arrays give you a linear order, but it's expensive to insert an element into the middle of either one. In addition, queues, dequeues, priority queues, and trees are about *ordering* the elements, not just putting them in and later finding them or moving through them linearly. These data structures are also useful, and that's why they were included in Standard C++. For this reason, you should consider the collections in the standard Java library only as a starting point, and, if you must use Java 1.0 or 1.1, use the JGL when your needs go beyond that.

If you can use Java 1.2 you should use only the new collections, which are likely to satisfy all your needs. Note that the bulk of this book was created using Java 1.1, so you'll see that the collections used through the rest of the book are the ones that are available only in Java 1.1: **Vector** and **Hashtable**. This is a somewhat painful restriction at times, but it provides better backward compatibility with older Java code. If you're writing new code in Java 1.2, the new collections will serve you much better.

Exercises

1. Create a new class called **Gerbil** with an **int gerbilNumber** that's initialized in the constructor (similar to the **Mouse** example in this chapter). Give it a method called **hop( )** that prints out which gerbil number this is and that it's hopping. Create a **Vector** and add a bunch of **Gerbil** objects to the **Vector**. Now use the **elementAt( )** method to move through the **Vector** and call **hop( )** for each **Gerbil**.

2. Modify Exercise 1 so you use an **Enumeration** to move through the **Vector** while calling **hop( )**.

3. In **AssocArray.java**, change the example so it uses a **Hashtable** instead of an **AssocArray**.

4. Take the **Gerbil** class in Exercise 1 and put it into a **Hashtable** instead, associating the name of the **Gerbil** as a **String** (the key) for each **Gerbil** (the value) you put in the table. Get an **Enumeration** for the **keys( )** and use it to move through the **Hashtable**, looking up the **Gerbil** for each key and printing out the key and telling the **gerbil** to **hop( )**.

5. Change Exercise 1 in Chapter 7 to use a **Vector** to hold the **Rodent**s and an **Enumeration** to move through the sequence of **Rodent**s. Remember that a **Vector** holds only **Object**s so you must use a cast (i.e.: RTTI) when accessing individual **Rodent**s.

6. (Intermediate) In Chapter 7, locate the **GreenhouseControls.java** example, which consists of three files. In **Controller.java**, the class **EventSet** is just a collection. Change the code to use a **Stack** instead of an **EventSet**. This will require more than just replacing **EventSet** with **Stack**; you'll also need to use an **Enumeration** to cycle through the set of events. You'll probably find it easier if at times you treat the collection as a **Stack** and at other times as a **Vector**.

7. (Challenging). Find the source code for **Vector** in the Java source code library that comes with all Java distributions. Copy this code and make a special version called **intVector** that holds only **int**s. Consider what it would take to make a special version of **Vector** for all the primitive types. Now consider what happens if you want to make a linked list class that works with all the primitive types. If parameterized types are ever implemented in Java, they will provide a way to do this work for you automatically (as well as many other benefits).