Generics and Collections in Java 5

Maurice Naftalin
naftalin@morninglight.co.uk

Philip Wadler
wadler@inf.ed.ac.uk

This version is a draft. Please send comments to the addresses above.

Contents

I Introduction to Generics 7

1 Getting started 11
- 1.1 Generics . . . 12
- 1.2 Boxing and unboxing . . . 14
- 1.3 Foreach . . . 16
- 1.4 Generic methods and varargs . . . 17
- 1.5 Assertions . . . 19
- 1.6 Summing up . . . 19

2 Subtyping and wildcards 21
- 2.1 Subtyping and The Substitution Principle . . . 21
- 2.2 Wildcards with extends . . . 22
- 2.3 Wildcards with super . . . 23
- 2.4 The Get and Put Principle . . . 24
- 2.5 Arrays . . . 27
- 2.6 Wildcards vs. type parameters . . . 29
- 2.7 Wildcard capture . . . 31
- 2.8 Restrictions on wildcards . . . 32

3 Comparison and bounds 35
- 3.1 Comparable . . . 35
- 3.2 Maximum of a collection . . . 37
- 3.3 A fruity example . . . 39
- 3.4 Comparator . . . 42
- 3.5 Enumerated types . . . 45
- 3.6 Multiple bounds . . . 46
- 3.7 Bridges . . . 48
- 3.8 Static members . . . 49
- 3.9 Nested classes . . . 51
- 3.10 How erasure works . . . 52

4 Evolution not revolution 57
- 4.1 Legacy library with legacy client . . . 57
- 4.2 Generic library and generic client . . . 58

4.3 Generic library with legacy client . . . 58
4.4 Legacy library with generic client . . . 62
4.5 Evolving a library — minimal changes . . . 62
4.6 Evolving a library — stubs . . . 63
4.7 Conclusions . . . 66

5 Reification 67
5.1 Reifiable types . . . 67
5.2 Instance tests and casts . . . 68
5.3 Exception handling . . . 72
5.4 Array creation . . . 75
5.5 The Principle of Truth in Advertising . . . 78
5.6 The Principle of Indecent Exposure . . . 81
5.7 How to define ArrayList . . . 83
5.8 Array creation and varargs . . . 85
5.9 Summing up . . . 87

6 Reflection 89
6.1 Generics for reflection . . . 89
6.2 Reflected types are reifiable types . . . 91
6.3 Reflection for primitive types . . . 92
6.4 A generic reflection library . . . 93
6.5 Reflection for generics . . . 94
6.6 Reflecting generic types . . . 95

II Collections 103

7 The Main Interfaces of the Java Collections Framework 107

8 Preliminaries 109
8.1 Iterable and Iterators . . . 109
8.2 Comparable and Comparator . . . 111
8.3 Implementations . . . 112
8.4 Efficiency and the O-notation . . . 113
8.5 Contracts . . . 115
8.6 Collections and Thread-Safety . . . 116
8.6.1 Thread Safety and Iterators . . . 117

9 The `Collection` Interface 119
9.1 Using the Methods of `Collection` . . . 121
9.2 Implementing `Collection` . . . 125
9.2.1 `AbstractCollection` subclasses . . . 128

10 Sets **131**
10.1 Implementing Set . . . 131
10.1.1 HashSet . . . 134
10.1.2 CopyOnWriteArraySet . . . 136
10.1.3 LinkedHashSet . . . 136
10.2 SortedSet . . . 137
10.2.1 TreeSet . . . 139
10.3 Comparison of Set Implementations . . . 141

11 Queues **143**
11.1 Using the Methods of Queue . . . 144
11.2 Implementing Queue . . . 145
11.2.1 LinkedList . . . 145
11.2.2 AbstractQueue . . . 147
11.2.3 PriorityQueue . . . 147
11.2.4 ConcurrentLinkedQueue . . . 148
11.3 BlockingQueue . . . 148
11.3.1 Implementing BlockingQueue . . . 150

12 Lists **155**
12.1 Using the Methods of List . . . 156
12.2 Implementing List . . . 159
12.2.1 AbstractList . . . 160
12.2.2 ArrayList . . . 163
12.2.3 AbstractSequentialList . . . 165
12.2.4 LinkedList . . . 165
12.2.5 CopyOnWriteArrayList . . . 166
12.3 Comparison of List Implementations . . . 166

13 Maps **169**
13.1 Using the Methods of Map . . . 170
13.2 Implementing Map . . . 171
13.2.1 AbstractMap . . . 172
13.2.2 HashMap . . . 173
13.2.3 LinkedHashMap . . . 173
13.2.4 WeakHashMap . . . 174
13.2.5 IdentityHashMap . . . 175
13.3 SortedMap . . . 176
13.3.1 TreeMap . . . 178
13.4 ConcurrentMap . . . 178
13.4.1 ConcurrentHashMap . . . 178
13.5 Comparison of Map Implementations . . . 179

14 Special Implementations **181**
14.1 Pre-populated Collections . . . 181
14.2 Decorated Collections . . . 182
14.2.1 Unmodifiable Collections . . . 182
14.2.2 Synchronized Collections . . . 182
14.2.3 Checked Collections . . . 183

15 Generic Algorithms **185**
15.1 Changing the Order of `List` Elements . . . 185
15.2 Finding Extreme Values in a `Collection` . . . 186
15.3 Finding Specific Values in a `List` . . . 186
15.4 Changing the Contents of a `List` . . . 187

III Using Generics Effectively **189**

16 Style and Technique in Generic Programming **191**
16.1 Avoid Unchecked Warnings . . . 191
16.2 Make Careful Use of `@SuppressWarnings` . . . 193
16.3 Favour Collections Over Arrays . . . 194
16.4 Favour Wildcards over Type Parameters for Generic Methods . . . 195
16.5 Use Generics for Typesafe Covariant Hierarchies . . . 197
16.6 Don't Use Generics to Enforce Immutability . . . 201

17 Evolution **203**
17.1 Use Checked Collections . . . 203
17.2 Provide Wrappers to Legacy Code . . . 205
17.3 Choose Appropriate Types for Generification . . . 208
17.4 Maintain Binary Compatibility when Generifying Existing Classes . . . 209

Part I

Introduction to Generics

Java 5 supports *generics*, the most significant change to the language since the addition of inner classes in Java 1.2 — some would say the most significant change to the language ever.

Say you wish to process lists. Some may be lists of bytes, others lists of strings, and yet others lists of lists of strings. In Java 1.4 this is simple. You can represent all three by the same class, called `List`, which has elements of class `Object`.

list of integers	`List`
list of strings	`List`
list of list of strings	`List`

In order to keep the language simple, you are forced to do some of the work yourself: you must keep track of the fact that you have a list of integers (or strings or lists of strings), and when you extract an element from the list you must cast it from `Object` back to `Integer` (or `Number` or `List`). For instance, the Collections Framework in Java 1.4 makes extensive use of this idiom.

As Einstein said, everything should be as simple as possible but no simpler. And some might say the above is too simple. In Java 5 it is possible to represent lists a more direct way.

list of integers	`List<Integer>`
list of string	`List<String>`
list of list of strings	`List<List<String>>`

Now the compiler keeps track of whether you have a list of integers (or numbers or lists of strings), and no explicit cast back to `Integer` (or `Number` or `List<String>`) is required. In some ways, this is similar to *generics* in Ada or *templates* in C++, but the actual inspiration is *parametric polymorphism* as found in functional languages such as ML and Haskell.

The design of generics for Java is influenced by a number of previous proposals, notably GJ by Bracha, Odersky, Stoutamire, and Wadler; the addition of wildcards to GJ proposed by Igarashi and Viroli; and further development of wildcards by Torgersen, Hansen, Ernst, von der Ahé, Bracha, and Gafter. Design of generics was carried out under the Java Community Process by a team led by Bracha, and including Odersky, Thorup, and Wadler (as parts of JSR 14 and JSR 201). Odersky's GJ compiler is the basis of Sun's current `javac` compiler.

Chapter 1

Getting started

Newton famously said "If I have seen farther than others, it is because I stand on the shoulders of giants". The best programmers live by this motto, building on existing frameworks and reusable code where appropriate. The Java Collections Framework provides reusable interfaces and implementations for a number of common collection types, including lists, sets, queues, and maps. There is also a framework for comparing values, useful in sorting or building ordered trees. (Of course, not all programmers exploit reuse. As Hamming said of computer scientists, "Instead of standing on each others' shoulders, we stand on each others' toes.")

Generics make it easier to use the Collections Framework, and the Collections Framework provides many examples of the use of generics; one might say that the two are made for each other. Here we will explain generics by examining their use in the context of the collections. A comprehensive introduction to collections appears in Part II. This section uses collections to illustrate key features of generics.

Generics and collections work well with a number of other new features introduced in Java 5, including boxing and unboxing, functions that accept a variable number of arguments, and a new form of loop. We begin with an example that illustrates all of these. As we shall see, their combination is *synergistic*: the whole is greater than the sum of its parts.

Taking that as our motto, let's do something simple with sums: put three numbers into a list and add them together. Here is how to do it in Java 5.

```
List<Integer> ints = Arrays.asList(1,2,3);
int s = 0;
for (int n : ints) { s += n; }
assert s == 6;
```

You can probably read this code without much explanation, but let's touch on the key new features. The interface `List` and the class `Arrays` are part of the Collections Framework (both are found in the package `java.util`). The type `List` is now *generic*, one write `List<E>` to indicate a list with elements of type `E`. Here we write `List<Integer>` to indicate that the elements of the list belong to the class `Integer`, the wrapper class that corresponds to the primitive type `int`. Boxing and unboxing operations, to convert from the primitive type to the wrapper class, are automatically inserted. The static method `asList` takes any number of arguments, places them into an array, and returns a new list backed by the array. The new form of *foreach* loop is used to bind a variable succesively to each element of the list, and the loop body adds these into the sum. The assertion statement, introduced in Java 1.4, is used to check that the sum is correct; when assertions are enabled, it throws an error if the condition does not evaluate to true.

Here is how the same code looks in Java 1.4.

```
List ints = Arrays.asList( new Integer[] {
  new Integer(1), new Integer(2), new Integer(3)
} );
int s = 0;
for (Iterator it = ints.iterator(); it.hasNext(); ) {
  int n = ((Integer)it.next()).intValue();
  s += n;
}
assert s == 6;
```

Reading this code is not quite so easy. Without generics, there is no way to indicate in the type declaration what kind of elements are intended to be stored in the list, so instead of writing `List<Integer>` you write `List`. Now it is the coder rather than the compiler that is responsible for remembering the type of the list elements, so you must write the cast to `(Integer)` when extracting elements from the list. Without boxing and unboxing, you must explicitly allocate each object belonging to the wrapper class `Integer` and use the `intValue` method to extract the corresponding primitive `int`. Without functions that accept a variable number of arguments, you must explicitly allocate an array to pass to the `asList` method. Without the new form of loop, you must explicitly declare an iterator and advance it through the list.

By the way, here is how to do the same thing with an array in Java 1.4.

```
int[] ints = new int[] { 1,2,3 };
int s = 0;
for (int i = 0; i < ints.size; i++) { s += ints[i]; }
assert s == 6;
```

This is slightly longer than the corresponding code that uses generics and collections, is arguably a bit less readable, and is certainly less flexible. Collections let you easily grow or shrink the size of the collection, or switch to a different representation when appropriate, such as a linked list or hash table or ordered tree. The introduction of generics, boxing and unboxing, *foreach* loops, and *varargs* in Java mark the first time where using collections is just as simple, perhaps even simpler, than using arrays.

Now let's look at each of these features in a little more detail.

1.1 Generics

An interface or class may be declared to take one or more type parameters, which are written in angle brackets, and should be supplied when you declare a variable belonging to the interface or class, or when you create a new instance of a class.

We saw one example above. Here is another.

```
List<String> words = new ArrayList<String>();
words.add("Hello ");
words.add("world!");
String s = words.get(0)+words.get(1);
assert s.equals("Hello world!");
```

In the Collections Framework, the class `ArrayList<E>` implements the interface `List<E>`. This trivial code fragment declares variable `words` to contain a list of string, creates an instance of a `ArrayList`, adds two strings to the list, and gets them out again.

In Java 1.4, the same code would be written as follows.

```
List words = new ArrayList();
words.add("Hello ");
words.add("world!");
String s = ((String)words.get(0))+((String)words.get(1))
assert s.equals("Hello world!");
```

Without generics, the type parameters are omitted, but one must explicitly cast whenever an element is extracted from the list.

In fact, the byte code compiled from the two sources above will be identical. We say that generics are implemented by *erasure*, because the types `List<Integer>`, `List<String>`, and `List<List<String>>`, are at run-time all represented by the same type, `List`. We also use *erasure* to describe the process that converts the first program to the second. Erasure is a slight misnomer, since the process erases type parameters but adds casts.

Generics implicitly perform the same cast that is explicitly performed without generics. If such casts could fail, it might be hard to debug code written with generics. This is why it is reassuring that generics come with the following *cast-iron guarantee*: the implicit casts added by erasure never fail. There is also some fine print on this guarantee: it applies only when no *unchecked warnings* have been issued. Later, we will discuss at some length what causes unchecked warnings to be issued, and explain the few situations where they are unavoidable.

Implementing generics by erasure has a number of important effects. It keeps things simple, in that generics do not add anything fundamentally new. It keeps things small, in that there is exactly one implementation of `List`, not one version for each type. And it eases evolution, since the same library can be accessed in both non-generic and generic form.

This last point is worth some elaboration. It means that one doesn't get nasty problems of maintaining two versions of the libraries, one that works with non-generic legacy code and another that works with generic Java 5 code. And it means that you can evolve your code to use generics one step at a time. At the byte-code level, code that doesn't use generics looks just like code that does. There is no need to switch to generics all at once, you can evolve your code by updating just one package, class, or method at a time to start using generics.

Another consequence of implementing generics by erasure is that array types differ in key ways from parameterized types. Executing

```
new String[size]
```

allocates an array, and stores in the array an indication that its elements are all of type `String`. In contrast, executing

```
new ArrayList<String>()
```

allocates a list, but does not store in the list any kind of indication of the type of its elements. In the jargon, we say that Java *reifies* array component types, but does not reify list element types (or other generic types). Later, we will see how this design eases evolution (see Chapter 4) but complicates casts, instance tests, and array creation (see Chapter 5).

Generics vs. templates Generics in Java resemble templates in C++. There are just two important things to bear in mind about the relationship between Java generics and C++ templates: syntax and semantics. The syntax is deliberately similar, the semantics is deliberately different.

Syntactically, angle brackets were chosen since they are familiar to C++ users. However, there is one difference in syntax. In C++, nested parameters require extra spaces, so you see things like this:

```
List< List<String> >.
```

In Java, no spaces are required, and it's fine to write this:

```
List<List<String>>.
```

You may use extra spaces if you prefer, but they're not required. (The problem in C++ arises because `>>` without the space denotes the right shift operator. Java fixes the problem by a trick in the grammar.)

Semantically, Java generics are defined by *erasure*, whereas C++ templates are defined by *expansion*. In C++ templates, each instance of a template at a new type is compiled seperately. If you use a list of integer, a list of string, and a list of list of string, then there will be three versions of the code. If you use lists of a hundred different types, there will be a hundred versions of the code, a problem known as *code bloat*. In Java, no matter how many types of list you use, there is always one version of the code, so no bloat.

Expansion may lead to more efficient implementation than erasure, since it offers more opportunities for optimization, particularly at primitive types such as `int`. For code manipulating large amounts of data – for instance, large arrays in scientific computing – this difference may be significant. However, in practice, for most purposes the difference in efficiency is not important, whereas the problems caused by code bloat can be significant.

1.2 Boxing and unboxing

Recall that every type in Java is either a *reference* type or a *primitive* type. A reference type is any class, instance, or array type. All reference types are subtypes of class `Object`, and any variable of reference type may be set to the value `null`. There are eight primitve types, and each of these has a corresponding library class of reference type. The library classes are located in the package `java.lang`.

primitive	reference	primitive	reference
`byte`	`Byte`	`bool`	`Boolean`
`short`	`Short`	`char`	`Character`
`int`	`Integer`	`float`	`Float`
`long`	`Long`	`double`	`Double`

Conversion of a primitive type to the corresponding reference type is called *boxing* and conversion of the reference type to the corresponding primitive type is called *unboxing*.

Java 5 automatically inserts boxing and unboxing coercions where appropriate. If an expression e of type `int` appears where a value of type `Integer` is expected, boxing converts it to the static method call `Integer.valueOf(`e`)`. (This has the same effect as calling `new Integer(`e`)`, except it may cache frequently occuring values.) If an expression e' of type `Integer` appears where a value of type `int` is expected, unboxing converts it to the expression e'`.intValue()`. For example, the sequence

```
List<Integer> ints = new ArrayList<Integer>();
ints.add(1);
int n = ints.get(0);
```

is equivalent to the sequence

```
List<Integer> ints = new ArrayList<Integer>();
ints.add(Integer.valueOf(1));
int n = ints.get(0).intValue();
```

Here again is the code to find the sum of a list of integers, conveniently packaged as a static method.

```
public static int sum (List<Integer> ints) {
    int s = 0;
    for (int n : ints) { s += n; }
    return s;
}
```

Why does the argument have type `List<Integer>` and not `List<int>`? Because type parameters must always be instantiated to reference types, not primitive types. Why does the result have type `int` and not `Integer`? Because result types may be either primitive or reference types, and it is more efficient to use the former than the latter. Unboxing occurs when each `Integer` in the list `ints` is bound to the variable `n` of type `int`.

We could rewrite the method, replacing each occurrence of `int` by `Integer`.

```
public static Integer sum_Integer (List<Integer> ints) {
  Integer s = 0;
  for (Integer n : ints) { s += n; }
  return s;
}
```

This code compiles, but performs a lot of needless work. Each iteration of the loop unboxes the values in `s` and `n`, performs the addition, and boxes up the result again. Measurements show this routine is about 60% slower.

One subtlety of boxing and unboxing is that == is defined differently on primitive and reference types. On type `int` it is equality of values, while on type `Integer` it is defined by object identity. So both of the following assertions succeed, using Sun's JVM for Java 5.

```
List<Integer> bigs = Arrays.asList(100,200,300);
assert sum_Integer(bigs) == sum(bigs);
assert sum_Integer(bigs) != sum_Integer(bigs);
```

In the first assertion, unboxing causes values to be compared, so the results are equal. In the second assertion, there is no unboxing, and the two method calls return distinct `Integer` objects, so the results are unequal — even though both `Integer` objects represent the same value, 600.

A further subtlety is that boxed values may be cached. In Sun's implementation, boxing an `int` value in the range from -128 to 127 returns an object with not just the same value, but also the same identity. Hence, in contrast to the above, we have the following.

```
List<Integer> smalls = Arrays.asList(1,2,3);
assert sum_Integer(smalls) == sum(smalls);
assert sum_Integer(smalls) == sum_Integer(smalls);
```

This is because 6 is smaller than 128, so boxing the value 6 always returns exactly the same object. In general, it is not specified whether boxing the same value twice should return identical or distinct objects, so the inequality assertion above may either fail or succeed depending on the implementation.

1.3 Foreach

Here again is our code that computes the sum of a list of integers.

```
List<Integer> ints = Arrays.asList(1,2,3);
int s = 0;
for (int n : ints) { s += n; }
assert s == 6;
```

The loop in the third line is called a *foreach* loop even though it is written with the keyword *for*. It is equivalent to the following.

```
for (Iterator<Integer> it = ints.iterator(); it.hasNext(); ) {
    int n = it.next();
    s += n;
}
```

The emphasized code corresponds to what was written by the user, and the unemphasized code is added in a systematic way by the compiler. It introduces the variable `it` of type `Iterator<Integer>` to iterate over the list `ints` of type `List<Integer>`. In general, the compiler invents a new name guaranteed not to clash with any name already in the code. Note that unboxing occurs when the expression `it.next()` of type `Integer` is assigned to the variable `n` of type `int`.

The *foreach* loop can be applied to any object that implements the interface `Iterable<E>` (in package `java.lang`), which in turn refers to the interface `Iterator<E>` (in package `java.util`). These define the methods `iterator`, `hasNext`, and `next`, used by the translation of the *foreach* loop. (Iterators also have a method `remove`, which is not used by the translation.)

```
interface Iterable<E> {
   public Iterator<E> iterator ();
}
interface Iterator<E> {
    public boolean hasNext ();
    public E next ();
    public void remove ();
}
```

All collections, sets, and lists in the Collections Framework implement the `Iterable<E>` interface; and classes defined by other vendors or users may implement it as well.

The *foreach* loop may also be applied to an array.

```
public static int sum_array (int[] a) {
    int s = 0;
    for (int n : a) { s += n; }
    return s;
}
```

The foreach loop was deliberately kept simple, and only catches the most common case. One needs to explicitly introduce an iterator if one wishes to use the `remove` method or to iterate over more than one list in parallel. Here is a method that removes negative elements from a list of doubles.

```
public static void removeNegative(List<Double> v) {
    for (Iterator<Double> it = v.iterator(); it.hasNext();) {
        if (it.next() < 0) it.remove();
    }
}
```

Here is a method to compute the dot product of two vectors, represented as lists of doubles, both of the same length. (Given two vectors $u_1, \ldots, u_n$ and $v_1, \ldots, v_n$ it computes $u_1 * v_1 + \cdots + u_n * v_n$.)

```
public static double dot(List<Double> u, List<Double> v) {
    if (u.size() != v.size())
        throw new IllegalArgumentException("different sizes");
    double d = 0;
    Iterator<Double> u_it = u.iterator();
    Iterator<Double> v_it = v.iterator();
    while (u_it.hasNext()) {
        assert v_it.hasNext();
        d += u_it.next() * v_it.next();
    }
    assert !v_it.hasNext();
    return d;
}
```

Two iterators, `u_it` and `v_it` advance across the lists `u` and `v` in lock step. The loop condition checks only the first iterator, but the assertions confirm that the second list has the same length as the first.

1.4 Generic methods and varargs

Here is a method that accepts an array of any type and converts it to a list,

```
public static <E> List<E> asList (E[] arr) {
    List<E> list = new ArrayList<E>();
    for (E elt : arr) list.add(elt);
    return list;
}
```

The static method `asList` accepts an array of type `E[]` and returns a list of type `List<E>`, and does so for *any* type `E`. This is indicated by writing `<E>` at the beginning of the method signature, which declares `E` as a new type variable. A method which declares a type variable in this way is called a *generic method*. The scope of the type variable `E` is local to the method itself; it may appear in the method signature and the method body, but not outside the method.

The method may be invoked as follows.

```
List<Integer> ints = asList(new Integer[] { 1, 2, 3 });
List<String> words = asList(new String[] { "hello", "world" });
```

In the first line, boxing converts 1, 2, 3 from `int` to `Integer`.

Packing the arguments into an array is cumbersome. The *vararg* feature permits a special more convenient syntax for case when the last argument of a method is an array. To use this feature, we replace `E[]` by `E...` in the method declaration.

```
public static <E> List<E> asList (E... arr) {
    List<E> list = new ArrayList<E>();
    for (E elt : arr) list.add(elt);
    return list;
}
```

Now the method may be invoked as follows.

```
List<Integer> ints = asList(1, 2, 3);
List<String> words = asList("hello", "world");
```

This is just a shorthand for what we wrote above. At run-time, the arguments are packed into an array which is passed to the method, just as previously.

Any number of arguments may precede a last *vararg* argument. Here is a method that accepts a list and adds all the additional arguments to the end of the list.

```
public static <E> void addAll (List<E> list, E... arr) {
    for (E elt : arr) list.add(elt);
}
```

Whenever a *vararg* is declared, one may pass either a list of arguments to be implicitly packed into an array, or explicitly pass the array directly. Thus, the above method may be invoked as follows.

```
List<Integer> ints = new ArrayList<Integer>();
addAll(ints, 1, 2);
addAll(ints, new Integer[] { 3, 4 });
assert ints.toString().equals("[1, 2, 3, 4]");
```

We will later see that attempting to create an array containing a generic type must always issue an *unchecked* warning. Since *varargs* always create an array, they should be used only when the argument does not have a generic type (see Section 5.8).

In the examples above, the type parameter to the generic method is inferred, but it may also be given explicitly. This may be particularly helpful when an argument is empty, since then the type information might not otherwise be inferred. The following initializes a list of integers to be the empty list.

```
    List<Integer> ints = this.<Integer>asList();
```

Here the type parameter is required, because there is not enough information to infer the type of the right-hand side otherwise. In Java, the type of the left-hand side of an assignment is never used when determining the type of the right-hand side: the type of an expression depends only on the parts of the expression, not its context.

When a type parameter is passed to a generic method invocation it appears in angle brackets to the left, just as in the method declaration. The Java grammar requires that type parameters may only be in method invocations that use a dotted form. Here we have assumed the method `asList` is defined in the same class that invokes the code. We cannot shorten it as follows.

```
    List<Integer> ints = <Integer>asList();  // compile-time error
```

The above is illegal because it would confuse the parser.

The Collection Framework defines static methods `java.util.Arrays.asList` and `java.util.Collection.addAll` similar to the ones above. The Collection Framework version of `asList` does not return an ordinary `ArrayList`, but instead a special form that is backed by a given array; and its version of `addAll` acts on general collections, not just lists.

1.5 Assertions

We clarify our code by liberal use of the *assert* statement. Each occurrence of *assert* is followed by a boolean expression that is expected to evaluate to `true`. If assertions are enabled and the expression evaluates to `false` then an `AssertionError` is thrown, including an indication of where the error occurred. Assertions are enabled by invoking the JVM with the `-ea` or `-enableAssertion` flag. Since assertions may not be enabled, an assertion should never have side effects upon which any non-assertion code depends.

1.6 Summing up

We have seen how generics, boxing and unboxing, *foreach* loops, and *varargs* work together to make Java code easier to write, illustrating this through the use of the Collections Framework.

Chapter 2

Subtyping and wildcards

Now that we've covered the basics, we can start to cover more advanced features of generics, such as subtyping and wildcards. In this section we'll review how subtyping works, and see how wildcards let you use subtyping in connection with generics. We'll illustrate our points with examples from the collections library.

2.1 Subtyping and The Substitution Principle

Subtyping is a key feature of object-oriented languages such as Java. In Java, one type is a *subtype* of another if it is related by an *extends* or *implements* clause. Here are some examples.

`Integer`	is a subtype of	`Number`
`Double`	is a subtype of	`Number`
`ArrayList<E>`	is a subtype of	`List<E>`
`List<E>`	is a subtype of	`Collection<E>`
`Collection<E>`	is a subtype of	`Iterable<E>`

Subtyping is transitive, meaning that if one type is a subtype of a second, and the second is a subtype of a third, then the first is a subtype of the third. So from the last two lines above it follows that `List<E>` is a subtype of `Iterable<E>`. If one type is a subtype of another, we also say that the second is a *supertype* of the first. Every type is a subtype of `Object`, and `Object` is a supertype of every type. We also say, trivially, that every type is a subtype of itself.

The *Substitution Principle* tells us that wherever a value of one type is expected one may provide a value of any subtype of that type.

> *Substitution Principle*. A variable of a given type may be assigned a value of any subtype of that type, and a method with a parameter of a given type may be invoked with an argument of any subtype of that type.

Consider the interface `Collection<E>`. One of its methods is `add`, which takes a parameter of type `E`.

```
interface Collections<E> {
    boolean add(E elt);
    ...
}
```

According to the Substitution Principle, if we have a collection of numbers, then we may add an integer or a double to it, because `Integer` and `Double` are subtypes of `Number`.

```
List<Number> nums = new ArrayList<Number>();
nums.add(2);
nums.add(3.14);
assert nums.toString().equals("[2, 3.14]");
```

Here, subtyping is used in two ways for each method call. The first call is permitted because `nums` has type `List<Number>`, which is a subtype of `Collection<Number>`, and `2` has type `Integer` (thanks to boxing), which is a subtype of `Number`. Similarly for the second call. In both calls, the `E` in `List<E>` is taken to be `Number`.

It may seem reasonable to expect that since `Integer` is a subtype of `Number`, it follows that `List<Integer>` is a subtype of `List<Number>`. But this is *not* the case, because the Substitution Principle would rapidly get us into trouble. It is not always safe to assign a value of type list of integer to a variable of type list of number. Consider the following code fragment.

```
List<Integer> ints = Arrays.asList(1,2);
List<Number> nums = ints;  // compile-time error
nums.add(3.14);
assert ints.toString().equals("[1, 2, 3.14]");  // uh oh!
```

This code assigns variable `ints` to point point at a list of integers, and then assigns `nums` to point at the *same* list of integers; hence the assignment in the third line adds the floating point number to this list, as shown in the fourth line. This must not be allowed! The problem is prevented by observing that here the Substitution Principle does *not* apply: the assignment on the second line is *not* allowed because `List<Integer>` is *not* a subtype of `List<Number>`, and the compiler reports that the second line is in error.

Can we do it the other way around, and take `List<Number>` to be a subtype of `List<Integer>`? No, that doesn't work either, as shown by the following.

```
List<Number> nums = Arrays.asList(2.78, 3.14);
List<Integer> ints = nums;  // compile-time error
assert ints.toString().equals("[2.78, 3.14]");  // uh oh!
```

So `List<Integer>` is not a subtype of `List<Number>`, nor is `List<Number>` a subtype of `List<Integer>`; all we have is the trivial case, where `List<Integer>` is a subtype of itself, and we also have that `List<Integer>` is a subtype of `Collection<Integer>`.

Arrays have a quite different behaviour, where `Integer[]` *is* a subtype of `Number[]`. We will compare the treatment of lists and arrays later (see Section 2.5).

Sometimes we would like lists to behave more like arrays, where we will accept not only a list with elements of a given type, but also a list with elements of any subtype of a given type. For this purpose we use *wildcards*.

2.2 Wildcards with extends

Another method in the `Collection` interface is `addAll`, which adds all of the members of one collection to another collection.

```
interface Collection<E> {
    ...
    boolean addAll(Collection<? extends E> c);
    ...
}
```

Clearly, given a collection of elements of type `E`, it is ok to add all members of another collection with elements of type `E`. What the quizzical phrase "`? extends E`" means is that it is also ok to add all members of a collection with elements of any type that is a *subtype* of `E`. The question mark is called a *wildcard*, since it stands for any type that is a subtype of `E`.

Here is an example. We create an empty list of numbers, and add to it first a list of integers and then a list of doubles.

```
List<Number> nums = new ArrayList<Number>();
List<Integer> ints = Arrays.asList(1, 2);
List<Double> dbls = Arrays.asList(2.78, 3.14);
nums.addAll(ints);
nums.addAll(dbls);
assert nums.toString().equals("[1, 2, 2.78, 3.14]");
```

The first call is permitted because `nums` has type `List<Number>`, which is a subtype of `Collection<Number>`, and `ints` has type `List<Integer>`, which is a subtype of `Collection<? extends Number>`. Similarly for the second call. In both calls, `E` is taken to be `Number`. If the method signature had been written without the wildcard then the calls would not have been permitted.

We can also use wildcards when declaring variables. Here is a variant of the example at the end of the preceding section, where we add a wildcard to the second line.

```
List<Integer> ints = Arrays.asList(1,2);
List<? extends Number> nums = ints;
nums.add(3.14);  // compile-time error
assert ints.toString().equals("[1, 2, 3.14]");  // uh oh!
```

Before, the second line raised a compile-time error (because `List<Integer>` is not a subtype of `List<Number>`), but the third line was fine (because one can add any number to a variable of type `List<Number>`). Now, the second line is fine (because `List<Integer>` is a subtype of `List<? extends Number>`), but the third line raises a compile time error. As before, the fourth line shows why one of the preceding lines had better be illegal!

In general, if a structure contains elements with a type of the form "`? extends E`", then we can get elements out of the structure, but we cannot put elements into the structure. To put elements into the struture we need another kind of wildcard, as explained in the next section.

2.3 Wildcards with super

Here is a method that copies into a destination list all of the elements from a source list, from the convenience class `Collections`.

```
public static <T> void copy(List<? super T> dst,
                            List<? extends T> src) {
    for (int i = 0; i < src.length(); i++) {
        dst.set(i, src.get(i));
    }
}
```

What the quizzical phrase "`? super T`" means is that the destination list may have elements of any type that is a *supertype* of `T`, just as the source list may have elements of any type that is a *subtype* of `T`.

Here is an example, where we copy a list of integers into a list of objects.

```
List<Object> objs = Arrays.<Object>asList(2, 3.14, "four");
List<Integer> ints = Arrays.asList(5, 6);
Collections.<Number>copy(objs, ints);
assert objs.toString().equals("[5, 6, four]");
```

The call is permitted because `objs` has type `List<Object>` which is a subtype of `List<? super Number>` (since `Object` is a supertype of `Number`, as required by a `super` wildcard) and `ints` has type `List<Integer>` which is a subtype of `List<? extends Number>` (since `Integer` is a subtype of `Number`, as required by an `extends` wildcard) Here `T` is taken to be `Number`, as indicated in the call. If the method signature had been written without the wildcards then the call would not have been permitted.

We have a choice to write the call several ways, all of which type check and all of which have the same effect.

```
Collections.<Object>copy(objs, ints);    // (a)
Collections.<Number>copy(objs, ints);    // (b)
Collections.<Integer>copy(objs, ints);   // (c)
Collections.copy(objs, ints);            // (d)
```

The last call leaves the type parameter implicit; it is taken to be `Integer`, since that is the smallest choice that works.

We could have omitted one of the two wildcards, at the cost of some flexibility. If we omitted the first wildcard then only calls (a) and (d) would work, with the type parameter taken to be `Object` in (d); and if we omitted the second wildcard then only calls (c) and (d) would work, with the type parameter taken to be `Integer` in (d). If we omitted *both* wildcards, then none of the calls would work.

2.4 The Get and Put Principle

It is good practice to insert wildcards whenever possible. But how do you decide *which* to use? Where should you use `extends` and where should you use `super` and where is it inappropriate to use a wildcard at all?

Fortunately, there is a simple principle that determines which is appropriate.

> *The Get and Put Principle*. Use an `extends` wildcard when you only *get* values out of a structure, use a `super` wildcard when you only *put* values into a structure, and don't use a wildcard when you *both* get and put.

We already saw this principle at work in the signature of the copy method.

```
public static <T> void copy(List<? super T> dest,
                            List<? extends T> src);
```

The method gets values out of the source, so it is declared with an `extends` wildcard, and puts values into the destination, so it is declared with a `super` wildcard.

Whenever you use an iterator, you get values out of a structure, so use an `extends` wildcard. Here is a method that takes a collection of numbers, converts each to a double, and sums them up.

```
public static double sum(Collection<? extends Number> nums) {
    double s = 0.0;
    for (Number num : nums) s += num.doubleValue();
    return s;
}
```

Since this uses `extends`, all of the following calls are legal.

```
List<Integer> ints = Arrays.asList(1,2,3);
assert sum(ints) == 6.0;

List<Double> doubles = Arrays.asList(2.78,3.14);
assert sum(doubles) == 5.92;

List<Number> nums = Arrays.<Number>asList(1,2,2.78,3.14);
assert sum(nums) == 8.92;
```

The first two calls would not be legal if `extends` was not used.

Whenever you use the `add` method, you are putting values into a structure, so use a `super` wildcard. Here is a method that takes a collection of numbers and an integer `n`, and puts the first `n` integers, starting from zero, into the collection.

```
public static void count(Collection<? super Integer> ints,
                         int n) {
    for (int i = 0; i < n; i++) ints.add(i);
}
```

Since this uses `super`, all of the following calls are legal.

```
List<Integer> ints = new ArrayList<Integer>();
count(ints, 5);
assert ints.toString().equals("[0, 1, 2, 3, 4]");

List<Number> nums = new ArrayList<Number>();
count(nums, 5);  nums.add(5.0);
assert nums.toString().equals("[0, 1, 2, 3, 4, 5.0]");

List<Object> objs = new ArrayList<Object>();
count(objs, 5);  objs.add("five");
assert objs.toString().equals("[0, 1, 2, 3, 4, five]");
```

The last two calls would not be legal if `super` was not used.

Whenever you both put values into and get values out of the same structure, you should not use a wildcard.

```
public static double sumcount (Collection<Number> nums, int n) {
    count(nums, n);
    return sum(nums);
}
```

The collection is passed both to `sum` and `count`, so its element type must both extend `Number` (as `sum` requires) and be super to `Integer` (as `count` requires). The only two classes that satisfy both these constraints are `Number` and `Integer`, and we have picked the first of these. Here is a sample call.

```
List<Number> nums = new ArrayList<Number>();
double sum = sumcount(nums);
assert sum == 6;
```

Since there is no wildcard, the argument must be a collection of `Number`.

If you don't like having to choose between `Number` and `Integer`, it might occur to you that if only Java let you write a wildcard with both `extends` and `super`, then you would not need to choose. For instance, one could write the following.

```
double sumcount(Collection<? super Integer,   extends Number> coll, int n)
// not legal Java!
```

Then we could call `sumcount` on either a collection of integers or a collection of numbers. But Java *doesn't* permit this. The only reason for outlawing it is simplicity, and conceivably Java might support such notation in the future. But until then, if you need to both put and get, don't use wildcards.

The Get and Put Principle also works the other way around. If an `extend` wildcard is present, then pretty much all you will be able to do is get but not put values of that type; and if a `super` wildcard is present, then pretty much all you will be able to do is put but not get values of that type.

For example, consider the following code fragment using a list declared with an `extends` wildcard.

```
List<Integer> ints = Arrays.asList(1,2,3);
List<? extends Number> nums = ints;
double dbl = sum(nums);  // ok
nums.add(3.14);  // compile-time error
```

The call to `sum` is fine, because we are getting values from the list, but the call to `add` is not, because we are putting a value into the list. This is just as well, since otherwise we could add a double to a list of integers!

Conversely, consider the following code fragment using a list declared with a `super` wildcard.

```
List<Object> objs = Arrays.<Object>asList(1,"two");
List<? super Integer> ints = objs;
ints.add(3);  // ok
double dbl = sum(ints);  // compile-time error
```

Now the call to `add` is fine, because we are putting a value into the list, but the call to `sum` is not, because we are getting a value from the list. This is just as well, because the sum of a list containing a string makes no sense!

The exception proves the rule, and each of these rules has one exception. You cannot put anything into a type declared with an `extends` wildcard — but you can put in the value `null`, which belongs to every reference type.

```
List<Integer> ints = Arrays.asList(1,2,3);
List<? extends Number> nums = ints;
nums.add(null);  // ok
assert nums.toString().equals("[1,2,3,null]");
```

Similarly, you cannot get anything out from a type declared with an `extends` wildcard — but you can extract a value at type `Object`, which is a supertype of every reference type.

```
List<Object> objs = Arrays.<Object>asList(1,"two");
List<? super Integer> ints = objs;
String str = "";
for (Object obj : ints) str += obj.toString();
assert str.equals("1two");
```

You may find it helpful to think of `? extends T` as containing every type in interval bounded by `null` below and `T` above (where `null` is a subtype of every reference type). Similarly, you may think of `? super T` as a containing every type in an interval bounded by `T` below and `Object` above.

2.5 Arrays

It is instructive to compare the treatment of lists and arrays in Java, keeping in mind the Substitution Principle and the Get and Put Principle.

In Java, array subtyping is *covariant*, meaning that type `S[]` is considered to be a subtype of `T[]` whenever `S` is a subtype of `T`. Consider the following code fragment, which allocates an array of integers, assigns it to an array of numbers, and then attempts to assign a float into the array.

```
Integer[] ints = new Integer[] {1,2,3};
Number[] nums = ints;
nums[2] = 3.14;  // array store exception
assert Arrays.toString(ints).equals("[1, 2, 3.14]");  // uh oh!
```

Something is wrong with this program, since it puts a float into an array of integers! Where is the problem? Since `Integer[]` is considered a subtype of `Number[]`, by the Substitution Principle the assignment on the second line must be legal. Instead, the problem is caught on the third line, and it is caught at run-time. When an array is allocated (as on the first line), it is tagged with its reified type, and every time it is assigned (as on the third line), an array store exception is raised if the reified type is not compatible with the assigned value.

In contrast, the subtyping relation for generics is *invariant*, meaning that type `List<S>` is *not* considered to be a subtype of `List<T>` except in the trivial case where `S` and `T` are identical. Here is a code fragment analogous to the one above, with lists replacing arrays.

```
List<Integer> ints = Arrays.asList(1,2,3);
List<Number> nums = ints;  // compile-time error
nums.put(2, 3.14);
assert ints.toString().equals("[1, 2, 3.14]");  // uh oh!
```

Since `List<Integer>` is not considered to be a subtype of `List<Number>`, the problem is detected on the second line, not the third, and it is detected at compile-time, not run-time.

Wildcards reintroduce covariant subtyping for generics, in that type `List<S>` *is* considered to be a subtype of `List<? extends T>`, when `S` is a subtype of `T`. Here is a third variant of the fragment.

```
List<Integer> ints = Arrays.asList(1,2,3);
List<? extends Number> nums = ints;
nums.put(2, 3.14);  // compile-time error
assert ints.toString().equals("[1, 2, 3.14]");  // uh oh!
```

Now it is again the third line which is in error, due to the Get and Put Principle: one cannot put a value into a type declared with an `extends` wildcard. In contrast to arrays, the problem it is detected at compile-time, not run-time.

Wildcards also introduce *contravariant* subtyping for generics, in that type `List<S>` is considered to be a *subtype* of `List<? super T>`, when `S` is a *supertype* of `T` (as opposed to a subtype). Arrays do not support contravariant subtyping. For instance, recall that the method `count` accepted a parameter of type `Collection<? super Integer>` and filled it with integers. There is no equivalent way to do this with an array: the type `Integer[]` would be too specific (it rules out passing in a `Number[]`, even though that should be an acceptable argument to `count`), and the type `Object[]` would be too general (it permits passing in a `String[]`, even though that should not be an acceptable argument to `count`).

Detecting problems at compile-time rather than at run-time brings two advantages, one minor and one major. The minor advantage is that it is more efficient. The system does not need to carry around a description of the element type at run-time, and the system does not need to check against this description every time an assignment is performed. The major advantage is that a common family of errors is detected by the compiler. This improves virtually all aspects of the program's life cycle: coding, debugging, testing, and maintenance are all made easier, quicker, and less expensive.

Apart from the fact that errors are caught earlier, there are many other reasons to prefer collection classes to arrays. Collections are far more flexible than arrays. The only operations supported on arrays are to get or set an element, and the representation is fixed. Collections may support many additional operations, including testing for containment, adding and removing elements, and operations like `containsAll` and `addAll` that act on two collections at once, and one may extract a sublist of a list. Collections may either be lists (where order is significant and elements may be repeated) or sets (where order is not significant and elements may not be repeated), and there are a number of representations available, including arrays, linked lists, trees, and hash tables; there are operations like `addAll` and `containsAll` that combine or compare two collections. Finally, a comparison of the convenience classes `Collections` and `Arrays` shows that the former supports many more general-purpose methods than the latter, including operations to rotate or shuffle a list, to find the maximum of a collection, or to make a collection unmodifiable or synchronized.

Nonetheless, there are a few cases where arrays are to be preferred to collections. Arrays of primitive type are much more efficient since they don't involve boxing; and assigning to such an array need not check for an array store exception, because primitive types don't have subtypes. And despite the check for array store exceptions, even arrays of reference type may be more efficient than collection classes with the current generation of compilers, so you may want to use arrays in crucial inner loops. As always, you should measure performance to justify such a design, especially since future compilers may optimize collection classes specially. Finally, in some cases arrays may be preferable for reasons of compatibility.

To summarize, it is better to detect errors at compile-time rather than run-time, but Java arrays are forced to detect certain errors at run-time by the decision to make array subtyping covariant. Was this a good decision? Before the advent of generics, it was absolutely necessary. For instance, look at the following methods, which are used to sort any array or to fill an array with a given value.

```
public static void sort (Object[] a);
public static void fill (Object[] a, Object val);
```

Thanks to covariance, these methods can be used to sort or fill arrays of any reference type. Without covariance and without generics, there would be no way to declare methods that apply for all types. However, now that we have generics, covariant arrays are no longer necessary. Now we can give the methods the following signatures, directly stating that they work for all types.

```
public static <T> void sort (T[] a);
public static <T> void fill (T[] a, T val);
```

In some sense, covariant arrays are an artefact of the lack of generics in earlier versions of Java. Once you have generics, covariant arrays are probably the wrong design choice, and the only reason for retaining them is backward compatibility.

2.6 Wildcards vs. type parameters

The method `contains` checks whether a collection contains a given object, and its generalization `containsAll` checks whether a collections contains every element of another collection. This section presents two alternate approaches to giving generic signatures for these methods. The first approach uses wildcards, and is the one used in the Java Collections Framework. The second approach uses type parameters, and is often a more appropriate alternative.

Wildcards Here are the types that the methods have in Java 5.

```
interface Collection<E> {
    ...
    public boolean contains (Object o);
    public boolean containsAll (Collection<?> c);
    ...
}
```

The first method does not use generics at all! The second method is our first sight of an important abbreviation, where `Collection<?>` stands for `Collection<? extends Object>`. Extending `Object` is one of the most common uses of wildcards, so it makes sense to provide a short form for writing it.

These methods let us test for membership and containment.

```
Object obj = "one";
List<Object> objs = Arrays.<Object>asList("one", 2, 3.14, 4);
List<Integer> ints = Arrays.asList(2, 4);
assert objs.contains(obj);
assert objs.containsAll(ints);
assert !ints.contains(obj);
assert !ints.containsAll(objs);
```

The given list of objects contains both the string `"one"` and the given list of integers, but the given list of integers does not contain the string `"one"` nor does it contain the given list of objects.

The tests `ints.contains(obj)` and `ints.containsAll(objs)` might seem silly. Of course a list of integers won't contain an arbitrary object, such as the string `"one"`. But it is permitted because sometimes such tests might succeed.

```
Object obj = 1;
List<Object> objs = Arrays.<Object>asList(1, 3);
List<Integer> ints = Arrays.asList(1, 2, 3, 4);
assert ints.contains(obj);
assert ints.containsAll(objs);
```

In this case the object may be contained in the list of integers because it happens to be an integer, and the list of objects may be contained within the list of integers because every object in the list happens to be an integer.

Type parameters One might reasonably have chosen an alternative design for collection, where one can only test for containment for subtypes of the element type.

```
interface MyCollection<E> {  // alternative design
    ...
    public boolean contains (E o);
    public boolean containsAll (Collection<? extends E> c);
    ...
}
```

Say we have a class `MyList` that implements `MyCollection`. Now the tests are legal only one way around.

```
Object obj = "one";
MyList<Object> objs = MyList.<Object>asList("one", 2, 3.14, 4);
MyList<Integer> ints = MyList.asList(2, 4);
assert objs.contains(obj);
assert objs.containsAll(ints)
assert !ints.contains(obj);       // compile-time error
assert !ints.containsAll(objs);   // compile-time error
```

The last two tests are illegal, because the type declarations require that we can only test whether a list contains element of a subtype of that list. So we can check whether a list of objects contains a list of integers, but not the other way around.

It is a matter of taste which of the two styles is better. The first permits more tests, while the second captures more errors at compile time (while also ruling out some sensible tests). The designers of the Java libraries chose the first, more liberal alternative, because someone using the Collection Framework *before* generics might well have written a test like `ints.containsAll(objs)`, and they would like that test to remain valid *after* generics were added to Java. However, when designing a new generic library, like `MyCollection`, when backwards compatibility is less important, then the design that catches more errors at compile time might make more sense.

Arguably, the library designers made the wrong choice. Tests such as `ints.containsAll(objs)` are rather rare, and when they occur they can still be permitted by taking `ints` to have type `List<Object>` rather than type `List<Integer>`. It might have been more sensible to get better error checking in the common case rather to allow more precise typing in an uncommon case.

The exact same design choice applies to other methods that contain `Object` or `Collection<?>` in their signature, such as `remove`, `removeAll`, and `retainAll`.

2.7 Wildcard capture

It is perfectly sensible, and even useful, to instantiate a type variable to the unknown type represented by a wildcard. This operation is called *wildcard capture*.

Consider the method `reverse` in the convenience class `java.util.Collections`, which accepts a list of any type and reverses it. It can be given either of the following two signatures.

```
public static void reverse(List<?> list);
public static void <T> reverse(List<T> list);
```

The wildcard signature is slightly shorter and clearer, and is the one used in the library. But since the signatures are equivalent, you should have the choice of using either one.

If you use the second signature, it is easy to implement the method.

```
public static void <T> reverse(List<T> list) {
    List<T> tmp = new ArrayList<T>(list);
    for (int i = 0; i < list.size(); i++) {
        list.set(i, tmp.get(list.size()-i-1));
    }
}
```

This copies the argument into a temporary list, then writes from the copy back into the original in reverse order.

If you try to use the first signature with a similar method body, it doesn't work.

```
public static void reverse(List<?> list) {
    List<Object> tmp = new ArrayList<Object>(list);
    for (int i = 0; i < list.size(); i++) {
        list.set(i, tmp.get(list.size()-i-1));  // compile-time error
    }
}
```

Now it is not legal to write from the copy back into the original, because we are trying to write from a list of object into a list of unknown type. Replacing `List<Object>` by `List<?>` won't fix the problem, because now we have two lists with (possibly different) unknown element types.

Instead, one can implement the method with the first signature by implementing a private method with the second signature, and calling the second from the first.

```
public static void reverse(List<?> list) { rev(list); }
private static <T> void rev(List<T> list) {
    List<T> tmp = new ArrayList<T>(list);
    for (int i = 0; i < list.size(); i++) {
        list.set(i, tmp.get(list.size()-i-1));
    }
}
```

Here we say that the type variable `T` has *captured* the wildcard. This is a generally useful technique when dealing with wildcards, and worth knowing.

Another reason it is worth knowing about wildcard capture is that it can show up in error messages, even if you don't use the above technique. In general, each occurrence of a wildcard is taken to stand for some unknown type. If the compiler prints an error message containing this type, it is referred to as `capture of ?`. For instance, the incorrect version of `reverse` generates the following error message.

```
Capture.java:6: set(int,capture of ?) in java.util.List<capture of ?>
cannot be applied to (int,java.lang.Object)
            list.set(i, tmp.get(list.size()-i-1));
                ^
```

Hence if you see the quizical phrase `capture of ?` in an error message, it will come from a wildcard type. Even if there are two distinct wildcards, the compiler will print the type associated with each as `capture of ?`. Bounded wildcards generate even more long winded names, such as `capture of ? extends Number`.

2.8 Restrictions on wildcards

Wildcards may not appear at the top-level in class instance creation expressions (`new`), in supertypes (`extends` and `implements`) and or in explicit type parameters in generic method calls.

Instance creation In a class instance creation expression, if the type is a parameterized type, then none of the type parameters may be wildcards. For example, the following are illegal.

```
List<?> list = new ArrayList<?>();  // compile-time error
Map<String,? extends Number> map
    = new HashMap<String, ? extends Number>();  // compile-time error
```

This is usually not a hardship. The Get and Put Principle tells us that if a structure contains a wildcard, then almost always we are only getting values out (if it is an `extends` wildcard) or only putting values in (if it is a `super` wildcard). For a structure to be useful we must do both. Therefore, we typically create a structure at a precise type, and assign to wildcard types in order to read or write the structure. Here's a more typical use of wildcards with lists.

```
List<Number> nums = new ArrayList<Number>();
List<? super nums> sink = nums;
List<? extend nums> source = nums;
for (int i=0; i<10; i++) sink.add(i);
double sum=0; for (double d : source) sum+=d;
```

Here the array creation does not involve a wildcard.

Only top-level parameters in instance creation are prohibited from containing wildcards. Nested wildcards are permitted. Hence the following is legal.

```
List<List<? super Number>> numslist
    = new ArrayList<List<? super Number>>();
numslist.add(Arrays.asList(1,2,3));
numslist.add(Arrays.asList(2.78,3.14));
System.out.println(numslist);
```

The types prohibits us from extracting elements from the inner lists at any type other than `Object`, but that is the type used by the print method, so this code is ok.

One way to remember this restriction is that the relation between wildcards and ordinary types is similar to the relation between interfaces and classes — a wildcards and interfaces are more general, ordinary types and classes more specific, and instance creation requires the more specific information. Consider the following three statements.

```
List<?> list = new ArrayList<Object>();   // ok
List<?> list = new List<Object>()  // compile-time error
List<?> list = new ArrayList<?>()  // compile-time error
```

The first is legal, the second is illegal because an instance creation expression requires a class, not an interface, and the third is illegal because an instance creation expression requires an ordinary type, not a wildcard.

You might wonder why this restriction is necessary. The Java designers had in mind that every wildcard type is a shorthand for some ordinary type, so they believed that ultimately every object should be created at an ordinary type. It is not clear whether in fact this restriction is necessary.

We give a workaround that let's you escape the restriction later, after first describing two similar restrictions.

Supertypes When a class instance is created, it invokes the initializer for its supertype. Hence, any restriction that applies to instance creation must also apply to supertypes. In a class declaration, if the supertype or any superinterface has type parameters, these types must not be wildcards.

For example, this declaration is illegal.

```
class AnyList extends ArrayList<?> {...} // compile-time error
```

And so is this.

```
class AnotherList implements List<?> {...} // compile-time error
```

But, as before, nested wildcards are permitted.

```
class NestedList implements ArrayList<List<? super Number>> {...}  // ok
```

The motivation for this restriction is similar to the previous one, and again it is not clear it is necessary.

Generic method calls If a generic method call includes explicit type parameters, then those type parameters must not be wildcards. For example, say we have the following generic method.

```
class Lists {
    public static <T> List<T> factory() { return new ArrayList<T>(); }
}
```

One may choose for the type parameters to be inferred, or one may pass an explicit type parameter. Both of the following are legal.

```
List<?> list = Lists.factory();
List<?> list = Lists.<Object>factory();
```

If an explicit type parameter is passed, it must not be a wildcard.

```
List<?> list = Lists.<?>factory();  // illegal
```

It is clear how this relates to the previous restrictions, in that if the type parameter was allowed to be a wildcard, then it would be easy to create a new instance at a wildcard type. As before, nested wildcards are permitted.

```
List<List<? super Number>> = Lists.<List<? super Number>>factory();
```

Workaround You are unlikely to want to use wildcards in any of the above situations. However, if a wildcard is essential, you can achieve the same effect using wildcard capture and the fact that nested wildcards are permitted.

Say that we have a generic method with the following signature, where $X_1, \ldots, X_m$ are type variables, and $x_1, \ldots, x_n$ are ordinary variables with types $T_1, \ldots, T_n$.

```
<X₁,...,Xₘ> method(T₁ x₁,...,Tₙ xₙ) {...}
```

Say that we want to invoke the method as follows, where $W_1, \ldots, W_m$ are arbitrary types, perhaps wildcards, and $e_1, \ldots, e_n$ are expressions.

```
this.<W₁,...,Wₘ>method(e₁,...,eₙ);
```

We can do this by creating a dummy type and a second (overloaded) method that accepts a parameter of the dummy type and invokes the first method.

```
class Dummy<X₁,...,Xₘ> {}
<X₁,...,Xₘ> method(Dummy<X₁,...,Xₘ> d, T₁ x₁,...,Tₙ xₙ) {
    return this.<X₁,...,Xₘ>method(x₁,...,xₙ);
}
```

Then we replace the method invocation above by the following.

```
Dummy<W₁,...,Wₘ> d = null;
this.method(d,e₁,...,eₙ);
```

Now type inference guarantees we pass the desired type parameters, and (by wildcard capture) this may include wildcard types.

Similarly, if one wants to create an instance at a wildcard type, one can do that by writing a factory for the object creator. Say we want to write

```
x = new C<W₁,...,Wₘ>(e₁,...,eₙ);
```

Then define the following factory.

```
<X₁,...,Xₘ> factory(Dummy<X₁,...,Xₘ> d, T₁ x₁,...,Tₙ xₙ) {
  return new C<X₁,...,Xₘ>(x₁,...,xₙ);
}
```

And replace the instance creation above by the following.

```
Dummy<W₁,...,Wₘ> d = null;
x = factory(d,e₁,...,eₙ);
```

Again, type inference guarantees this passes the correct type parameters.

The existence of these workarounds suggest that the restrictions on wildcards accomplish little save to increase complication, and should be removed from the language.

Chapter 3

Comparison and bounds

Now that we have the basics, let's look at some more advanced uses of generics. This section describes the interfaces `Comparable<E>` and `Comparator<E>` that are used to support comparison on elements. These interfaces are useful, for instance, if you want to find the maximum element of a collection or sort a list. Along the way we will introduce *bounds* on type variables, an important feature of generics that is particularly useful in combination with the `Comparable<E>` interface.

3.1 Comparable

The interface `Comparable<T>` contains a single method that can be used to compare one object to another.

```
interface Comparable<T> {
  int compareTo(T o);
}
```

The `compareTo` method returns a value that is negative, zero, or positive depending upon whether the argument is less than, equal to, or greater than the given object. The order specified by this interface is called the *natural order* for the object.

Typically, elements in certain classes can be compared only with elements of the same class. For instance, `Integer` implements `Comparable<Integer>`.

```
Integer int0 = 0;
Integer int1 = 1;
assert int0.compareTo(int1) < 0;
```

The comparison returns a negative number, since `0` precedes `1` in numerical order. Similarly `String` implements `Comparable<String>`.

```
String str0 = "zero";
String str1 = "one";
assert str0.compareTo(str1) > 0;
```

This comparison returns a positive number, since `"zero"` follows `"one"` in alphabetic order.

The type parameter to the interface allows nonsensical comparisons to be caught at compile time.

```
Boolean bool0 = false;
Boolean bool1 = true;
assert bool0.compareTo(bool1) > 0;  // compile-time error
```

Here the comparison is illegal, because class `Boolean` does not implement interface `Comparable`. Here is another example.

```
Integer int0 = 0;
String str1 = "one";
assert int0.compareTo(str1) > 0;  // compile-time error
```

One can compare an integer with an integer or a string with a string, but attempting to compare an integer with a string signals a compile-time error.

We use standard idioms for comparison. Instead of `x < y` we write

`x.compareTo(y) < 0`

and instead of `x <= y` we write

`x.compareTo(y) <= 0.`

The contract for the `Comparable<T>` interface specifies three properties. First, equal objects should have compare as the same:

if `x.equals(y)` then `x.compareTo(y) == 0.`

The converse usually holds, but not always; we'll later see an example where using `compareTo` to relate two objects returns zero while using `equals` on the same two objects returns false (see the end of Section 3.3). Second, reversing the order of arguments should reverse the result:

`x.compareTo(y) < 0` if and only if `y.compareTo(x) > 0.`

This generalizes the property for integers: `x < y` if and only if `y < x`. Third, comparison should be transitive:

if `x.compareTo(y) < 0` and `y.compareTo(z) < 0` then `x.compareTo(z) < 0`

This generalizes the property for integers: if `x < y` and `y < z` then $x < z$.

It's worth pointing out a subtlety in the defintion of comparison. Here is the right way to compare two integers.

```
class Integer implements Comparable<Integer> {
    ...
    public int compare (Integer that) {
        return this.value < that.value ? -1 :
               this.value == that.value ? 0 : 1 ;
    }
    ...
}
```

The conditional expression returns -1, 0, or 1 depending on whether the receiver is less than, equal to, or greater than the argument. You might think the following code work instead, since the method is permitted to return any negative integer if the reciever is less than the argument.

```
class Integer implements Comparable<Integer> {
    ...
    public int compareTo (Integer that) {
        // bad implementation -- don't do it this way!
        return this.value - that.value;
    }
    ...
}
```

This may give the wrong answer when there is overflow. For instance, when comparing a large negative value to a large positive value, the difference may be more than the largest value that can be stored in an integer, `Integer.MAX_VALUE`.

3.2 Maximum of a collection

In this section, we show how to use the `Comparable<T>` interface to find the maximum element in a collection. We begin with a simplified version. The actual version found in the Java 5 Collection Framework has a type signature that is a bit more complicated, and later we will see why.

Here is code to find the maximum element in a non-empty collection, from the class `Collections`.

```
public static <T extends Comparable<T>> T max (Collection<T> coll) {
    T candidate = coll.iterator().next();
    for (T elt : coll) {
        if (candidate.compareTo(elt) < 0) candidate = elt;
    }
    return candidate;
}
```

We first saw generic methods that declare new type variables in the signature in Section 1.4. For instance, the method `asList` takes an array of type `E[]` and returns a result of type `List<E>`, and does so for *any* type `E`. Here we have a generic method that declares a *bound* on the type variable. The method `max` takes a collection of type `Collection<T>` and returns a `T`, and it does this for *any* type `T` *such that* `T` is a subtype of `Comparable<T>`.

The highlighted phrase in angle brackets at the beginning of the type signature declares the type variable `T`, and we say that `T` is *bounded* by `Comparable<T>`. As with wildcards, bounds for type variables are always indicated by the keyword `extends`, even when the bound is an interface rather than a class, as is the case here. Unlike wildcards, type variables must always be bounded using `extends`, never `super`.

The method body sets a candidate to the first element in the collection, then compares the candidate with each element in the collection, setting the candidate to the element when the element is larger. We use `iterator().next()` rather than `get(0)` to get the first element, because `get(0)` is not defined on collections other than lists. The method raises a `NoSuchElement` exception when the collection is empty.

When calling the method, `T` may be chosen to be `Integer` (since `Integer` implements `Comparable<Integer>`) or `String` (since `String` implements `Comparable<String>`).

```
List<Integer> ints = Arrays.asList(0,1,2);
assert Collections.max(ints) == 2;

List<String> strs = Arrays.asList("zero","one","two");
assert Collections.max(strs).equals("zero");
```

But we may not choose `T` to be `Boolean` (since `Boolean` does not implement `Comparable` at all).

```
List<Boolean> bools = Arrays.asList(false,true);
assert Collections.max(bools) == true;  // compile-time error
```

As expected, here the call to `max` is illegal.

Here's an efficiency tip. The above implementation used a *foreach* loop to increase brevity and clarity. If efficiency was a pressing concern, one might want to rewrite the method to use an explicit iterator, as follows.

```
public static <T extends Comparable<T>> T max (Collection<T> coll) {
    Iterator<T> it = coll.iterator();
    T candidate = it.next();
    while (it.hasNext()) {
        T elt = it.next();
        if (candidate.compareTo(elt) < 0) candidate = elt;
    }
    return candidate;
}
```

This allocates an iterator once instead of twice, and performs one less comparison.

Signatures for methods should be as general as possible to maximize utility. If one can replace a type parameter by a wildcard then one should do so. we can improve the signature of `max` by replacing

```
<T extends Comparable<T>> T max (Collection<T> coll)
```

with

```
<T extends Comparable<? super T>> T max (Collection<? extends T> coll)
```

Following the Get and Put Principle, we use `extends` with `Collection` because we *get* values of type `T` from the collection, and we use we use **super** with `Comparable` because we *put* value of type `T` into the `compareTo` method. We'll see an example that would not type check if the `super` clause above was omitted in the next section.

If you look at the signature of this method in the Java library, you will see something that looks even worse than the above.

```
<T extends Object & Comparable<? super T>>
        T max (Collection<? extends T> coll)
```

This is there for backward compatibility, as explained at the end of Section 3.6.

3.3 A fruity example

The `Comparable<T>` interface gives fine control over what can and cannot be compared. Say that we have a class `Fruit` with subclasses `Apple` and `Orange`. Depending on how we set things up, we may *prohibit* comparison of apples with oranges or we may *permit* such comparison.

Example 3.1 prohibits comparison of apples with oranges. Here are the three classes it declares.

```
class Fruit {...}
class Apple extends Fruit implements Comparable<Apple> {...}
class Orange extends Fruit implements Comparable<Orange> {...}
```

Each fruit has a name and a size, and two fruits are equal if they have the same name and the same size. Apples are compared by comparing their sizes, and so are oranges. Since `Apple` implements `Comparable<Apple>`, it is clear that one can compare apples with apples, but not with oranges. The test code builds three lists, one of apples, one of oranges, and one containing mixed fruits. We may find the maximum of the first two lists, but attempting to find the maximum of the mixed list signals an error at compile time.

Example 3.2 permits comparison of apples with oranges. Compare the three class declarations below with those given previously.

```
class Fruit implements Comparable<Fruit> {...}
class Apple extends Fruit {...}
class Orange extends Fruit {...}
```

As before, each fruit has a name and a size, and two fruits are equal if they have the same name and the same size. Now any two fruits are compared by ignoring their names and comparing their sizes. Since `Fruit` implements `Comparable<Fruit>`, any two fruits may be compared. Now the test code can find the maximum of all three lists, including the one that mixes apples with oranges.

Recall that at the end of the previous section we extended the type signature of `compareTo` to use `super`.

```
<T extends Comparable<? super T>> T max (Collection<? extends T> coll)
```

The second example shows why this wildcard is needed. If we want to compare two oranges, we take `T` in the above to be `Orange`.

`Orange extends Comparable<? super Orange>`

And this is true because both of the following hold.

`Orange extends Comparable<Fruit>` and `Fruit super Orange`

Without the `super` wildcard, finding the maximum of a list of oranges would be illegal, even though finding the maximum of a list of fruits is ok.

Also, note that this is a case where equality and comparison differ, in that using `compareTo` to relate two objects may return zero, while using `equals` to compare the same two objects may return false. This will happen, for instance, if one compares an apple and an orange of the same size.

Example 3.1 Prohibiting comparison of apples with oranges

```
import java.util.*;
class Fruit {
    protected String name;
    protected int size;
    public Fruit (String name, int size) {
        this.name = name; this.size = size;
    }
    public boolean equals (Object o) {
        if (o instanceof Fruit) {
            Fruit that = (Fruit)o;
            return this.name==that.name && this.size==that.size;
        } else return false;
    }
    protected int compareTo (Fruit that) {
        return this.size < that.size ? - 1 :
               this.size == that.size ? 0 : 1 ;
    }
}
class Orange extends Fruit implements Comparable<Orange> {
    public Orange (int size) { super("Orange", size); }
    public int compareTo (Orange o) { return super.compareTo(o); }
}
class Apple extends Fruit implements Comparable<Apple> {
    public Apple (int size) { super("Apple", size); }
    public int compareTo (Apple a) { return super.compareTo(a); }
}
class Test {
    public static void main (String[] args) {

        Apple a1 = new Apple(1);  Apple a2 = new Apple(2);
        Orange o3 = new Orange(3);  Orange o4 = new Orange(4);

        List<Apple> apples = Arrays.asList(a1,a2);
        assert Collections.max(apples).equals(a2);

        List<Orange> oranges = Arrays.asList(o3,o4);
        assert Collections.max(oranges).equals(o4);

        List<Fruit> mixed = Arrays.<Fruit>asList(a1,o3);
        assert Collections.max(mixed).equals(o3);  // compile-time error
    }
}
```

Example 3.2 Permitting comparison of apples with oranges

```
class Fruit implements Comparable<Fruit> {
    String name;
    int size;
    public Fruit(String name, int size) {
        this.name = name; this.size = size;
    }
    public boolean equals (Object o) {
        if (o instanceof Fruit) {
            Fruit that = (Fruit)o;
            return this.name==that.name && this.size==that.size;
        } else return false;
    }
    public int compareTo(Fruit that) {
        return this.size < that.size ? - 1 :
               this.size == that.size ? 0 : 1 ;
    }
}
class Apple extends Fruit {
    public Apple (int size) { super("Apple", size); }
}
class Orange extends Fruit {
    public Orange (int size) { super("Orange", size); }
}
class Test {
    public static void main (String[] args) {

        Apple a1 = new Apple(1);  Apple a2 = new Apple(2);
        Orange o3 = new Orange(3);  Orange o4 = new Orange(4);

        List<Apple> apples = Arrays.asList(a1,a2);
        assert Collections.max(apples).equals(a2);

        List<Orange> oranges = Arrays.asList(o3,o4);
        assert Collections.max(oranges).equals(o4);

        List<Fruit> mixed = Arrays.<Fruit>asList(a1,o3);
        assert Collections.max(mixed).equals(o3);  // ok
    }
}
```

3.4 Comparator

Sometimes we want to compare objects that do not implement the `Comparable` interface, or to compare objects using a different order than the one specified by that interface. The order specified by the `Comparable` interface is called the *natural order*, so we might regard `Comparator` as providing, so to speak, an unnatural order.

We can specify additional orderings using the `Comparator` interface, which contains a single method.

```
interface Comparator<T> {
    public int compare(T o1, T o2);
}
```

The `compare` method returns a value that is negative, zero, or positive depending upon whether the first object is less than, equal to, or greater than the second object — just like `compareTo`.

For instance, here is a comparator that compares strings such that the shorter string is considered smaller. Only if two strings have the same length are they compared using the natural (alphabetic) order.

```
Comparator<String> sizeOrder
    = new Comparator<String> () {
        public int compare (String s1, String s2) {
            return
                s1.length() < s2.length() ? -1 :
                s1.length() > s2.length() ? 1 :
                s1.compareTo(s2) ;
            }
    };
```

In the natural alphabetic ordering, `"three"` is smaller than `"two"`, while in the size ordering it is greater.

```
assert "two".compareTo("three") > 0;
assert sizeOrder.compare("two","three") < 0;
```

In the Java libraries, one always has a choice between `Comparable` and `Comparator`. For every generic method with a type variable bounded by `Comparable`, there is another generic method with an additional argument of type `Comparator`. For instance, corresponding to

```
<T extends Comparable<? super T>> T max (Collection<? extends T> coll)
```

we also have

```
<T> T max (Collection<? extends T> coll, Comparator<? super T> cmp)
```

The code for the new version of `max` can be seen in Example 3.3. Compared to the old version, the only change is that where before we wrote `candidate.compareTo(elt)` now we write `cmp.compare(candidate,elt)`. Here are examples to find the maximum of the list using the natural order and using a given comparator.

```
Collection<String> strings = Arrays.asList("from","aaa","to","zzz");
assert max(strings).equals("zzz");
assert max(strings,sizeOrder).equals("from");
```

It is easy to define a comparator that provides the natural order.

```
public static <T extends Comparable<? super T>>
        Comparator<T> naturalOrder()
{
    return new Comparator<T> {
        public int compare (T o1, T o2) { return o1.compareTo(o2); }
    }
}
```

Using this, it is easy to define the version of `max` that uses the natural order in terms of the version that uses a given comparator.

```
public static <T extends Comparable<? super T>>
        T max (Collection<T> coll)
{
    return max(coll, Comparators.<T>naturalOrder());
}
```

A type parameter must be explicitly supplied for the invocation of the generic method `naturalOrder`. Java follows a general principle: the type of an expression depends only on the types of its subexpressions, not on the type expected by the context in which it is used. Since the method has no parameters it has no subexpressions that can be used to determine its type, so the type must be given explicitly. The same reasoning applies in general, so an explicit type parameter must always be provided when a generic method has no arguments.

Example 3.3 also gives two overloaded versions of the generic method `reverseOrder`. One takes no arguments and returns a comparator for the reverse of the natural order, and the other takes a comparator and returns a new comparator with the order reversed. Similar methods are provided in `java.util.Collections`.

It is easy to define versions of `min` for the natural order and for a given comparator, in terms of the version of `max` that accepts a comparator using the `reverseOrder` methods.

```
Collection<String> strings = Arrays.asList("from","aaa","to","zzz");
assert min(strings).equals("aaa");
assert min(strings,sizeOrder).equals("to");
```

There are actually two different ways that `min` over the natural order can be defined. Here is the body of the method from the figure, followed by the alternative body.

```
return max(coll, Comparators.<T>reverseOrder());
return min(coll, Comparators.<T>naturalOrder());
```

The first of these is more direct so it is slightly faster. Measurements show a speedup of about 6%.

The Collections Framework does provide two version each of `min` and `max`, with the signatures given here. However, if you examine the source code of the library, you will see that none of the four is defined in terms of any of the others, each is defined directly instead. The more direct version is longer and harder to maintain, but faster. Measurements show a speedup of around 30%. Whether such a speedup is worth the code duplication depends on the situation in which the code is used. Since the Java utilities might well be used in a critical inner loop, the designers of the library were right to prefer speed of execution over economy of expression. But this is not always the case. An improvement of 30% may sound like a lot, but it's insignificant unless the total time of the program is significant and the routine appears in a heavily used inner loop. Don't make your own code needlessly prolix just to eek out a small improvement.

Example 3.3 Comparators

```
class Comparators {
    public static <T> T max (Collection<T> coll, Comparator<T> cmp) {
        T candidate = coll.iterator().next();
        for (T elt : coll) {
            if (cmp.compare(candidate, elt) < 0) { candidate = elt; }
        }
        return candidate;
    }
    public static <T extends Comparable<? super T>>
                             T max (Collection<T> coll) {
        return max(coll, Comparators.<T>naturalOrder());
    }
    public static <T> T min (Collection<T> coll, Comparator<T> cmp) {
        return max(coll, reverseOrder(cmp));
    }
    public static <T extends Comparable<? super T>>
                             T min (Collection<T> coll) {
        return max(coll, Comparators.<T>reverseOrder());
    }
    public static <T extends Comparable<? super T>>
                             Comparator<T> naturalOrder () {
        return new Comparator<T> () {
            public int compare (T o1, T o2) { return o1.compareTo(o2); }
        };
    }
    public static <T> Comparator<T> reverseOrder (final Comparator<T> cmp) {
        return new Comparator<T> () {
            public int compare (T o1, T o2) { return cmp.compare(o2,o1); }
        };
    }
    public static <T extends Comparable<? super T>>
                             Comparator<T> reverseOrder () {
        return new Comparator<T> () {
            public int compare (T o1, T o2) { return o2.compareTo(o1); }
        };
    }
    public static void main (String[] args) {
        Comparator<String> sizeOrder = new Comparator<String> () {
            public int compare (String s1, String s2) {
                return
                    s1.length() < s2.length() ? -1 :
                    s1.length() > s2.length() ? 1 :
                    s1.compareTo(s2) ;
                }
        };
        Collection<String> strings = Arrays.asList("from","aaa","to","zzz");
        assert max(strings).equals("zzz");
        assert min(strings).equals("aaa");
        assert max(strings,sizeOrder).equals("from");
        assert min(strings,sizeOrder).equals("to");
    }
}
```

3.5 Enumerated types

Java 5 includes support for enumerated types. Here is a simple example.

```
enum Season { WINTER, SPRING, SUMMER, FALL }
```

Each enumerated type declaration can be expanded into a corresponding class in a stylized way. The corresponding class is designed so that it has exactly one instance for each of the enumerated constants, bound to a suitable static final variable. For example, the `enum` declaration above expands into a class called `Season`. There exist exactly four instances of this class, bound to four static final variables with the names `WINTER`, `SPRING`, `SUMMER`, and `FALL`.

Each class that corresponds to an enumerated type is a subclass of the type `java.lang.Enum`. It's definition in the Java documentation begins like this:

```
public abstract class Enum<E extends Enum<E>>
```

Most people find this pretty frightening at first sight — we certainly did! It's easier to understand this after seeing a worked example, let's go through and see how it works.

Example 3.4 shows the base class `Enum` and Example 3.5 shows the class `Season` that corresponds to the enumerated type declaration above. We have followed the actual Java code, but simplified a few points. Here is the beginning of the declaration for the base class.

```
public abstract class Enum<E extends Enum<E>> implements Comparable<E>
```

And here is the beginning of the declaration for the class corresponding to the enumeration.

```
class Season extends Enum<Season>
```

Matching things up, we can begin to see how this works. The type variable `E` stands for the subclass of `Enum` that implements a particular enumerated type, such as `Season`. The constraint `E extends Enum<E>` in the declaration of class `Enum` is indeed satisfied in the case that `E` is `Season`, since the declaration of class `Season` tells us that `Season extends Enum<Season>`. Furthermore, since `Enum<E> implements Comparable<E>`, it follows that `Season implements Comparable<Season>`. So we are allowed to compare two values of type `Season` with each other, but we cannot compare a value of type `Season` with a value of any other type.

Without the type variable, the declaration of the base class would begin like this.

```
class Enum implements Comparable<Enum>
```

And the declaration for the class corresponding to the enumeration would begin like this.

```
class Season extends Enum
```

This is simpler, but it is *too* simple. Now we have that `Season` implements `Comparable<Enum>` rather than `Comparable<Season>`, which would mean that we could compare a value of type `Season` with a value of *any* enumerated type. That is certainly *not* what we want!

The rest of the definition is a straightforward application of the *typesafe enum* pattern described by Josh Bloch in *Effective Java*, which in turn is an instance of the *singleton* pattern described by Gamma, Helm, Johnson, and Vlissides in *Design Patterns*.

The base class `Enum` defines two fields, a string `name` and an integer `ordinal`, that are possessed by every instance of an enumerated type; the fields are final because once they are initialized their value never changes. The constructor for the class is protected, to ensure it is only used within subclasses of this class. Each enumeration class makes the constructor private, to ensure that it is only used to create the enumerated constants. For instance, class `Season` has a private constructor, that is invoked exactly four times, to initialize the final variables `WINTER`, `SPRING`, `SUMMER`, and `FALL`.

The base class defines accessor methods for the two fields `name` and `ordinal`. Method `toString` returns the name, and method `compareTo` just returns the difference of the ordinals for the two enumerated values. (Unlike the definition of `Integer` in Section 3.1, this is safe because there is no possibility of overflow.) Hence, constants have the same order as their ordinals, for example `WINTER` precedes `SUMMER`. There are also two static methods in every class that corresponds to an enumerated type. Method `values` returns an array of all the constants of the type, and method `getValue` takes a string and returns the corresponding constant.

3.6 Multiple bounds

We have seen many examples where a type variable or wildcard is bounded by a single class or interface. In rare situations, it may be desirable to have multiple bounds, and we show how to do so here.

To demonstrate, we will use three interfaces from the Java library. Interface `Readable` has a method `read` to read into a buffer from a source, `Appendable` has a method `append` to copy from a buffer into a target, and `Closeable` has a method `close` to close a source or target. Possible sources and targets include files, buffers, streams, and so on.

For maximum flexibility, we might want to write a copy method that takes any source implementing both `Readable` and `Closeable` and any target implementing both `Appendable` and `Closeable`.

```
public static <S extends Readable & Closeable,
               T extends Appendable & Closeable>
        void copy (S src, T trg, int size)
        throws IOException
{
    CharBuffer buf = CharBuffer.allocate(size);
    int i = src.read(buf);
    while (i >= 0) {
        buf.flip();
        trg.append(buf);
        buf.clear();
        i = src.read(buf);
    }
    src.close();
    trg.close();
}
```

This method repeatedly reads from the source into a buffer and appends from the buffer into a target. When the source is empty, it closes both the source and target. The first line specifies that `S` ranges over any type that implements both `Readable` and `Closeable`, and that `T` is ranges over any type that implements `Appendable` and `Closeable`. When multiple bounds on a type variable appear, they are separated by ampersands. (One cannot use a comma, since that is already used to separate declarations of type variables.)

Example 3.4 Base class for enumerated types

```
public abstract class Enum<E extends Enum<E>> implements Comparable<E> {
    private final String name;
    private final int ordinal;
    protected Enum(String name, int ordinal) {
        this.name = name; this.ordinal = ordinal;
    }
    public final String name() { return name; }
    public final int ordinal() { return ordinal; }
    public String toString() { return name; }
    public final int compareTo(E o) {
        return ordinal - o.ordinal;
    }
}
```

Example 3.5 Class corresponding to an enumerated type

```
// corresponds to
// enum Season { WINTER, SPRING, SUMMER, FALL }
final class Season extends Enum<Season> {
    private Season(String name, int ordinal) { super(name,ordinal); }
    public static final Season WINTER = new Season("WINTER",0);
    public static final Season SPRING = new Season("SPRING",1);
    public static final Season SUMMER = new Season("SUMMER",2);
    public static final Season FALL   = new Season("FALL",3);
    private static final Season[] VALUES = { WINTER, SPRING, SUMMER, FALL };
    public static Season[] values() { return VALUES; }
    public static Season valueOf(String name) {
        for (T e : VALUES) if (e.name().equals(name)) return e;
        throw new IllegalArgumentException();
    }
}
```

For example, this method may be called with two files as source and target, or with the same two files wrapped in buffers as source and target.

```
int size = 32;
FileReader r = new FileReader("file.in");
FileWriter w = new FileWriter("file.out");
copy(r,w,size);
BufferedReader br = new BufferedReader(new FileReader("file.in"));
BufferedWriter bw = new BufferedWriter(new FileWriter("file.out"));
copy(br,bw,size);
```

Other possible sources include `FilterReader` or `PipedReader` or `StringReader`, and other possible targets include `FilterWriter`, `PipedWriter`, and `PrintStream`. But one could not use `StringBuffer` as a target, since it implements `Appendable` but not `Closeable`.

If you are picky, you may have spotted that all classes that implement both `Readable` and `Closeable` are subclasses of `Reader`, and almost all classes that implement `Appendable` and `Closeable` are subclasses of `Writer`. So you might wonder why we don't simplify the method signature like this.

```
public static void copy (Reader src, Writer trg, int size)
```

This will indeed admit most of the same classes, but not quite all. For instance, `PrintStream` implements `Appendable` and `Closeable` but is not a subclass of `Writer`. Furthermore, you can't rule out the possibility that some client of your code might have their own custom class that, say, implements `Readable` and `Closeable` but is not a subclass of `Reader`.

When multiple bounds appear, it is the first bound that is used for erasure. We saw a use of this in the section on maximum, above.

```
public static <T extends Object & Comparable<? super T>>
    T max(Collection<? extends T> coll)
```

Without the highlighted text the erased type signature for `max` would have `Comparable` as the return type, whereas in legacy libraries the return type is `Object`.

3.7 Bridges

As we have mentioned, generics are implemented by erasure: when you write code with generics, it compiles in almost exactly the same way as the code you would have written without generics. In the case of a parameterized interface such as `Comparable<T>`, this may cause additional methods to be inserted by the compiler; these additional methods are called *bridges*.

Example 3.6 shows the interface `Comparable` and a simplified version of the class `Integer` in Java before the introduction of generics. In the non-generic interface, the `compareTo` method takes an argument of type `Object`. In the non-generic class there are two `compareTo` methods. The first is the naive method you might expect, to compare an integer with another integer. The second compares an integer with an arbitrary object: it casts the object to an integer and calls the first method. The second method is necessary in order to overrride the `compareTo` method in the `Comparable` interface, because overriding occurs only when the method signatures are identical. This second method is called a *bridge*.

Example 3.7 shows what happens when the `Comparable` interface and the `Integer` class are generified. In the generic interface, the `compareTo` method takes an argument of type `T`. In the generic class there is a single `compareTo` method that takes an argument of type `Integer`. The bridge method is generated automatically by the compiler. Indeed, the compiled version of the code for both figures is essentially identical.

You can see the bridge if you apply reflection. Here is code to print all methods with the name `compareTo` in the class `Integer`, using `toGenericString` to print the generic signature of a method (see Section 6.5).

```
for (Method m : Integer.class.getMethods())
    if (m.getName().equals("compareTo"))
        System.out.println(m.toGenericString());
```

Running this code on the generic version of the `Integer` class produces the following output.

```
public int Integer.compareTo(Integer)
public bridge int Integer.compareTo(java.lang.Object)
```

This indeed contains two methods, both the declared method that takes an argument of type `Integer` and the bridge method that takes an argument of type `Object`. (The current implementation of Java prints `volatile` instead of `bridge`, because the bit used in Java byte code to indicate bridge methods is also used to indicate volatile fields; this bug will be fixed in the next release.)

Bridges can play an important role when converting legacy code to use generics, see Section 17.4.

3.8 Static members

Because generics are compiled by erasure, at run time the classes `List<Integer>`, `List<String>`, and `List<List<String>>` are all implemented by a single class, namely `List`. You can see this using reflection.

```
List<Integer> ints = Arrays.asList(1,2,3);
List<String> strings = Arrays.asList("one","two");
assert ints.getClass() == strings.getClass();
```

Here the class associated with a list of integers at run-time is the same as the class associated with a list of strings.

One consequence is that static members of a generic class are shared across all instantiations of that class, including instantiations at different types. Static members of a class cannot refer to the type parameter of a generic class, and when accessing a static member the class name should not be parameterized.

For example, here is a class `Cell<T>`, where each cell has an integer identifier and a value of type `T`. A static field `count` is used to allocate a distinct identifier to each cell.

```
class Cell<T> {
    private final int id;
    private final T value;
    private static int count = 0;
    public Cell(T value) { this.value=value; id=count++; }
    public T getValue() { return value; }
    public int getId() { return id; }
    public static int getCount() { return count; }
}
```

Example 3.6 Legacy code for comparable integers.

```
interface Comparable {
    public int compareTo(Object o);
}
class Integer implements Comparable {
    private final int value;
    public Integer(int value) { this.value = value; }
    public int compareTo(Object o) {
        return compareTo((Integer)o);
    }
    public int compareTo(Integer i) {
        return (value < i.value) ? -1 : (value == i.value) ? 0 : 1;
    }
}
```

Example 3.7 Generic code for comparable integers.

```
interface Comparable<T> {
    public int compareTo(T o);
}
class Integer implements Comparable<Integer> {
    private final int value;
    public Integer(int value) { this.value = value; }
    public int compareTo(Integer i) {
        return (value < i.value) ? -1 : (value == i.value) ? 0 : 1;
    }
}
```

Here is code that allocates a cell containing a string and a cell containing an integer, which are allocated the identifiers `0` and `1` respectively.

```
Cell<String> a = new Cell<String>("one");
Cell<Integer> b = new Cell<Integer>(2);
assert a.getId()==0 && b.getId()==1 && Cell.getCount()==2;
```

Because static members are shared across all instantiations of a class, the same count is incremented when allocating either a string or integer cell.

Because static members are independent of any type parameters, we are not permitted to follow the class name with type parameters when accessing a static member.

```
Cell.getCount();             // ok
Cell<Integer>.getCount();    // compile-time error
Cell<?>.getCount();          // compile-time error
```

Of the three calls above, only the first is legal, the other two generate compile-time errors.

For the same reason, one may not refer to a type parameter anywhere within a static member. Here is a second version of cell, that attempts to use a static variable to keep a list of all values stored in any cell.

```
class Cell2<T> {
    private final T value;
    private static List<T> values = new ArrayList<T>(); // illegal
    public Cell(T value) { this.value=value; values.add(value); }
    public T getValue() { return value; }
    public static List<T> getValues() { return values; } // illegal
}
```

Since the class may be instantiated at many different types, it makes no sense to refer to `T` in the declaration of the static field `values` or the static method `getValues()`, and these lines are reported as errors at compile-time. If we want a list of all values kept in cells, then we need to use a list of objects, as in the following variant.

```
class Cell2<T> {
    private final T value;
    private static List<Object> values = new ArrayList<Object>();  // ok
    public Cell(T value) { this.value=value; values.add(value); }
    public T getValue() { return value; }
    public static List<Object> getValues() { return values; }  // ok
}
```

This code compiles and runs with no difficulty.

```
Cell2<String> a = new Cell2<String>("one");
Cell2<Integer> b = new Cell2<Integer>(2);
assert Cell2.getValues().toString().equals("[one, 2]");
```

3.9 Nested classes

Java permits nesting one class inside another. If the outer class has type parameters and the inner class is not static, then type parameters of the outer class are visible within the inner class.

Example 3.8 shows a class implementing collections as a singly-linked list. The class extends `java.util.AbstractCollection`, so it only needs to define the methods `size`, `add`, and `iterator`. The class contains an inner class, `Node`, for the list nodes, and an anonymous inner class implementing `Iterator<E>`. The type parameter `E` is in scope within both of these classes.

For contrast, Example 3.9 shows a similar implementation, but this time the nested `Node` class is static, and so the type parameter `E` is *not* in scope for this class. Instead, the nested class is declared with its own type parameter, also called `E` for convenience. Where the previous version referred to `Node` the new version refers to `Node<E>`. The anonymous iterator class in the preceding example has also been replaced by a nested static class, again with its own type parameter.

If the node classes had been made public rather than private, one would refer to the node class in the first example as `LinkedCollection<E>.Node`, while one would refer to the node class in the second example as `LinkedCollection.Node<E>`.

3.10 How erasure works

The erasure of a type is defined as follows: drop all type parameters from parameterized types, and replace any type variable by the erasure of its bound, or by `Object` if it has no bound, or by the erasure of the leftmost bound if it has multiple bounds. Here are some examples:

- The erasure of `List<Integer>`, `List<String>`, and `List<List<String>>` is `List`.
- The erasure of `List<Integer>[]` is `List[]`.
- The erasure of `List` is itself, similarly for any raw type.
- The erasure of `int` is itself, similarly for any primitive type.
- The erasure of `Integer` is itself, similarly for any type without type parameters.
- The erasure of `T` in the definition of `asList` (see Section 1.4) is `Object`, because `T` has no bound.
- The erasure of `T` in the definition of `max` (see Section 3.2) is `Comparable`, because `T` has bound `Comparable<? super T>`.
- The erasure of `T` in the later definition of `max` (see Section 3.6) is `Object`, because `T` has bound `Object & Comparable<T>` so we take the erasure of the leftmost bound.
- The erasure of `LinkedCollection<E>.Node` or `LinkedCollection.Node<E>` (see Section 3.9) is `LinkedCollection.Node`.

In Java, two distinct methods cannot have the same signature. Since generics are implemented by erasure, it also follows that two distinct methods cannot have signatures with the same erasure. A class cannot overload two methods whose signature has the same erasure, and a class cannot implement two interfaces that have the same erasure.

For example, here is a class with two convenience methods. One adds together every number in a list of numbers, the other concatenates together every string in a list of strings.

Example 3.8 Type parameters *are* in scope for nested, *non-static* classes.

```
import java.util.*;
public class LinkedCollection<E> extends AbstractCollection<E> {
    private class Node {
        private E head;
        private Node tail = null;
        private Node (E elt) { head = elt; }
    }
    private Node first = new Node(null);
    private Node last = first;
    private int size = 0;
    public LinkedCollection () {}
    public LinkedCollection (Collection<? extends E> c) { addAll(c); }
    public int size () { return size; }
    public boolean add (E elt) {
        last.tail = new Node(elt); last = last.tail; size++;
        return true;
    }
    public Iterator<E> iterator() {
        return new Iterator<E>() {
            private Node current = first;
            public boolean hasNext () {
                return current.tail != null;
            }
            public E next () {
                if (current.tail != null) {
                    current = current.tail;
                    return current.head;
                } else throw new NoSuchElementException();
            }
            public void remove () {
                throw new UnsupportedOperationException();
            }
        };
    }
}
```

Example 3.9 Type parameters are *not* in scope for nested, *static* classes.

```
import java.util.*;
class LinkedCollection<E> extends AbstractCollection<E> {
    private static class Node<E> {
        private E head;
        private Node<E> tail = null;
        protected Node (E elt) { head = elt; count++; }
    }
    private Node<E> first = new Node<E>(null);
    private Node<E> last = first;
    private int size = 0;
    public LinkedCollection () {}
    public LinkedCollection (Collection<? extends E> c) { addAll(c); }
    public int size () { return size; }
    public boolean add (E elt) {
        last.tail = new Node<E>(elt); last = last.tail; size++;
        return true;
    }
    private static class LinkedIterator<E> implements Iterator<E> {
        private Node<E> current;
        public LinkedIterator (Node<E> first) { current = first; }
        public boolean hasNext () {
            return current.tail != null;
        }
        public E next () {
            if (current.tail != null) {
                current = current.tail;
                return current.head;
            } else throw new NoSuchElementException();
        }
        public void remove () {
            throw new UnsupportedOperationException();
        }
    }
    public Iterator<E> iterator () {
        return new LinkedIterator<E>(first);
    }
}
```

```
class Overloaded {
    // compile-time error, cannot overload two methods with same erasure
    public static double sum(List<? extend Number> nums) {
        double sum = 0;
        for (Number n : nums) sum += n.doubleValue();
        return sum;
    }
    public static String sum(List<String> strings) {
        int sum = "";
        for (String s : strings) sum += s;
        return sum;
    }
}
```

It is not possible to give both methods the same name and try to distinguish them by overloading, because after erasure it is impossible to distinguish one argument list from the other.

For another example, here is a bad version of the integer class, that tries to make it possible to compare an integer with either an integer or a long.

```
class Integer implements Comparable<Integer>, Comparable<Long> {
    // compile-time error, cannot implement two interfaces with same erasure
    private final int value;
    public Integer(int value) { this.value = value; }
    public int compareTo(Integer i) {
        return (value < i.value) ? -1 : (value == i.value) ? 0 : 1;
    }
    public int compareTo(Long l) {
        return (value < l.value) ? -1 : (value == l.value) ? 0 : 1;
    }
}
```

If this was supported, it would in general require a complex and confusing definition of bridge methods (see Section 3.7), by far the simplest and easiest to understand option is to ban this case.

Chapter 4

Evolution not revolution

One motto underpinning the design of generics for Java is *evolution not revolution*. It must be possible to migrate a large, existing body of code to use generics gradually (evolution) without requiring a radical all-at-once change (revolution). An important consequence is that old code compiles against the new Java libraries, avoiding the unfortunate situation where half your code needs old libraries and half your code needs new libraries.

This section shows how to add generics by considering a small example, a library for stacks that extends the Collections Framework, together with an associated client. We begin with the legacy stack library and client (written for Java 1.4). We then present the corresponding generic library and client (written for Java 5), which provides an opportunity to see how to write a generic class. Our example code is small, so it is easy to update to generics all in one go, but in practice the library and client may evolve separately. This is aided by *raw types*, which are the legacy counterpart of parameterized types.

The parts of the program may evolve in either order. One may have a generic library with a legacy client; this is the common case for anyone that uses the Collections Framework in Java 5 with legacy code. Or one may have a legacy library with a generic client; this requires that one provide generic signatures for the library without rewriting the entire library. We see two ways to do this. One is to combine generic signatures with legacy bodies, the other is to combine generic stub files with legacy class files. The latter is particularly useful when class files are available but source is not.

In practice, the library and client may involve many interfaces and classes, and there may not even be a clear distinction between library and client. But the same principles discussed here still apply, and may be used to evolve any part of a program independently of any other.

4.1 Legacy library with legacy client

We begin with a simple library of stacks and an associated client, as presented in Example 4.1. This is *legacy* code, written for Java 1.4 and its version of the Collections Framework. Like the Collections Framework, we structure the library as an interface `Stack` (which extends `List`), an implementation class `ArrayStack` (which extends `ArrayList`), and a utility class `Stacks` providing operations on stacks (analogous to `Collections`). The interface `Stack` provides just three methods, `empty`, `push`, and `pop`. The implementation class `ArrayStack` provides a single constructor with no arguments, implemented in terms of the superclass, and implements the methods `empty`, `push`, and `pop` using methods `size`, `add`, and `remove` inherited from the superclass. The utility class provides just one method, `reverse`, which repeatedly pops from one stack and pushes into another.

The client allocates a stack, pushes a few integers onto it, pops an integer off, and then reverses the remainder into a fresh stack. Since this is Java 1.4, integers must be explicitly boxed when passed to `push`, and explicitly unboxed when returned by `pop`. The `toString` method used to examine the stack is inherited by `ArrayStack` from its superclass `ArrayList`.

4.2 Generic library and generic client

Next we update the library and client to use generics, as presented in Example 4.2. This is *generic* code, written for Java 5 and its version of the Collections Framework. The major change is that the interface takes a type parameter, becoming `Stack<E>` (extending `List<E>`), and so does the implementing class, becoming `ArrayStack<E>` (extending `ArrayList<E>`). The type `Object` in the signatures and bodies of `push` and `pop` is replaced by the type parameter `E`. As this is our first look at a definition of a parameterized interface and class, it is worth noting that a type parameter is added just as one might expect to the occurrence of `Stack<E>` and `ArrayStack<E>` in the first line of the declaration, but is *not* added to the constructor `ArrayList` as it is already available from the enclosing declaration. The utility class `Stacks` does not take a parameter, but its method `reverse` becomes a generic method with argument and result of type `Stack<E>`. The updates to the client are straightforward.

In short, the conversion process is straightforward, requiring adding a few type parameters and replacing occurrences of `Object` by the appropriate type parameter. All differences between the legacy and generic versions can be spotted by comparing the highlighted portions of the two figures. The implementation of generics is designed so that the two versions generate essentially equivalent class files. Some auxiliary information about the types may differ, but the actual byte codes to be executed will be identical. Hence, executing the legacy and generic versions yields the same results. The fact that legacy and generic source yield the identical class files eases the process of evolution, as we next discuss.

4.3 Generic library with legacy client

Now let's consider the case where the library is updated to generics while the client remains in its legacy version. This may occur because there is not time to convert everything all at once, or because the library and client are controlled by different organizations. This corresponds to the most important case of backward compatibility, where the generic Collections Framework of Java 5 must still work with legacy clients written against the Collections Framework in Java 1.4.

In order to support evolution, whenever a parameterized type is defined, Java also recognizes the corresponding unparameterized version of the type, called a *raw type*. For instance, the parameterized type `Stack<E>` corresponds to the raw type `Stack`, and the parameterized type `ArrayStack<E>` corresponds to the raw type `ArrayStack`. Java permits a raw type to be passed where the corresponding parameterized type is expected, or vice versa, but it generates an *unchecked* warning in such circumstances.

To be specific, consider compiling the generic source for `Stack<E>`, `ArrayStack<E>` and `Stacks` from Example 4.2 (say, in directory `g`) with the legacy source for `Client` from Example 4.1 (say, in directory `l`). Sun's Java 5 compiler yields the following message.

Example 4.1 Legacy library and legacy client

```
l/Stack.java:
    interface Stack extends java.util.List {
        public boolean empty();
        public void push(Object elt);
        public Object pop();
    }

l/ArrayStack.java:
    class ArrayStack extends java.util.ArrayList implements Stack {
        public ArrayStack() { super(); }
        public boolean empty() { return size()==0; }
        public void push(Object elt) { add(elt); }
        public Object pop() {
            Object elt = get(size()-1);
            remove(size()-1);
            return elt;
        }
    }

l/Stacks.java:
    class Stacks {
        public static Stack reverse(Stack in) {
            Stack out = new ArrayStack();
            while (!in.empty()) {
                Object elt = in.pop();
                out.push(elt);
            }
            return out;
        }
    }

l/Client.java:
    class Client {
        public static void main (String[]  args) {
            Stack stack = new ArrayStack();
            for (int i = 0; i<4; i++) stack.push(new Integer(i));
            assert stack.toString().equals("[0, 1, 2, 3]");
            int top = ((Integer)stack.pop()).intValue();
            assert top==3 && stack.toString().equals("[0, 1, 2]");
            Stack reverse = Stacks.reverse(stack);
            assert stack.empty();
            assert reverse.toString().equals("[2, 1, 0]");
        }
    }
```

Example 4.2 Generic library with generic client

```
g/Stack.java:
    interface Stack<E> extends java.util.List<E> {
        public boolean empty();
        public void push(E elt);
        public E pop();
    }

g/ArrayStack.java:
    class ArrayStack<E> extends java.util.ArrayList<E> implements Stack<E> {
        public ArrayStack() { super(); }
        public boolean empty() { return size()==0; }
        public void push(E elt) { add(elt); }
        public E pop() {
            E elt = get(size()-1);
            remove(size()-1);
            return elt;
        }
    }

g/Stacks.java:
    class Stacks {
        public static <E> Stack<E> reverse(Stack<E> in) {
            Stack<E> out = new ArrayStack<E>();
            while (!in.empty()) {
                E elt = in.pop();
                out.push(elt);
            }
            return out;
        }
    }

g/Client.java:
    class Client {
        public static void main (String[] args) {
            Stack<Integer> stack = new ArrayStack<Integer>();
            for (int i = 0; i<4; i++) stack.push(i);
            assert stack.toString().equals("[0, 1, 2, 3]");
            int top = stack.pop();
            assert top==3 && stack.toString().equals("[0, 1, 2]");
            Stack<Integer> reverse = Stacks.reverse(stack);
            assert stack.empty();
            assert reverse.toString().equals("[2, 1, 0]");
        }
    }
```

```
% javac g/Stack.java g/ArrayStack.java g/Stacks.java l/Client.java
Note: Stack.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

The unchecked warning indicates that the compiler cannot offer the same safety guarantees that are possible when generics are used uniformly throughout. However, when the generic code was generated by updating legacy code, we know that equivalent class files are produced from both, and hence (despite the unchecked warning) running a legacy client with the generic library will yield the same result as running the legacy client with the legacy library. Here we assume that the only change in updating the library was to introduce generics, and that no change to the behaviour was introduced, either on purpose or by mistake.

If we follow the suggestion above and re-run the compiler with the appropriate switch enabled, we get more details.

```
% javac -Xlint:unchecked g/Stack.java g/ArrayStack.java \
%    g/Stacks.java l/Client.java
l/Client.java:6: warning: [unchecked] unchecked call
to push(E) as a member of the raw type Stack
        for (int i = 0; i<4; i++) stack.push(new Integer(i));
                                            ^
l/Client.java:10: warning: [unchecked] unchecked conversion
found   : Stack
required: Stack<E>
        Stack reverse = Stacks.reverse(stack);
                                       ^
l/Client.java:10: warning: [unchecked] unchecked method invocation:
<E>reverse(Stack<E>) in Stacks is applied to (Stack)
        Stack reverse = Stacks.reverse(stack);
                                      ^
3 warnings
```

Not every use of a raw type gives rise to a warning. Putting a value into a raw type issues a warning (hence the warning for the invocation of `push`) but getting a value from a raw type is safe (hence no warning for the invocation of `pop`); this works for reasons similar to the Get and Put Principle. Passing a raw type where a parameterized type is expected issues a warning (hence the warning when passing an argument to `reverse`) but passing a parameterized type where a raw type is expected is safe (hence no warning for getting the result from `reverse`); this works for reasons similar to the Substitution Principle.

The most important case of using generic libraries with legacy clients is that of using the Java 5 Collections Framework with legacy code written for the Java 1.4 Collections Framework. In particular, applying the Java 5 compiler to the legacy code in Example 4.1 also issues unchecked warnings, because of the uses of generic versions of `List` from the legacy version of `ArrayStack`. Here is what happens when we compile legacy versions of all the files.

```
% javac -Xlint:unchecked l/Stack.java l/ArrayStack.java \
%    l/Stacks.java l/Client.java
l/ArrayStack.java:5: warning: [unchecked] unchecked call to add(E)
as a member of the raw type ArrayStack
    public void push(Object elt)  add(elt);
                                      ^
1 warning
```

Here the warning for the use of generic method `add` in legacy method `push` is issued for similar reasons to the previous warning for use of generic method `push` from the legacy client.

It is never a good idea to cry wolf. In the case of pure legacy code, such warnings can be turned off by using the `-source 1.4` switch.

```
% javac -source 1.4 l/Stack.java l/ArrayStack.java \
%    l/Stacks.java l/Client.java
```

This compiles the legacy code and issues no warnings or errors. This method of turning off warnings is only applicable to true legacy code, with no 1.5 features, generic or otherwise. One can also turn off unchecked warnings by using annotations, as described below, and this works even with 1.5 features.

4.4 Legacy library with generic client

Usually it makes sense to update the library before the client, but there may be cases when you wish to do it the other way around. For instance, you may be responsible for maintaining the client but not the library; or the library may be large, so you want to update it gradually rather than all-at-once; or you may have class files for the library, but no source.

In such cases, it makes sense to update the library to use parameterized types in its method signatures, but not to change the method bodies. There are two ways to do this. One involves making minimal alterations to the source, the other involves creating stub files.

4.5 Evolving a library — minimal changes

The minimal changes technique is shown in Example 4.3. Here the source of the library has been edited, but only to change method signatures, not method bodies. The exact changes required are highlighed in boldface.

To be precise, the changes required are (a) adding type parameters to interface or class declarations as appropriate (for interface `Stack<E>` and class `ArrayStack<E>`), (b) adding type parameters to any newly parameterized interface or class in an extends or implements clause (for `Stack<E>` in the implements clause of `ArrayStack<E>`), (c) adding type parameters to each method signature as appropriate (for `push`, and `pop` in `Stack<E>` and `ArrayStack<E>`, and for `reverse` in `Stacks`), (d) adding a cast to any return statement, when the return type is a type parameter (for `pop` in `ArrayStack<E>`, where the return type is `E` — without this cast one will get an error rather than an unchecked warning), and (e) optionally adding annotations to suppress unchecked warnings (for classes `ArrayStack<E>` and `Stacks`).

It is worth noting a few changes that we do *not* need to make. In method bodies, we can leave occurrences of `Object` as they stand, and we do not need to add type parameters to any occurrences of raw types (see the first line of `pop` in `ArrayStack`, and of `reverse` in `Stacks`). Also, we only need to add a cast to a return clause when the return type is a type parameter (as in `pop`) but not when the return type is a parameterized type (as in `reverse`).

Finally, if we intend only to call generic versions of the new methods `empty`, `push`, and `pop`, but not of the library methods `size`, `add`, `get`, and `toString`, then we do not need to parameterize occurrences of other library classes (such as the occurrences of `List` in the extends clause of `Stack<E>` or of `ArrayList` in the extends clause of `ArrayStack<E>`). These changes are not required because the types involved are raw types, and so already yield unchecked warnings rather than errors. We might want to make them anyway, but it is helpful to know that they are not required when a large body of code needs to be changed and we want to keep the work to a minimum. In practice, it is usually better to genericize superinterfaces and superclasses if generic versions exist, since then you cna call generic versions of the methods inherited from them.

With these changes, the library will compile successfully, although it will issue a number of unchecked warnings. To indicate that we expect unchecked warnings when compiling the library classes, these have been annotated to suppress such warnings.

```
@SuppressWarnings("unchecked");
```

This prevents the compiler from crying wolf — we've told it not to issue unchecked warnings that we expect, so it will be easy to spot any that we *don't* expect. In particular, once we've updated the library, we should not see any unchecked warnings from the client. Note well that we've suppressed warnings on the library classes, but *not* on the client! (The suppress warnings annotation does not work in early versions of Java 5.)

There is no way to get rid of the unchecked warnings generated by compiling the library, short of updating the entire library source to use generics. This is entirely reasonable, as unless the entire source is updated there is no way the compiler can check that the declared generic types are correct. Indeed, unchecked warnings are warnings — rather than errors — largely because they support the use of this technique. It is best to use this technique only if you are sure that the generic signatures are in fact correct. The best practice is to use this technique only as an intermediate step in evolving code to use generics throughout.

4.6 Evolving a library — stubs

The stubs technique is shown in Example 4.4. Here we write stubs with generic signatures but no bodies. We compile the generic client against the generic signatures, but run the code against the legacy class files. This technique is appropriate when the source is not released, or when others are responsible for maintaining the source.

To be precise, we introduce the same modifications to interface and class declarations and method signatures as with the minimal changes technique, except we completely delete all executable code, replacing each method body by code that throws a `StubException` (a new exception that extends `UnsupportedOperationException`).

When we compile the generic client, we do so against the class files generated from the stub code, which contain appropriate generic signatures (say, in directory `s`). When we run the client, we do so against the original legacy class files (say, in directory `l`).

```
% javac -classpath s g/Client.java
% java -ea -classpath l g/Client
```

Again, this works because the class files generated for legacy and generic files are essentially identical, save for auxiliary information about the types. In particular, the generic signatures that client is compiled against match the legacy signatures (apart from auxiliary information about type parameters), so the code runs successfully and gives the same answer as previously.

Example 4.3 Evolving a library — minimal changes

```
m/Stack.java:
    interface Stack<E> extends java.util.List {
        public boolean empty();
        public void push(E elt);
        public E pop();
    }

m/ArrayStack.java:
    @SuppressWarnings("unchecked")
    class ArrayStack<E> extends java.util.ArrayList implements Stack<E> {
        public boolean empty() { return size()==0; }
        public void push(E elt) { add(elt); }
        public E pop() {
            Object elt = get(size()-1);
            remove(size()-1);
            return (E)elt;
        }
    }

m/Stacks.java:
    @SuppressWarnings("unchecked")
    class Stacks {
        public static <E> Stack<E> reverse(Stack<E> in) {
            Stack out = new ArrayStack();
            while (!in.empty()) {
                Object elt = in.pop();
                out.push(elt);
            }
            return out;
        }
    }

m/Client.java:
    class Client {
        public static void main (String[] args) {
            Stack<Integer> stack = new ArrayStack<Integer>();
            for (int i = 0; i<4; i++) stack.push(i);
            assert stack.toString().equals("[0, 1, 2, 3]");
            int top = stack.pop();
            assert top==3 && stack.toString().equals("[0, 1, 2]");
            Stack<Integer> reverse = Stacks.reverse(stack);
            assert stack.empty();
            assert reverse.toString().equals("[2, 1, 0]");
        }
    }
```

Example 4.4 Evolving a library — stubs

```
s/Stack.java:
    interface Stack<E> extends List {
        public boolean empty();
        public void push(E elt);
        public E pop();
    }

s/StubException.java:
    class StubException extends UnsupportedOperationException {}

s/ArrayStack.java:
    class ArrayStack<E> extends ArrayList implements Stack<E> {
        public boolean empty() { throw new StubException(); }
        public void push(E elt) { throw new StubException(); }
        public E pop() { throw new StubException(); }
    }

s/Stacks.java:
    class Stacks {
        public static <E> Stack<E> reverse(Stack<E> in) {
            throw new StubException();
        }
    }
```

4.7 Conclusions

To review, we have seen both generic and legacy versions of a library and client. These generate equivalent class files, which greatly eases evolution. One can use a generic library with a legacy client, or a legacy library with a generic client. In the latter case, one can update the legacy library with generic method signatures, either by minimal changes to the source or by use of stub files.

The secret ingredient which makes all of this work is the decision to implement generics by erasure, so that decision that generic code generates essentially the same class files as legacy code. This explains why generic types do not reify information about type parameters at run time, in contrast to the design of arrays which do reify information about array element types at run time.

It is interesting to compare the design of generics in Java and in C#. In C#, generic types do reify information about type information at run time. In some ways, this increases the power of the language. Consider casting from `Object` to `List<String>`. In Java this raises an unchecked warning, because at run-time the only way to tell one has a list of string is to check that every element of the list is string. In C# the object is labeled with its type, so the equivalent cast can be easily checked at run time. However, evolution from legacy to generic code in C# is more difficult. In particular, adaptor classes are needed to view a legacy class as a generic class or vice versa. If one had a nested type, such as `List<List<String>>`, then complex nesting of adaptors will be required. In contrast, as we've seen, evolution in Java can be quite straightforward.

Chapter 5

Reification

The Oxford English Dictionary defines *reify* thus: “To convert mentally into a thing; to materialize.” A plainer word with the same meaning is *thingify*. In computing, *reification* has come to mean an explicit representation of a type. In Java, arrays reify information about their element types, while generic types do not reify information about their type parameters.

The previous chapter was, in a sense, about the advantages of *not* reifying parameter types. Legacy code makes no distinction between `List<Integer>` and `List<String>` and `List<List<String>>`, so not reifying parameter types is essential to easing evolution, and promoting compatibility between legacy code and new code.

But now the time has come to pay the piper. Reification plays a critical role in certain aspects of Java, and the absence of reification that is beneficial for evolution also necessarily leads to some rough edges. This chapter warns you of the limitations and describes some workarounds. We begin with a precise definition of what it means for a type in Java to be *reifiable*. We then consider corner cases related to reification, including instance tests and casts, exceptions, and arrays. The fit between arrays and generics is the worst rough corner in the language, and we encapsulate some of the problems in the Principle of Truth in Advertising and the Principle of Indecent Exposure.

5.1 Reifiable types

In Java, the type of an array is reified *with* its component type, while the type of a parameterized type is reified *without* its type parameters. For instance, an array of numbers might carry the reified type `Number[]`, while a list of numbers might carry the reified type `ArrayList` not `ArrayList<Number>`; it is the raw type but not the parameterized type that is reified. Of course, each element of the list will have a reified type attached to it, say `Integer` or `Double`, but this is not the same as reifying the parameter type. If every element in the list was an integer, we could not tell if what we had was an `ArrayList<Integer>` or `ArrayList<Number>` or `ArrayList<Object>`; if the list was empty, we could not tell what kind of empty list it was.

In Java, we say that a type is *reifiable* if the type is completely represented at run time. To be precise, a type is reifable if it is

- a primitive type (such as `int`),

- a non-parameterized class or interface type (such as `Number`, `String`, or `Runnable`)
- a parameterized type instantiated with unbounded wildcards (such as `List<?>`, `ArrayList<?>`, or `Map<?,?>`).
- a raw type (such as `List`, `ArrayList`, or `Map`).
- or an array whose component type is reifiable (such as `int[]`, `Number[]`, `List<?>[]`, `List[]`, or `int[][]`).

A type is *not* reifiable if it is

- a type variable (such as `T`),
- a parameterized type with actual parameters (such as `List<Number>`, `ArrayList<String>`, or `Map<String,Integer>`),
- or a parameterized type with a bound (such as `List<? extends Number>` or `Comparable<? super String>).`

The type `List<? extends Object>` is *not* reifiable, even though it is equivalent to `List<?>`. Defining reifiable types in this way makes it easy to identify reifiable types syntactically.

5.2 Instance tests and casts

Instance tests and casts depend on examining types at run-time, and hence depend on reification. For this reason, an instance test against a type that is not reifiable reports an error, and a cast to a type that is not reifiable usually issues a warning.

As an example, consider the use of instance tests and casts in writing equality. Here is a fragment of the definition of the class `java.lang.Integer` (slightly simplified from the actual source).

```
public class Integer extends Number {
    private final int value;
    public Integer(int value) { this.value=value; }
    public int intValue() { return value; }
    public boolean equals(Object o) {
        if (!(o instanceof Integer)) return false;
        return value == ((Integer)o).intValue();
    }
    ...
}
```

The equality method takes an argument of type `Object`, checks whether the object is an instance of class `Integer`, and if so casts it to `Integer` and compares the values of the two integers. This code works because `Integer` is a reifiable type: all of the information needed to check whether an object is an instance of `Integer` is available at run time.

Now consider how one might define equality on lists, as in the class `java.util.AbstractList`. A natural — but incorrect! — way to define this is as follows.

```
import java.util.*;
public abstract class AbstractList<E>
extends AbstractCollection<E> implements List<E> {
    public boolean equals(Object o) {
        if (!(o instanceof List<E>)) return false;  compile-time error
        Iterator<E> it1 = iterator();
        Iterator<E> it2 = ((List<E>)o).iterator();  unchecked cast
        while (it1.hasNext() && it2.hasNext()) {
            E e1 = it1.next();
            E e2 = it2.next();
            if (!(e1==null ? e2==null : e1.equals(e2)))
                return false;
        }
        return !it1.hasNext() && !it2.hasNext();
    }
    ...
}
```

Again, the equality method takes an argument of type `Object`, checks whether the object is an instance of type `List<E>`, and if so casts it to `List<E>` and compares corresponding elements of the two lists. This code does *not* work because `List<E>` is *not* a reifiable type: some of the information needed to check whether an object is an instance of `List<E>` is *not* available at run time. One can test whether an object implements the interface `List`, but not whether its type parameter is `E`. Indeed, information on `E` is missing doubly, as it is not available for either the receiver or the argument of the method call.

Compiling the above reports two problems, an error for the instance test and an unchecked warning for the cast.

```
% javac -Xlint:unchecked AbstractList.java
AbstractList.java:6: illegal generic type for instanceof
        if (!(o instanceof List<E>)) return false;  // compile-time error
                           ^
AbstractList.java:8: warning: [unchecked] unchecked cast
found   : java.lang.Object
required: List<E>
        Iterator<E> it2 = ((List<E>)o).iterator();  // unchecked cast
                                    ^
1 error
1 warning
```

The instance check reports an error because there is no possible way to test whether the given object belongs to the type `List<E>`. The cast reports an unchecked warning; it will perform the cast, but it cannot check that the list elements are in fact of type `E`.

To fix the problem, we replace the unreifiable type `List<E>` by the reifiable type `List<?>`. Here is a corrected definition (again, slightly simplified from the actual source).

```
import java.util.*;
public abstract class AbstractList<E>
extends AbstractCollection<E> implements List<E> {
    public boolean equals(Object o) {
        if (!(o instanceof List<?>)) return false;
        Iterator<E> it1 = iterator();
        Iterator<?> it2 = ((List<?>)o).iterator();
        while (it1.hasNext() && it2.hasNext()) {
            E e1 = it1.next();
            Object e2 = it2.next();
            if (!(e1==null ? e2==null : e1.equals(e2)))
                return false;
        }
        return !it1.hasNext() && !it2.hasNext();
    }
    ...
}
```

In addition to changing the type of the instance test and the cast, the type of the second iterator is changed from `Iterator<E>` to `Iterator<?>` and the type of the second element is changed from `E` to `Object`. The code type checks, because even though the element type of the second iterator is unknown, it is guaranteed that it must be a subtype of object, and all the nested call to `equals` requires is that its second argument be an object.

Alternative fixes are possible. Instead of the wildcard types `List<?>` and `Iterator<?>`, one could use use the raw types `List` and `Iterator`, which are also reifiable. We recommend using unbounded wildcard types in preference to raw types because they provide stronger static typing guarantees; many mistakes that are caught as an error when one uses unbounded wildcards will only be flagged as a warning if one uses raw types. Also, one could change the declaration of the first iterator to `Iterator<?>` and of the first element to `Object`, so that they match the second iterator, and the code will still type check. We recommend always using type declarations that are as specific as possible; this helps the compiler to catch more errors and to compile more efficient code.

Non-reifiable casts An instance test against a type that is not reifiable is always an error. However, in some exceptional circumstances, a cast against a type that is not reifiable is permitted.

For example, the following method converts a collection to a list.

```
public static <T> List<T> asList(Collection<T> c)
        throws InvalidArgumentException {
    if (c instanceof List<?>) {
        return (List<T>)c;
    } else throw new InvalidArgumentException("Argument not a list");
}
```

Compiling this codes succeeds with no errors or warnings. The instance test is not in error because `List<?>` is a reifiable type. The cast does not report a warning because the source of the cast has type `Collection<T>` and any object with this type that implements the interface `List` must in fact have type `List<T>`.

Unchecked casts An instance test against a type that is not reifiable is caught as an error, while a cast to a type that is not reifiable is flagged with an unchecked warning. This is because there is never any point to an instance test that cannot be performed, but there may be a point to a cast that cannot be checked.

More generally, there are some circumstances where the programmer knows something about the types that the compiler cannot know; hence the compiler is designed to only issue warnings in these situations, permitting the programmer a workaround in these unusual circumstances.

For example, here is code that promotes a list of objects into a list of strings, if the list of objects contains only strings, and that throws a class cast exception otherwise.

```
import java.util.*;
class Promote {
    public static List<String> promote(List<Object> objs) {
        for (Object o : objs)
            if (!(o instanceof String))
                throw new ClassCastException();
        return (List<String>)(List)objs; // unchecked cast
    }
    public static void main(String[] args) {
        List<Object> objs1 = Arrays.<Object>asList("one","two");
        List<Object> objs2 = Arrays.<Object>asList(1,"two");
        List<String> strs1 = promote(objs1);
        List<String> strs2 = null;
        boolean caught = false;
        try {
            strs2 = promote(objs2);
        } catch (ClassCastException e) { caught = true; }
        assert (List)strs1 == (List)objs1 && strs2 == null && caught;
    }
}
```

The method `promote` loops over the list of objects, and throws a class cast exception if any object is not a string. Hence, when the last line of the method is reached it must be safe to cast the list of objects to a list of strings.

But the compiler cannot know this, so the programmer must use an unchecked cast. It is illegal to cast a list of object to a list of string, so the cast must take place in two steps. First, cast the list of objects into a raw list: this cast is safe. Second, cast the raw list into a list of string: this cast is permitted but generates an unchecked warning.

```
% javac -Xlint:unchecked Promote.java
Promote.java:7: warning: [unchecked] unchecked cast
found   : java.util.List
required: java.util.List<java.lang.String>
        return (List<String>)(List)objs; // unchecked cast
                             ^
1 warning
```

The test code applies the method to two lists, one containing only strings (so it succeeds) and one containing an integer (so it raises an exception). The assertion code at the end confirms that the first object list and the first string list are identical, and that the second string list remains unassigned and that the exception was raised. To compare the object list and the string list we must first cast both to the raw type

List (this cast is safe), because attempting to compare a list of objects with a list of strings raises a type error.

Exactly the same technique can be used to promote a raw list to a list of strings, if the raw list contains only strings. This technique is important for fitting together legacy and generic code, and is one of the chief reasons for using erasure to implement generics.

You should minimize the number of unchecked casts in your code, but sometimes, as in the case above, they cannot be avoided. In this book, we follow the convention that we always place the comment "unchecked cast" on the line containing the cast, to document that this is an intentional workaround rather than an unintended slip, and we recommend you adopt a similar convention. It is important to put the comment on the same line as the cast, so that when scanning the warnings issued by the compiler, it is easy to confirm that each line contains the comment. If it does not, then we regard the warning as equivalent to an error!

If a method deliberately contains unchecked casts, you may wish to precede it with the annotation @SuppressWarnings("unchecked") in order to avoid spurious warnings. We saw an application of this technique in Section 4.5, and it is further discussed in Section 16.2.

As another example of the use of unchecked casts, in Section 5.5 we will see code that uses an unchecked cast from type Object[] to type T[]. Because of the way the object array is created, it is in fact guaranteed that the array will always have the correct type.

Unchecked casts in C (and its descendants C++ and C#) are much more dangerous than unchecked casts in Java. Unlike C, the Java run time guarantees important security properties even in the presence of unchecked casts; for instance, it is never permitted to access an array with an index outside of the array bounds. Nonetheless, unchecked casts in Java are a workaround that should be used with caution.

5.3 Exception handling

In a *try* statement, each *catch* clause checks whether the thrown exception matches a given type. This is the same as the check performed by an instance test, so the same restriction applies: the type must be reifiable. Further, the type in a *catch* clause is required to be a subclass of Throwable. Since there is little point in creating a subclass of Throwable that cannot appear in a *catch* clause, the Java compiler complains if you attempt to create a parameterized subclass of Throwable.

For example, here is a permissible definition of a new exception, which contains an integer value.

```
class IntegerException extends Exception {
    private final int value;
    public IntegerException(int value) { this.value = value; }
    public int getValue() { return value; }
}
```

And here is a simple example of how to use the exception.

```
class IntegerExceptionTest {
    public static void main (String[] args) {
        try {
            throw new IntegerException(42);
        } catch (IntegerException e) {
            assert e.getValue()==42;
        }
    }
}
```

The body of the *try* statement throws the exception with a given value, which is caught by the *catch* clause.

In contrast, the following definition of a new exception is prohibited, because it creates a parameterized type.

```
class ParametricException<T> extends Exception {  // compile-time error
    private final T value;
    public ParametricException(T value) { this.value = value; }
    public T getValue() { return value; }
}
```

An attempt to compile the above reports an error.

```
% javac ParametricException.java
ParametricException.java:1: a generic class may not extend java.lang.Throwable
class ParametricException<T> extends Exception {  // compile-time error
                                     ^
1 error
```

This restriction is sensible because almost any attempt to catch such an exception must fail, because the type is not reifiable. One might expect a typical use of the exception to be something like the following.

```
class ParametricExceptionTest {
    public static void main (String[] args) {
        try
            throw new ParametricException<Integer>(42);
         catch (ParametricException<Integer> e) {  // compile-time error
            assert e.getValue()==42;
        }
    }
}
```

This is not permitted, because the type in the *catch* clause is not reifiable. At the time of writing, the Sun compiler reports a cascade of syntax errors in such a case.

```
% javac ParametricExceptionTest.java
ParametricExceptionTest.java:5: <identifier> expected
        } catch (ParametricException<Integer> e) {
                                    ^
ParametricExceptionTest.java:8: ')' expected
    }
    ^
ParametricExceptionTest.java:9: '}' expected
}
 ^
3 errors
```

Because the type in a *catch* clause must be reifiable, the syntax is restricted so that the type must be written as an identifier, with no following parameter.

Type variable in a throws clause Although subclasses of `Throwable` cannot be parametric, it is possible to use a type variable in a *throws* clause in a method declaration. This may be useful when different instances of a class contain methods that may raise different checked exceptions.

Recall that the class `Throwable` has two major subclasses, `Exception` and `Error`, and that the first of these has another major subclass, `RuntimeException`. An exception is *checked* if it is a subclass of `Exception` but not a subclass of `RuntimeException`. The *throws* clause of a method may list any subclass of `Throwable`, but *must* list any checked exception that might be thrown by the method body, including any checked exceptions declared for the methods invoked within the body.

An example of the use of a type variable in a *throws* clause is shown in Figure 5.1. The figure defines a class `Function<A,B,T>`, which represents a function. The class contains an abstract method `apply` that accepts an argument of type `A`, returns a result of type `B`, and may throw an exception of type `T`. The class also contains a method `applyAll` that accepts an argument of type `List<A>`, returns a result of type `List<B>`, and again may throw an exception of type `T`; the method invokes the `apply` method on each element of the argument list to produce the result list.

The main method of the class defines three objects of this type. The first is `length` of type `Function<String,Integer,Error>`. It accepts a string and returns an integer, which is the length of the given string. Since it raises no checked exceptions, the third type is set to `Error`. (Setting it to `RuntimeException` would work as well.)

The second is `forName` of type `Function<String,Class<?>,ClassNotFoundException>`. It accepts a string and returns a class, namely the class named by the given string. The `apply` method may throw a `ClassNotFoundException`, so this is taken as the third type parameter.

The third is `getRunMethod` of type `Function<String,Method,Exception>`. It accepts a string and returns a method, namely the method named `run` in the class named by the given string. The body of the method might raise either a `ClassNotFoundException` or a `NoSuchMethodException`, so the third type parameter is taken to be `Exception`, the smallest class that contains both of these exceptions.

The main method also defines `strings` of type `List<String>` to be a list of the strings passed when invoking the method. It then uses `applyAll` to apply each of the three functions to this list. Each of the three invocations is wrapped in a *try* statement appropriate to the exceptions it is may throw. Function `length` has no *try* statement, because it throws no checked exceptions. Function `forName` has a *try* statement with a *catch* clause for `ClassNotFoundException`, the one kind of exception it may throw. Function `getRunMethod` requires a *try* statement with *catch* clauses for

ClassNotFoundException and NoSuchMethodException, the two kinds of exception it may throw. But the function is declared to throw type Exception, so we need two additional "catch-all" clauses, one to rethrow any run-time exception that is raised, and to assert that no other exception should be raised other than those handled by the previous three clauses. For this particular example, re-raising runtime exceptions is not required, but it is good practice if there may be other code that handles such exceptions.

For example, here is a typical run of the code, printing the list of lengths, the list of classes, and the list of methods. (The last list has been reformatted for readability, since it doesn't fit on one line.)

```
%  java Function java.lang.Thread java.lang.Runnable
[16, 18]
[class java.lang.Thread, interface java.lang.Runnable]
[public void java.lang.Thread.run(),
 public abstract void java.lang.Runnable.run()]
```

And here is a run that raises NoSuchMethodException, since java.util.List has no run method.

```
% java Function java.lang.Thread java.util.List
[16, 14]
[class java.lang.Thread, interface java.util.List]
java.lang.NoSuchMethodException: java.util.List.run()
```

And here is a run that raises ClassNotFoundException, since there is no class named Fred.

```
% java Function java.lang.Thread Fred
[16, 8]
java.lang.ClassNotFoundException: Fred
java.lang.ClassNotFoundException: Fred
```

The exception is raised twice, once when applying forName and once when applying getRunMethod.

5.4 Array creation

Arrays reify their element types. The reified type is used in instance tests and casts, and also used to check that assignments to array elements are permitted. Recall this example from Section 2.5.

```
Integer[] ints = new Integer[] {1,2,3};
Number[] nums = ints;
nums[2] = 3.14;  // array store exception
```

The first line allocates a new array, with reified type information indicating that it is an array of integers. The second line assigns this array to a variable containing an array of numbers; this is permitted because arrays, unlike generic types, are covariant. The assignment on the third line raises an array store exception at run-time, because the assigned value is of type double, and this is not compatible with the reified type attached to the array.

Because arrays must reify their element types, it is an error to create a new array unless its element type is reifiable. The two main problems you are likely to encounter are when the type of the array is a type variable, and when the type of the array is a parameterized type.

Consider the following (incorrect) code to convert a collection to an array.

Example 5.1 Type parameter in a *throws* clause

```
import java.util.*;
import java.lang.reflect.*;
abstract class Function<A,B,T extends Throwable> {
    public abstract B apply(A x) throws T;
    public List<B> applyAll(List<A> list) throws T {
        List<B> result = new ArrayList<B>(list.size());
        for (A x : list) result.add(apply(x));
        return result;
    }
    public static void main (String[] args) {
        Function<String,Integer,Error> length =
            new Function<String,Integer,Error>() {
                public Integer apply(String s) {
                    return s.length();
                }
            };
        Function<String,Class<?>,ClassNotFoundException> forName =
            new Function<String,Class<?>,ClassNotFoundException>() {
                public Class<?> apply(String s)
                    throws ClassNotFoundException {
                    return Class.forName(s);
                }
            };
        Function<String,Method,Exception> getRunMethod =
            new Function<String,Method,Exception>() {
                public Method apply(String s)
                    throws ClassNotFoundException,NoSuchMethodException {
                    return Class.forName(s).getMethod("run");
                }
            };
        List<String> strings = Arrays.asList(args);
        System.out.println(length.applyAll(strings));

        try { System.out.println(forName.applyAll(strings)); }
        catch (ClassNotFoundException e) { System.out.println(e); }

        try { System.out.println(getRunMethod.applyAll(strings)); }
        catch (ClassNotFoundException e) { System.out.println(e); }
        catch (NoSuchMethodException e) { System.out.println(e); }
        catch (RuntimeException e) { throw e; }
        catch (Exception e) { assert false; }
    }
}
```

```
import java.util.*;
class Annoying {
    public static <T> T[] toArray(Collection<T> c) {
        T[] a = new T[c.size()];  // compile-time error
        int i=0; for (T x : c) a[i++] = x;
        return a;
    }
}
```

This is an error, because a type variable is not a reifiable type. An attempt to compile this code reports a *generic array creation* error.

```
% javac Annoying.java
Annoying.java:4: generic array creation
        T[] a = new T[c.size()];  // compile-time error
                ^
1 error
```

We will discuss below workarounds for this problem below.

As a second example, consider the following (incorrect) code that returns an array containing two lists.

```
import java.util.*;
class AlsoAnnoying {
    public static List<Integer>[] twoLists() {
        List<Integer> a = Arrays.asList(1,2,3);
        List<Integer> b = Arrays.asList(4,5,6);
        return new List<Integer>[] {a, b};  // compile-time error
    }
}
```

This is an error, because a parameterized type is not a reifiable type. An attempt to compile this code also reports a *generic array creation* error.

```
% javac AlsoAnnoying.java
AlsoAnnoying.java:6: generic array creation
        return new List<Integer>[] {a, b};  // compile-time error
                                   ^
1 error
```

We also discuss below workarounds for this problem below.

Inability to create generic arrays is one of the most serious restrictions in Java. Because it is so annoying, it is worth reiterating why it occurs: generic arrays are problematic because generics are implemented via erasure, but erasure is beneficial because it eases evolution.

The best workaround is to use `ArrayList` or some other class from the Collection Framework in preference to an array. We discussed the tradeoffs between collection classes and arrays in Section 2.5, and noted that in many cases collections are preferable to arrays: because they catch more errors at compile-time, because they provide more operations, and because they offer more flexibility in representation. By far, the best solution to the problems offered by arrays is to "just say no": use collections in preference to arrays.

Sometimes this won't work, because you need an array for reasons of compatibility or efficiency. Examples of this occur in the Collections Framework: for compatibility the method `toArray` converts a collection to an array, and for efficiency the class `ArrayList` is implemented by storing the list elements in an array. We discuss both of these cases in detail below, together with associated pitfalls and principles that help you avoid them: the Principle of Truth in Advertising and the Principle of Indecent Exposure. We also consider problems that arise with varargs and generic array creation.

5.5 The Principle of Truth in Advertising

We saw in the previous section that a naive method to convert a collection to an array will not work. The first fix we might try is to add an unchecked cast, but we will see shortly that leads to even more perplexing problems. The correct fix will turn out to require a resort to reflection. Since the same issues arise when converting any generic structure to an array, it is worth understanding the problems and their solution. We will study variations of the static `toArray` method from the previous section; the same ideas apply to the `toArray` method in the interface `Collection` of the Collections Framework.

Here is a second attempt to convert a collection to an array, this time using an unchecked cast, and with test code added.

```
import java.util.*;
class Wrong {
    public static <T> T[] toArray(Collection<T> c) {
        T[] a = (T[])new Object[c.size()];  // unchecked cast
        int i=0; for (T x : c) a[i++] = x;
        return a;
    }
    public static void main(String[] args) {
        List<String> l = Arrays.asList("one","two");
        System.out.println(l);
        String[] a = toArray(l);  // class cast error
    }
}
```

The previous code used the phrase `new T[c.size()]` to create the array, causing the compiler to report a *generic array creation* error. The new code instead allocates an array of object and casts it to type `T[]`, which causes the compiler to issue an *unchecked cast* warning.

```
% javac -Xlint Wrong.java
Wrong.java:4: warning: [unchecked] unchecked cast
found   : java.lang.Object[]
required: T[]
        T[] a = (T[])new Object[c.size()];  // unchecked cast
                     ^
1 warning
```

As you might guess from the name chosen for this program, this warning should not be ignored. Indeed, running this program gives the following result.

```
% java Wrong
Exception in thread "main" java.lang.ClassCastException: [Ljava.lang.Object;
        at Wrong.main(Wrong.java:11)
```

The obscure phrase `[Ljava.lang.Object` is the reified type of the array, where `[L` indicates that it is an array of reference type, and `java.lang.Object` is the component type of the array. The class cast error message refers to the line containing the call to `toArray`. This error message may be confusing, since that line does not appear to contain a cast!

In order to see what went wrong with this program, let's look at how the program is translated using erasure. Erasure drops type parameters on `Collection` and `List`, replaces occurrences of the type variable `T` by `Object`, and inserts an appropriate cast on the call to `toArray`, yielding the following equivalent code.

```
import java.util.*;
class Wrong {
    public static Object[] toArray(Collection c) {
        Object[] a = (Object[])new Object[c.size()];  // unchecked cast
        int i=0; for (Object x : c) a[i++] = x;
        return a;
    }
    public static void main(String[] args) {
        List l = Arrays.asList(args);
        String[] a = (String[])toArray(l);  // class cast error
    }
}
```

Erasure converts the unchecked cast to `T()` into a cast to `Object[]`, and inserts a cast to `String[]` on the call to `toArray`. When run, the first of these casts succeeds. But even though the array only contains strings, its reified type indicates that it is an array of objects, so the second cast fails.

In order to avoid the problem above, you must stick to the following principle.

The Principle of Truth in Advertising. The reified type of an array must be a subtype of the erasure of its static type.

The principles is obeyed within the body of `toArray` itself, where the erasure of `E` is `Object`, but not within the `main` method, where `E` has been instantiated to `String` but the reified type of the array is still `Object`.

Before we see how to create arrays in accordance with this principle, there is one more point worth stressing. Recall that generics for Java are accompanied by a *cast-iron guarantee*: no cast inserted by erasure will fail, so long as there are no unchecked warnings. The above illustrates the converse: if there are unchecked warnings, then casts inserted by erasure may fail. Further, *the cast that fails may be in a different part of the source code than was responsible for the unchecked warning!* This is why code that generates unchecked warnings must be written with extreme care.

Array begets array "Tis money that begets money", said Thomas Fuller in 1732, observing that one way to get money is to already have money. Similarly, the best way to get an new array of a generic type is to already have an array of that type. Then the reified type information for the new array can be copied from the old.

We therefore alter the method above to take two arguments, a collection and an array. If the array is big enough to hold the collection, then the collection is copied into the array. Otherwise, reflection is used to allocate a new array with the same reified type as the old, and then the collection is copied into the new array.

Here is code to implement the alternative.

```
import java.util.*;
class Right {
    public static <T> T[] toArray(Collection<T> c, T[] a) {
        if (a.length < c.size())
            a = (T[])java.lang.reflect.Array.  // unchecked cast
                newInstance(a.getClass().getComponentType(), c.size());
        int i=0; for (T x : c) a[i++] = x;
        if (i < a.length) a[i] = null;
        return a;
    }
    public static void main(String[] args) {
        List<String> l = Arrays.asList("one", "two");
        String[] a = toArray(l, new String[0]);
        assert Arrays.toString(a).equals("[one, two]");
        String[] b = new String[] { "x","x","x","x" };
        toArray(l, b);
        assert Arrays.toString(b).equals("[one, two, null, x]");
    }
}
```

This uses three methods from the reflection library to allocate a new array with the same component type as the old array: method `getClass` (in `java.lang.Object`) returns a class token (of class `Class`) representing the array type, `T[]`; method `getComponentType` (from `java.lang.Class`) returns a second class token (also of class `Class`) representing the array's component type, `T`; and method `newInstance` (in `java.lang.reflect.Array`) allocates a new array with the given component type and size, again of type `T[]`. The result type of the call to `newInstance` is `Object`, so an unchecked cast is required to cast the result to the correct type `T[]`

In Java 5, the class `Class` has been updated to a generic class `Class<T>`; more on this below.

(A subtle point: In the call to `newInstance`, why is the result type `Object` rather than `Object[]`? In general, `newInstance` may return an array of primitive type such as `int[]`, which is a subtype of `Object` but not `Object[]`. That won't happen here because the type variable `T` must stand for a reference type.)

The size of the new array is taken to be the size of the given collection. If the old array is big enough to hold the collection and there is room left over, a null is written just after the collection to mark its end.

The test code creates a list of strings of length two and then performs two demonstration calls. Neither encounters the problem described previously, because the returned array has reified type `String[]`, in accordance with the Principle of Truth in Advertising. The first call is passed an array of length zero, so the list is copied into a freshly allocated array of length two. The second call is passed an array of length four, so the list is copied into the existing array, and a null is written past the end; the original array content after the null is not affected. The utility method `toString` (in `java.util.Arrays`) is used to convert the array to a string in the assertions.

The Collections Framework contains two methods for converting collections to arrays, similar to the one discussed above.

```
interface Collection<E> {
    ...
    public Object[] toArray();
    public <T> T[] toArray(T[] a)
}
```

The first method returns an array with reified component type `Object`, while the second copies the reified component type from the argument array, just as in the static method above. Like that method, it copies the collection into the array if there is room (and writes a null past the end of the collection if there is room for that), and allocates a fresh array otherwise. A call to the first method, `c.toArray()`, is equivalent to a call to the second method with an empty array of objects, `c.toArray(new Object[0])`. These methods are discussed further in Section 5.5.

Often on encountering this design, programmers presume that the array argument exists mainly for reasons of efficiency, in order to minimize allocations by reusing the array. This is indeed a benefit of the design, but its main purpose is to get the reified types correct! Most calls to `toArray` will be with an argument array of length zero.

A classy alternative Some days it may seem that the only way to get money is to have money. Not quite the same is true for arrays. An alternative to using an array to create an array is to use a class token.

In Java 5, the class `Class` has been made generic, and now has the form `Class<T>`. What does the `T` stand for? A class token of type `Class<T>` represents the type `T`. For instance, `String.class` has type `Class<String>`.

We can define a variant of our previous method, that accepts a class token of type `Class<T>` rather than an array of type `T[]`. Applying `newInstance` to a class token of type `Class<T>` returns a new array of type `T[]`, with the component type specified by the class token. The `newInstance` method still has a return type of `Object` (because of the same problem with primitive arrays), so an unchecked cast is still required.

```
import java.util.*;
class RightWithClass {
    public static <T> T[] toArray(Collection<T> c, Class<T> k) {
        T[] a = (T[])java.lang.reflect.Array.  // unchecked cast
                  newInstance(k, c.size());
        int i=0; for (T x : c) a[i++] = x;
        return a;
    }
    public static void main(String[] args) {
        List<String> l = Arrays.asList("one", "two");
        String[] a = toArray(l, String.class);
        assert Arrays.toString(a).equals("[one, two]");
    }
}
```

The test method is now passed the class token `String.class` rather than an array of strings.

The new type `Class<T>` is an interesting use of generics, a use quite different from that for container classes or comparators. If you still find this use of generics confusing, don't worry — we'll cover this subject at greater length in Chapter 6.

5.6 The Principle of Indecent Exposure

Although it is an error to create an array with an element type that is not reifiable, it is *possible* to declare an array with such a type and to perform an unchecked cast to such a type. These features must be used with extreme caution, and it is worthwhile to understand what can go wrong if they are not used properly. In particular, a library should *never* publically expose an array with a non-reifiable type.

Recall that Section 2.5 (page 27), presents an example of why reification is necessary.

```
Integer[] ints = new Integer[] {1};
Number[] nums = ints;
nums[0] = 1.01;  // array store exception
int n = ints[0];
```

This assigns an array of integers to an array of numbers, and then attempts to store a double into the array of numbers. The attempt raises an array store exception because of the check against the reified type. This is just as well, since otherwise the last line would attempt to store a double into an integer variable.

Here is a similar example, where arrays of numbers are replaced by arrays of *lists of* numbers.

```
List<Integer>[] intLists
  = (List<Integer>[])new List[] {Arrays.asList(1)}; // unchecked cast
List<? extends Number>[] numLists = intLists;
numLists[0] = Arrays.asList(1.01);
int n = intLists[0].get(0);  // class cast exception!
```

This assigns an array of lists of integers to an array of lists of numbers, then attempts to store a list of doubles into the array of lists of numbers. This time the attempted store *does not* fail, even though it should, because the check against the reified type is inadequate: the reified information contains only the erasure of the type, indicating that it is an array of `List`, not an array of `List<Integer>`. Hence the store succeeds, and the program fails unexpectedly elsewhere.

Example 5.2 presents a similar example, divided into two classes in order to demonstrate how a poorly designed library can create problems for an innocent client. The first class, called *FaultyLibrary*, defines a static method that returns an array of lists of integers of a given size. Since generic array creation is not permitted, the array is created with elements of raw type `List`, and a cast is used to give the elements the parameterized type `List<Integer>`. The cast generates an unchecked warning.

```
FaultyLibrary.java:5: warning: [unchecked] unchecked cast
found   : java.util.List[]
required: java.util.List<java.lang.Integer>[]
            (List<Integer>[]) new List[size];  // unchecked cast
                              ^
1 warning
```

Since the array really is an array of lists of integers, the cast appears reasonable, and one might think that this warning could be safely ignored. As we shall see, you ignore this warning at your peril!

The second class, called *InnocentClient*, has a main method similar to the example above. Because the unchecked cast appears in the library, no unchecked warning is issued when compiling this code. However, running the code overwrites a list of integers with a list of doubles. Attempting to extract an integer from the array of lists of integer causes the cast implicity inserted by erasure to fail.

```
Exception in thread "main" java.lang.ClassCastException: java.lang.Double
        at InnocentClient.main(InnocentClient.java:7)
```

As in the previous section, this error message may be confusing, since that line does not appear to contain a cast!

In order to avoid the problem above, you must stick to the following principle.

> *Principle of Indecent Exposure*. Never publically expose an array where the elements do not have a reified type.

Again, this is a case where an unchecked cast in one part of the program may lead to a class cast error in a completely different part, where the cast does not appear in the source code but is instead introduced by erasure. Since such errors can be extremely confusing, unchecked casts must be used with extreme caution.

The Principle of Truth in Advertising and the Principle of Indecent Exposure are closely linked. The first requires that the run-time type of an array is properly reified, the second requires that the compile-time type of an array must be reifiable.

It has taken some time for the importance of The Principle of Indecent Exposure to be understood, even among the designers of generics for Java. For example, the following two methods in the library violate the principle.

```
public TypeVariable<Class<T>>[] java.lang.Class.getTypeParameters()
public TypeVariable<Method>[] java.lang.Reflect.method.getTypeParameters()
```

Following the model above, it is not hard to create your own version of `InnocentClient` that throws a class cast error at a point where there is no cast, where in this case the role of `FaultyLibrary` is played by the official Java library! (At the time of going to press, remedies for this bug are under consideration. Possible fixes are to delete the type parameter from `TypeVariable`, so that the methods return an array of reified type; or to replace the arrays with lists.)

Don't get caught out in the same way: be sure to follow the Principle of Indecent Exposure rigorously in your own code!

5.7 How to define ArrayList

We have argued elsewhere that it is usually preferable to use a list to an array. There are a few places where this is not appropriate. In rare circumstance, one will need to use an array for reasons of efficiency or compatibility. Also, of course, one needs to use arrays to implement `ArrayList` itself. Here we present the implementation of `ArrayList` as a model of what to do in the rare circumstance where one does need to use an array. Such implementations need to be written with care, as they necessarily involve use of unchecked casts. We see how the Principles of Indecent Exposure and of Truth in Advertising figure in the implementation.

Example 5.3 shows the implementation. We have derived `ArrayList` by subclassing from `AbstractList`. Classes derived from this class only need to define four methods, namely `get`, `set`, `add`, and `remove`, the other methods being defined in terms of these. We also indicate that the class implements `RandomAccess`, indicating that clients of the class will have more efficient access using `get` than using the list iterator.

The class represents a list with elements of type `E` by two private fields, `size` of type `int` containing the length of the list, and `arr` of type `E[]` containing the elements of the list. The array must have length at least equal to `size`, but it may have additional unused element at the end.

There are two places where new instances of the array are allocated, one in the initializer for the class and one in the method that increases the array capacity (which in turn is called from the `add` method). In both places, the array is allocated as an `Object[]` and an unchecked cast is made to the type `E[]`.

It is essential that the field containing the array is private, otherwise it would violate both the Principle of Truth in Advertising and the Principle of Indecent Exposure. It would violate the Principle of Truth

Example 5.2 Avoid arrays of non-reifiable type

```
FaultyLibrary.java:
import java.util.*;
public class FaultyLibrary {
    public static List<Integer>[] intLists(int size) {
        List<Integer>[] intLists =
            (List<Integer>[]) new List[size];  // unchecked cast
        for (int i = 0; i < size; i++)
            intLists[i] = Arrays.asList(i+1);
        return ints;
    }
}

InnocentClient.java:
import java.util.*;
public class InnocentClient {
    public static void main(String[] args) {
        List<Integer>[] intLists = FaultyLibrary.intLists(1);
        List<? extends Number>[] numLists = intLists;
        numLists[0] = Arrays.asList(1.01);
        int i = intLists[0].get(0);  // class cast error!
    }
}
```

in Advertising, because `E` might be instantiated to a type (like `String`), the erasure of which does not have `Object` as as subtype. It would violate the Principle of Indecent Exposure, because `E` might be instantiated to a type (like `List<Integer>`) that is not a reifiable type. However, neither of these principles is violated because the array is not public: it is stored in a private field, and no pointer to the array escapes from the class. We might call this the Principle of Anything Goes Behind Closed Doors.

The method `toArray` does return an array in public, but it uses the techniques described in Section 5.5 to keep in accord with the Principle of Truth in Advertising. As before, there is an argument array, and if it is not big enough to hold the collection then reflection is used to allocate a new array with the same reified type. The implementation is similar to the one we saw earlier, except that the more efficient `arraycopy` routine can be used to copy the private array into the public array to be returned.

5.8 Array creation and varargs

The convenient *vararg* notation allows methods to accept a variable number of arguments and packs them into an array, as discussed in Section 1.4. This notation is not as convenient as one might like, because the arrays it creates suffer from the same issues involving reification as other arrays.

In Section 1.4 we discussed the method `java.util.Array.asList`, which is declared as follows.

```
public static <E> List<E> asList(E... arr)
```

For instance, here are three calls to this method.

```
List<Integer> a = Arrays.asList(1, 2, 3);
List<Integer> b = Arrays.asList(4, 5, 6);
List<List<Integer>> x = Arrays.asList(a, b);  // generic array creation
```

Recall that an argument list of variable length is implemented by packing the arguments into an array and passing that. Hence the three calls above are equivalent to the following.

```
List<Integer> a = Arrays.asList(new Integer[] { 1, 2, 3 });
List<Integer> b = Arrays.asList(new Integer[] { 4, 5, 6 });
List<List<Integer>> x
    = Arrays.asList(new List<Integer>[] { a, b });  // generic array creation
```

The first two calls are fine, but since `List<Integer>` is not a reifiable type, the third warns of an unchecked "generic array creation" at compile time.

```
VarargError.java:6: warning: [unchecked] unchecked generic array creation
of type java.util.List<java.lang.Integer>[] for varargs parameter
          List<List<Integer>> x = Arrays.asList(a, b);
```

This warning can be confusing, particularly since that line of source code does not contain an explicit instance of array creation!

Normally, generic array creation reports an error. As a workaround, one can create the array at a reifiable type and perform an unchecked cast. That workaround is not available for the array creation that is implicit in the use of varargs, so in this case generic array creation issues a warning rather than an error. In our opinion, the convenience offered by varargs is outweighed by the danger inherent in unchecked warnings, and we recommend that you never use varargs when the argument is of a non-reifiable type.

Example 5.3 How to define `ArrayList`

```
import java.util.*;
class ArrayList<E> extends AbstractList<E> implements RandomAccess {
    private E[] arr;
    private int size = 0;
    public ArrayList(int cap) {
        if (cap < 0)
            throw new IllegalArgumentException("Illegal Capacity: "+cap);
        arr = (E[])new Object[cap];  // unchecked cast
    }
    public ArrayList() { this(10); }
    public ArrayList(Collection<? extends E> c) {
        this(c.size()); addAll(c);
    }
    public void ensureCapacity(int cap) {
        if (cap > arr.length) {
            E[] oldarr = arr;
            arr = (E[])new Object[cap];  // unchecked cast
            System.arraycopy(oldarr,0,arr,0,size);
        }
    }
    public int size() { return size; }
    private void checkBounds(int i, int size) {
        if (i < 0 || i >= size)
            throw new IndexOutOfBoundsException("Index: "+i+", Size: "+size);
    }
    public E get(int i) {
        checkBounds(i,size); return arr[i];
    }
    public E set(int i, E elt) {
        checkBounds(i,size); E old = arr[i]; arr[i] = elt; return old;
    }
    public void add(int i, E elt) {
        checkBounds(i,size+1);
        ensureCapacity(size+1);
        System.arraycopy(arr,i,arr,i+1,size-i);
        arr[i] = elt;  size++;
    }
    public E remove(int i) {
        checkBounds(i,size);
        E old = arr[i];  size--;
        System.arraycopy(arr,i+1,arr,i,size-i);
        return old;
    }
    public <T> T[] toArray(T[] a) {
        if (a.length < size)
            a = (T[])java.lang.reflect.Array.  // unchecked cast
                     newInstance(a.getClass().getComponentType(), c.size());
        System.arraycopy(arr,0,a,0,size);
        if (size < a.length) a[size] = null;
        return a;
    }
}
```

This problem would not have arisen if the vararg notation had been defined to pack arguments into a list rather than an array, taking `T...` to be equivalent to `List<T>` rather than `T[]`. Unfortunately, the vararg notation was designed before this problem was fully understood. Perhaps a future version of Java will supply a second form of vararg, that uses lists in place of arrays. It may be time to start thinking of arrays as a deprecated type!

5.9 Summing up

We conclude by giving a checklist of places where reifiable types are required or recommended.

- An instance test must be against a reifiable type.
- A cast should usually be at a reifiable type. (A cast against a non-reifiable type usually issues an unchecked warning.)
- A class that extends `Throwable` must not be parameterized.
- An array instance creation must be at a reifiable type.
- The reified type of an array must be a subtype of the erasure of its static type (See the Principle of Truth in Advertising), and a publically exposed array should be of a reifiable type (See the Principle of Indecent Exposure).
- Varargs should be of a reifiable type. (A vararg of a non-reifiable type will issue an unchecked warning.)

These restrictions arise from the fact that generics are implemented via erasure, and should be regarded as the price one pays for the ease of evolution that we explored in the previous chapter.

For completeness, we also list here restrictions connected with reflection.

- Class tokens correspond to reifiable types, and the type parameter in `Class<T>` should be a reifiable type. (See Section 6.2.)

These are discussed in the next chapter.

Chapter 6

Reflection

Reflection is the name for a set of features that allows a program to examine its own definition. Reflection in Java plays a role in class browsers, object inspectors, debuggers, interpreters, services such as JavaBeans and Object Serialization, and any tool that creates, inspects, or manipulates arbitrary Java objects on the fly.

Reflection has been present in Java since the beginning, but the advent of generics changes reflection in two important ways, introducing both *generics for reflection* and *reflection for generics*.

By *generics for reflection* we mean that some of the types used for reflection are now generic types. In particular, the class `Class` becomes a generic class, `Class<T>` — this may seem confusing at first, but once understood it can make programs using reflection much clearer. Class literals and the special method `Object.getClass` use special tricks to return more precise type information. Generics are used to especially good effect in the reflection of annotations. We observe that the type parameter `T` in `Class<T>` should always be instantiated to a reifiable type, and we present a short library use of which can avoid many common cases of unchecked casts.

By *reflection for generics* me mean that reflection now returns information about generic types. There are new interfaces to represent generic types, including type variables, parameterized types, and wildcard types, and there are new methods that get the generic types of fields, constructors, and methods.

We explain each of these points in turn. We don't assume any previous knowledge of reflection, but we focus on the aspects tied to generics.

6.1 Generics for reflection

Java has contained facilities for *reflection* since version 1.0. Central to these is a class `Class`, that represents information about the type of an object at run-time. One may write a type followed by `.class` as a literal that denotes the class token corresponding to that type, and the method `getClass` is defined on every object, and returns a class token that represents the reified type information carried by that object at run-time. Here is an example.

```
Class ki = Integer.class;
Number n = new Integer(42);
Class kn = n.getClass();
assert ki == kn;
```

The class loader ensures that the same type is always represented by the same class token, so it is appropriate to compare class tokens using identity (the == operator) rather than equality (the `equals` method).

One of the changes in Java 5 is that class `Class` now takes a type parameter, so that `Class<T>` is the type of the class token for the type `T`. The code above is now written as follows.

```
Class<Integer> ki = Integer.class;
Number n = new Integer(42);
Class<? extends Number> kn = n.getClass();
assert ki == kn;
```

Class tokens and the `getClass` method are treated specially by the compiler. In general, if T is a type without type parameters, then T`.class` has type `Class<`T`>`, and if e is an expression of type T then e`.getClass()` has type `Class<? extends `T`>`. (We'll see what happens when T does have type parameters shortly.) The wildcard is needed because the type of the object referred to by the variable may be a subtype of the type of the variable, as in this case where a variable of type `Number` contains an object of type `Integer`.

For many uses of reflection, you won't know the exact type of the class token (if you did, you probably wouldn't need to use reflection), and in those cases you can write `Class<?>` for the type, using an unbounded wildcard. However, in some cases the type information provided by the type parameter is invaluable, as in the variant of `toArray` that we discussed above.

```
public static <T> T[] toArray(Collection<T> c, Class<T> k)
```

Here the type parameter lets the compiler check that the type represented by the class token matches the type of the collection and of the array.

Further examples of generics for reflection The class `Class<T>` contains just a few methods that use the type parameter in an interesting way.

```
class Class<T> {
    public T newInstance();
    public T cast(Object o);
    public Class<? super T> getSuperclass();
    public <U> Class<? extends U> asSubclass(Class<U> k);
    public <A extends Annotation> A getAnnotation(Class<A> k);
    public boolean isAnnotationPresent(Class<? extends Annotation> k);
    ...
}
```

The first returns a new instance of the class, which will of course have type `T`. The second casts an arbitrary object to the receiver class, and so either throws a class cast exception or returns a result of type `T`. The third returns the superclass, which must have the specified type. The fourth checks that the receiver class is a subclass of the argument class, and either throws a class cast exception or returns the receiver with its type suitably changed.

The fifth and sixth methods are part of the new annotation facility. The methods are interesting, because they show how the type parameter for classes can be used to good effect. For example, `Retention` is a subclass of `Annotation`, so one can extract the 'retention' annotation on a class `k` as follows.

```
Retention r = k.getAnnotation(Retention.class);
```

Here the generic type gains two advantages. First, it means that no cast is required on the result of the call, because the generic type system can assign it precisely the correct type. Second, it means that if one accidentally calls the method with a class token for a class that is not a subclass of `Annotation`, then this is detected at compile-time rather than run-time.

Another use of class tokens, similar to that for annotations, appears in the `getListeners` method of the class `Component` in package `java.awt`.

```
public <T extends EventListener>
    T[] getListeners(Class<T> listenerType);
```

Again, this means that the code of `getListeners` requires no cast, and it means that the compiler can check that the method is called with a class token of an apprörpriate type.

As as final example of an interesting use of class tokens, the convenience class `Collections` contains a method that builds a wrapper that checks whether every element added to or extracted from the given list belongs to the given class. (There are similar methods for other collection classes, such as sets and maps.) It has the following signature.

```
public static <T> List<T> checkedList(List<T> l, Class<T> k)
```

The wrapper supplements static checking at compile time with dynamic checking at run time, which can be useful to improve security or to interface with legacy code. The implementation calls the method `cast` in class `Class` described above, where the receiver is the class token passed into the method, and the cast is applied to any element read from or written into the list using `get`, `set`, or `add`. Yet again, the type parameter on `Class<T>` means that the code of `checkList` requires no additional casts (beyond calls to the `cast` method in class `Class`), and that the compiler can check that the method is called with a class token of an approrpriate type. Wrapper methods are further discussed in Section 17.1.

6.2 Reflected types are reifiable types

Reflection makes reified type information available to the program. Of necessity, therefore, each class token corresponds to a reifiable type. If you try to reflect a parameterized type, you get the reified information for the corresponding raw type.

```
List<Integer> ints = new ArrayList<Integer>();
List<String> strs  = new ArrayList<String>();
assert ints.getClass() == strs.getClass();
assert ints.getClass() == ArrayList.class;
```

Here the type list of integers and the type list of strings are both represented by the same class token, the class literal for which is written `ArrayList.class`.

Because the class always represents a reifiable type, there is no point in parameterizing the class `Class` with a type that is not reifiable. Hence the two main methods for producing a class with a type parameter, namely the `getClass` method and class literals, both are adjusted to always produce a reifiable type for the type parameter.

Recall that the `getClass` method is treated specially by the compiler. In general, if e is an expression of type T, then e`.getClass()` has type `Class<? extends` $|T|$`>`, where $|T|$ is the erasure of the type $|T|$. Here's an example.

```
List<Integer> ints = new ArrayList<Integer>();
Class<? extends List> k = ints.getClass();
assert k == ArrayList.class;
```

Here `ints` has type `List<Integer>` so that `int.getClass()` has type `Class<? extends List>`, since erasing `List<Integer>` yields the raw type `List`. The actual value of `k` is `ArrayList.class` which has type `Class<ArrayList>`, which is indeed a subtype of `Class<? extends List>`.

Class literals are also restricted; it is not even syntactically valid to supply a type parameter to the type in a class literal. Thus, the following fragment is illegal.

```
class ClassLiteral {
    public Class<?> k = List<Integer>.class;  // syntax error
}
```

Indeed, Java's grammar makes a phrase like the above difficult to parse, and it may trigger a cascade of syntax errors.

```
% javac ClassLiteral.java
ClassLiteral.java:2: illegal start of expression
    public Class<?> k = List<Integer>.class;  // syntax error
                                     ^
ClassLiteral.java:2: ';' expected
    public Class<?> k = List<Integer>.class;  // syntax error
                                      ^
ClassLiteral.java:2: <identifier> expected
    public Class<?> k = List<Integer>.class;  // syntax error
                                            ^
ClassLiteral.java:4: '}' expected
^
4 errors
```

The parser has so much trouble with this phrase that it is still confused when it reaches the end of the file!

This syntax problem leads to an irregularity. Everywhere else that a reifiable type is required, one may supply either a raw type (such as `List`) or a parameterized type with unbounded wildcards (such as `List<?>`). However, for class tokens one must supply a raw type, not even unbounded wildcards may appear. Replacing `List<Integer>` by `List<?>` in the above leads to a similar error cascade.

The restrictions on class tokens lead to a useful property. Wherever a type of the form `Class<T>` appears, the type `T` should be a reifiable type. The same is true for types of the form `T[]`.

6.3 Reflection for primitive types

Every type in Java has a corresponding class token, including primitive types and array types, and these also have class literals.

For instance, `int.class` denotes the class token for the primitive type for integers (this token is also the value of the static field `Integer.TYPE`). The type of this class token cannot be `Class<int>`, since `int` is not a reference type, so it is taken to be `Class<Integer>`. Arguably, this is an odd choice, since according to this type one might expect the calls `int.class.cast(o)` or `int.class.newInstance()` to return a value of type `Integer`, but in fact these calls raise an exception. Similarly, one might expect the call

```
java.lang.reflect.array.newInstance(int.class,size)
```

to return a value of type `Integer[]`, but in fact the call returns a value of type `int[]`. These examples suggest it might have made more sense to give the class token `int.class` the type `Class<?>`.

On the other hand, `int[].class` denotes the class token for arrays with elements of the primitive type integer, and the type of this class token is `Class<int[]>`, which is permitted since `int[]` is a reference type.

6.4 A generic reflection library

As we've seen, careless use of unchecked casts can lead to problems, such as violating the Principle of Truth in Advertising or the Principle of Indecent Exposure (see Sections 5.5 and 5.6). One technique to minimize the use of unchecked casts is to encapsulate these within a library. The library can be carefully scrutinized to ensure its use of unchecked casts is safe, while code that calls the library can be free of unchecked casts.

Example 6.1 provides a library of generic functions that use reflection in a type-safe way. It defines a convenience class `GenericReflection` containing the following methods.

```
public static <T> T newInstance(T object)
public static <T> Class<T> getComponentType(T[] a)
public static <T> T[] newInstance(Class<T> k, int size)
public static <T> T[] newInstance(T[] a, int size)
```

The first takes an object, finds the class of that object, and returns a new instance of the class; this must have the same type as the original object. The second takes an array and returns a class token for its component type. Conversely, the third allocates a new array with component type specified by a given class token and a specified size. The fourth takes an array and a size, and allocates a new array with the same component type as the given array and the given size; it simply composes calls to the previous two methods. The code for each of the first three methods consists of a call to one or two corresponding methods in the Java reflection library and an unchecked cast to the appropriate return type.

Unchecked casts are required because the methods in the Java reflection library cannot return sufficiently accurate types for various reasons. The method `getComponentType` is in class `Class<T>`, and Java provides no way to restrict the receiver type to be `Class<T[]>` in the signature of the method (though the call raises an exception if the receiver is not a class token for an array type). The method `newInstance` in `java.lang.reflect.Array` must have return type `Object`, rather than return type `T[]`, because it may return an array of primitive type. The method `getClass` when called on a reciever of type `T` returns a token not of type `Class<? extend T>` but of type `Class<?>`, because of the erasure that is required to ensure that class tokens always have a reifiable type. However, in each case the unchecked cast is safe, and users can call on the four library routines defined here without violating the cast-iron guarantee.

The third method raises an illegal argument exception if its class argument is a primitive type. This catches the following tricky case: if the first argument is, say `int.class`, then its type is `Class<Integer>`, but the new array will have type `int[]` which is not a subtype of

Integer[]. This problem would not have arisen if int.class had the type Class<?> rather than Class<Integer>.

As an example of the use of the first method, here is a method that copies a collection into a fresh collection of the same kind, preserving precisely the type of the argument.

```
public static <T, C extends Collection<T>> C copy(C coll) {
    C copy = GenericReflection.newInstance(coll);
    copy.addAll(coll);  return copy;
}
```

Calling copy on an ArrayList returns a new ArrayList, while calling copy on a HashSet returns a new HashSet.

As an example of the use of the last method, here is the toArray method of Section 5.5, rewritten to replace its unchecked casts by a call to the generic reflection library.

```
public static <T> T[] toArray(Collection<T> c, T[] a) {
    if (a.length < c.size())
        a = GenericReflection.newInstance(a, c.size());
    int i=0; for (T x : c) a[i++] = x;
    if (i < a.length) a[i] = null;
    return a;
}
```

It would be helpful if Sun would define a convenience class in the reflection library similar to this one, in order to encourage a style of coding that encapsulates unchecked casts into a few well-designed library routines.

6.5 Reflection for generics

Generics change the reflection library in two ways. We have discussed *generics for reflection*, where Java added a type parameter to the class Class<T>. We now discuss *reflection for generics*, where Java adds methods and classes that support access to generic types.

Example 6.2 shows a simple demonstration of the use of reflection for generics. It uses reflection to find the class associated with a given name, and prints out the fields, constructors, and methods associated with the class, using the reflection library classes Field, Constructor, and Method. Two different methods are available for converting a field, constructor, or method to a string for printing, the old toString method and the new toGenericString method; the old method is maintained mainly for backward compatibility. A small sample class is shown in Example 6.3, and a sample run with this class is shown in Example 6.4.

The example demonstrates that although the reified type information for objects and class tokens keeps no information about generic types, the actual byte code of the class does encode information about generic types as well as erased types. The information about generic types is essentially a comment, and is ignored when running the code; it is only present for use in reflection.

Unfortunately, there is no toGenericString method for the class Class itself, even though this would be useful. This is likely to be fixed in a future version. In the meantime, all the necessary information is available, and we explain how to access it in the next section.

Example 6.1 A type-safe library for generic reflection

```
class GenericReflection {
    public static <T> T newInstance(T object) {
        return (T)object.getClass().newInstance();  // unchecked cast
    }
    public static <T> Class<T> getComponentType(T[] a) {
        return (Class<T>)a.getClass().getComponentType();  // unchecked cast
    }
    public static <T> T[] newInstance(Class<T> k, int size) {
        if (k.isPrimitive())
            throw new IllegalArgumentException
                ("Argument cannot be primitive: "+k);
        return (T[])java.lang.reflect.Array.  // unchecked cast
                newInstance(k, size);
    }
    public static <T> T[] newInstance(T[] a, int size) {
        return newInstance(getComponentType(a), size);
    }
}
```

6.6 Reflecting generic types

The reflection library provides an interface `Type` to describe a generic type. There is one class that implements this interface and four other interfaces that extend it, corresponding to the five different kinds of types:

- class `Class`, representing a primitive type or raw type;
- interface `ParameterizedType`, representing a generic class or interface, from which you can extract an array of the actual parameter types;
- interface `TypeVariable`, representing a type variable, from which you can extract the bounds on the type variable;
- interface `GenericArrayType`, representing an array, from which you can extract the array component type;
- interface `WildcardType`, representing a wildcard, from which you can extract a lower or upper bound on the wildcard.

By performing a series of instance tests on each of these interfaces, you may determine which kind of type you have, and print or process the type; we will see an example of this shortly.

Methods are available to return the superclass and superinterfaces of a class as types, and to access the generic type of a field, the argument types of a constructor, and the argument and result types of a method.

One can also extract the type variables that stand for the formal parameters of a class or interface declaration, or of a generic method or constructor. The type for type variables takes a parameter, and

Example 6.2 Reflection for generics

```
import java.lang.reflect.*;
import java.util.*;
class ReflectionForGenerics {
    public static void toString(Class<?> k) {
        System.out.println(k + " (toString)");
        for (Field f : k.getDeclaredFields())
            System.out.println(f.toString());
        for (Constructor c : k.getDeclaredConstructors())
            System.out.println(c.toString());
        for (Method m : k.getDeclaredMethods())
            System.out.println(m.toString());
        System.out.println();
    }
    public static void toGenericString(Class<?> k) {
        System.out.println(k + " (toGenericString)");
        for (Field f : k.getDeclaredFields())
            System.out.println(f.toGenericString());
        for (Constructor c : k.getDeclaredConstructors())
            System.out.println(c.toGenericString());
        for (Method m : k.getDeclaredMethods())
            System.out.println(m.toGenericString());
        System.out.println();
    }
    public static void main (String[] args)
    throws ClassNotFoundException {
        for (String name : args) {
            Class<?> k = Class.forName(name);
            toString(k);
            toGenericString(k);
        }
    }
}
```

Example 6.3 A sample class

```
class Cell<T> {
    private T value;
    public Cell(T value) { this.value=value; }
    public T getValue() { return value; }
    public void setValue(T value) { this.value=value; }
    public static <T> Cell<T> copy(Cell<T> cell) {
        return new Cell<T>(cell.getValue());
    }
}
```

Example 6.4 A sample run

```
% java ReflectionForGenerics Cell
class Cell (toString)
private java.lang.Object Cell.value
public Cell(java.lang.Object)
public java.lang.Object Cell.getValue()
public static Cell Cell.copy(Cell)
public void Cell.setValue(java.lang.Object)

class Cell (toGenericString)
private T Cell.value
public Cell(T)
public T Cell.getValue()
public static <T> Cell<T> Cell.copy(Cell<T>)
public void Cell.setValue(T)
```

is written `TypeVariable<D>`, where `D` represents the type of object that declared the type variable. Thus, the type variables of a class have type `TypeVariable<Class<?>>`, while the type variables of a generic method have type `TypeVariable<Method>`. Arguably, the type parameter is confusing and not very helpful. Since it is responsible for the problem described in Section 5.6, it may be removed in future.

Example 6.5 uses these methods to print out all of the header information associated with a class. Here are two examples of its use.

```
% java ReflectionDemo java.util.AbstractList
class java.util.AbstractList<E>
extends java.util.AbstractCollection<E>
implements java.util.List<E>

% java ReflectionDemo java.lang.Enum
class java.lang.Enum<E extends java.lang.Enum<E>>
implements java.lang.Comparable<E>,java.io.Serializable
```

Most of this code would be unnecessary if the interface `Type` supported a `toGenericString` method. This may happen in future.

Example 6.5 How to manipulate the type `Type`.

```
import java.util.*;
import java.lang.reflect.*;
import java.io.*;
class ReflectionDemo {
    private final static PrintStream out = System.out;
    public static void printSuperclass (Type sup) {
        if (sup != null && !sup.equals(Object.class)) {
            out.print("extends ");
            printType(sup);
            out.println();
        }
    }
    public static void printInterfaces (Type[] impls) {
        if (impls != null && impls.length > 0) {
            out.print("implements ");
            int i = 0;
            for (Type impl : impls) {
                if (i++ > 0) out.print(",");
                printType(impl);
            }
            out.println();
        }
    }
    public static void printTypeParameters (TypeVariable<?>[] vars) {
        if (vars != null && vars.length > 0) {
            out.print("<");
            int i = 0;
            for (TypeVariable<?> var : vars) {
                if (i++ > 0) out.print(",");
                out.print(var.getName());
                printBounds(var.getBounds());
            }
            out.print(">");
        }
    }
    public static void printBounds (Type[] bounds) {
        if (bounds != null && bounds.length > 0
            && !(bounds.length==1 && bounds[0]==Object.class)) {
            out.print(" extends ");
            int i = 0;
            for (Type bound : bounds) {
                if (i++ > 0) out.print("&");
                printType(bound);
            }
        }
    }
```

Example 6.6 Example 6.5, continued.

```
public static void printParams (Type[] types) {
    if (types != null && types.length > 0) {
        out.print("<");
        int i = 0;
        for (Type type : types) {
            if (i++ > 0) out.print(",");
            printType(type);
        }
        out.print(">");
    }
}
public static void printType (Type type) {
    if (type instanceof Class) {
        Class<?> c = (Class)type;
        out.print(c.getName());
    } else if (type instanceof ParameterizedType) {
        ParameterizedType p = (ParameterizedType)type;
        Class c = (Class)p.getRawType();
        Type o = p.getOwnerType();
        if (o != null) { printType(o); out.print("."); }
        out.print(c.getName());
        printParams(p.getActualTypeArguments());
    } else if (type instanceof TypeVariable<?>) {
        TypeVariable<?> v = (TypeVariable<?>)type;
        out.print(v.getName());
    } else if (type instanceof GenericArrayType) {
        GenericArrayType a = (GenericArrayType)type;
        printType(a.getGenericComponentType());
        out.print("[]");
    } else if (type instanceof WildcardType) {
        WildcardType w = (WildcardType)type;
        Type[] upper = w.getUpperBounds();
        Type[] lower = w.getLowerBounds();
        if (upper.length==1 && lower.length==0) {
            out.print("? extends ");
            printType(upper[0]);
        } else if (upper.length==0 && lower.length==1) {
            out.print("? super ");
            printType(lower[0]);
        } else assert false;
    }
}
```

Example 6.7 Example 6.5, continued.

```
    public static void printClass (Class c) {
        out.print("class ");
        out.print(c.getName());
        printTypeParameters(c.getTypeParameters());
        out.println();
        printSuperclass(c.getGenericSuperclass());
        printInterfaces(c.getGenericInterfaces());
    }
    public static void main (String[] args) throws ClassNotFoundException {
        for (String name : args) {
            Class<?> c = Class.forName(name);
            printClass(c);
        }
    }
}
```

Part II

Collections in Java 5

Part I presented the features of Java 5 generics, often using generified collection classes as illustration. Collections aren't just a convenient way to illustrate how generics can be used, though—they are a primary motivation for having generics in the language at all. Virtually all working Java programs make heavy use of collections so it is a big benefit that, with the Java 5 generified collections, code that uses collections can be safe from type errors at runtime and also be clearer and more transparent to read. But the changes that make this possible take some work to understand and, together with the new interfaces and implementations and other changes in Java 5, justify reviewing all the features of the Java Collections Framework as a whole.

The Java Collections Framework (which we will often refer to as the Collections Framework, or just the Framework) is a set of interfaces and classes in the packages `java.util` and `java.util.concurrent`. They provide client programs with various models of how to organise their objects, and various implementations of each model. These models are sometimes called *abstract datatypes*, and we need them because different programs need different ways of organising their objects. In one situation you might want to organise your program's objects in a sequential list, because their ordering is important and there are duplicates. In another a set might be the right datatype, because now ordering is unimportant and you want to discard the duplicates. These two datatypes (and others) are represented by different interfaces in the Collections Framework, and we will look at examples of using them in this chapter. But that's not all; none of these datatypes has a single "best" implementation—that is, one implementation that is better than all the others for all the operations. For example, a linked list may be better than an array implementation of lists for inserting and removing elements from the middle, but much worse for random access. So choosing the right implementation for your program involves knowing how it will be used as well as what is available.

This chapter starts with an overview of the Framework, then looks in detail at each of the main interfaces, the standard implementations, and various special-purpose modifications of these. Finally we will look at the generic algorithms in the `Collections` class

Chapter 7

The Main Interfaces of the Java Collections Framework

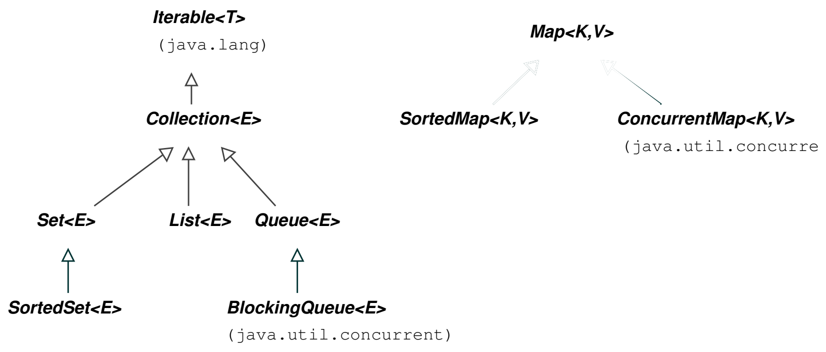


Figure 7.1: The main interfaces of the Java Collections Framework (extracted from back endpaper)

Figure 7.1 shows the main interfaces of the Java Collections Framework, together with one other—`Iterable`—which is outside the Framework but is an essential adjunct to it. It has the following purpose:

- `Iterable` defines the contract that a class has to fulfil for its instances to be usable with the *foreach* statement

and the Framework interfaces have the following purposes:

- `Collection` contains the core functionality required of any non-keyed collection. It has no direct concrete implementations; the concrete collection classes all implement one of its subinterfaces as well
- `Set` is a collection in which order is not significant, without duplicates. `SortedSet` automatically sorts its elements and returns them in order

- `Queue` is a collection providing first-in-first-out element access. Threads calling methods on the `BlockingQueue` interface can be blocked until the requested operation can be carried out
- `List` is a collection in which order is significant, accomodating duplicate elements
- `Map` is a collection which uses key-value associations to store and retrieve elements. In addition, `SortedMap` guarantees to return its values in ascending key order and `ConcurrentMap` provides methods that will be performed atomically

Chapters 9 to 13 will concentrate on each of the Collection Framework interfaces in turn. First, though, in chapter 8 we need to cover some preliminary ideas which run through the entire Framework design.

Chapter 8

Preliminaries

In this chapter we will take time to discuss the concepts underlying the framework before we get into the detail of the collections themselves.

8.1 Iterable and Iterators

An iterator is an object that implements the interface `Iterator` (Figure 8.1). The purpose of iterators

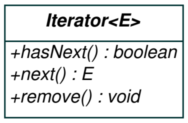


Figure 8.1: `Iterator`

is to provide a uniform way of accessing collection elements sequentially. So whatever kind of collection you are dealing with, and however it is implemented, you always know how to process its elements in turn. For example, the following code for printing the string representation of a collection's contents will always work:

```
// coll refers to an object which implements Collection
// ----- not the preferred idiom in JDK1.5! -----------
Iterator itr = coll.iterator() ;
while( it.hasNext() ) {
  System.out.println( itr.next() );
}
```

This works because any class implementing `Collection` will have an `iterator` method which returns an iterator appropriate to objects of that class. It's no longer the approved idiom because Java 5 introduced something better: the *foreach* statement which you met in Part I. Using *foreach*, we can write the code above more concisely

```
for( Object o : coll ) {
  System.out.println( o );
}
```

and, what's more, this will work with anything that can give an iterator, not just collections. Formally, that means any object which implements the interface `Iterable` (Figure 8.2). Since in Java 5 the

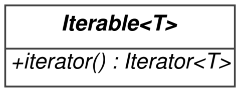


Figure 8.2: `Iterable`

`Collection` interface has been made to extend `Iterable`, any set, list or queue can be the target of *foreach*, as can arrays. If you write your own implementation of `Iterable`, that too can be used with *foreach*. Example 8.1 shows just about the simplest possible example of how `Iterable` can be directly implemented. A `Counter` object is initialised with a count of `Integer` objects; its iterator returns these in ascending order in response to calls of `next()`.

Example 8.1 Directly implementing `Iterable`

```
class Counter implements Iterable<Integer> {
  private int count;
  public Counter( int count ) { this.count = count; }
  public Iterator<Integer> iterator() {
    return new Iterator<Integer>() {
      private int i = 0;
      public boolean hasNext() { return i < count; }
      public Integer next() { i++; return i; }
      public void remove(){ throw new UnsupportedOperationException(); }
    };
  }
}
```

Now `Counter` objects can be the target of a *foreach* statement:

```
int total = 0;
for( int i : new Counter(3) ) {
   total += i;
}
assert total == 6;
```

In practice, it is unusual to implement `Iterable` directly in this way, as *foreach* is most commonly used with arrays and the standard collections classes.

8.2 Comparable and Comparator

One common requirement of collections is that they should be able to sort their contents according to some defined order. So a mechanism is needed for defining sort order for objects of any given class. In fact Java provides two mechanisms: in the first, a class can define a "natural order", which an object of that class can use to compare itself with another. A collection can rely on the natural order or it can impose its own, overriding the default natural order.

To establish a natural order, a class implements the interface `Comparable` (Figure 8.3). According to

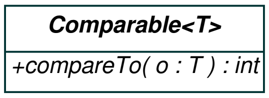


Figure 8.3: `Comparable`

the contract for `compareTo` the result should be a negative integer, zero, or a positive integer according to whether the supplied object is greater than, equal to, or less than the object making the comparison. Objects belonging to classes which implement `Comparable`—including many core classes such as String and the wrapper classes—can be sorted "automatically" by a collection or, as in this case, a utility method:

```
Integer[] ai = { 3, 2, 1, 4 };
Arrays.sort( ai );
assert Arrays.toString( ai ).equals( "[1, 2, 3, 4]" );
```

The alternative to allowing the set members to compare themselves is to provide an external means of comparing them. That means is usually an object implementing the interface `Comparator` (Figure 8.4). The `compare` method returns a negative integer, zero, or a positive integer according to whether its first

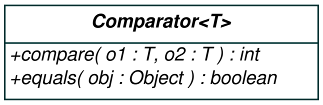

Comparator<T>
+compare(o1 : T, o2 : T) : int
+equals(obj : Object) : boolean

Figure 8.4: `Comparator`

argument is less than, equal to, or greater than its second. (The `equals` method is defined on this interface only so that its documentation can record that the motivation for overriding it is that clients may need to determine quickly if two different `Comparators` are equal—that is, if they impose the same order). Here is an example of declaring and using a `Comparator` that does not override `equals`:

```
Comparator<Integer> revComp =
  new Comparator<Integer>(){
      public int compare(Integer o1, Integer o2) {
          return (o1 < o2) ? 1 : (o1 > o2) ? -1 : 0;
      }
  };
Integer[] ai = { 3, 2, 1, 4 };
Arrays.sort( ai, revComp );
assert Arrays.toString( ai ).equals( "[4, 3, 2, 1]" );
```

The effect of the last example can be achieved more easily using the supplied method `reverseOrder` in the `Collections` class, which we shall look at in more detail later in this Part. The method `reverseOrder` has two overloads:

```
static <T> Comparator<T> reverseOrder()
static <T> Comparator<T> reverseOrder(Comparator<T> cmp)
```

The first of these returns a `Comparator` which, applied to two objects, returns the reverse of what the natural order would produce. The second returns a `Comparator` with the reverse effect of the one supplied to it. So both the following `assert` statements will hold:

```
assert Arrays.sort( ai, revComp ).equals(
            Arrays.sort( ai, Collections.reverseOrder() ));
Comparator natural = Collections.reverseOrder( revComp );
assert Arrays.sort( ai ).equals( Arrays.sort( ai, natural );
```

Besides their use in the generic methods of `Collections` and `Arrays`, `Comparable` and `Comparator` have an important role in the construction of the sorted collections `SortedSet` and `SortedMap`. These collections maintain their elements in order, as determined by their `Comparator` if one is supplied or otherwise by natural ordering. One aspect of this situation that may sometimes come as a surprise is that instead of using `equals` to determine equality between their elements, these collections rely on the `compare` or `compareTo` method they are using to return a value of 0 for equal elements. To avoid these (unpleasant) surprises, define `compare` or `compareTo` to be consistent with `equals`—to return 0 in exactly those cases that `equals` returns true. The discussion of `SortedSet` (Section 10.2) gives an example of what can happen otherwise.

8.3 Implementations

We have looked briefly at the interfaces of the Collections Framework, which define the behaviour that we can expect of each collection. But as we mentioned in the introduction to this chapter, there are several ways of implementing each of these interfaces. Why doesn't the Framework just use the best implementation for each interface? That would certainly make life simpler—too simple, in fact to be anything like life really is. If an implementation is a greyhound for some operations, Murphy's Law tells us that it will be a tortoise for others. Because there is no "best" implementation of any of the interfaces, you have to make a tradeoff— judging which operations are used most frequently in your application and choosing the implementation which optimises those operations.

The three main kinds of operations that most collection interfaces require are: insertion and removal of elements by position, retrieval of elements by content, and iteration over the collection elements. A glance at the back endpaper shows that there are many variations on these operations, but the main differences between the implementations can be discussed in terms of how they carry out these three. In this section we'll briefly survey the four main structures used as the basis of the implementations, and later when we need each one we will look at it in more detail. The four kinds are

- Arrays: these are the structures familiar from the Java language—and just about every other programming language since Fortran. Because arrays are implemented directly in hardware, they have the properties of random-access memory: very fast for accessing elements by position and for iterating over them, but slower for inserting and removing elements at arbitrary positions (because that may require adjusting the position of other elements). Arrays are used in the Collections Framework as the backing structure for `ArrayList` and `CopyOnWriteArrayList`, and in most of the `Queue` implementations. They also form an important part of the mechanism for implementing hash tables (see below).

- Linked lists: as the name implies, these consist of chains of linked cells. Each cell contains a reference to data and a reference to the next cell in the list (and, in some implementations, the previous cell also). Linked lists perform quite differently from arrays: accessing elements by position is slow, because you have to follow the reference chain from the start of the list, but insertion and removal operations can be performed in constant time by rearranging the cell references. Linked lists are the primary backing structure used for the class `LinkedList`, which implements both `Queue` and `List`, and are also used in implementing `HashSet` and `LinkedHashSet`

- Hash tables provide a way of storing elements indexed on their content rather than on an integer-valued index as with lists. In contrast to arrays and linked lists, hash tables provide no support for accessing elements by position, but access by content is usually very fast, as are insertion and removal. The first time that we will meet hash tables in use is in implementing the `Set` interface, in the class `HashSet` (Section 10.1.1).

- Trees also organise their elements by content, but with the important difference that they can store and retrieve them in sorted order. They are relatively fast for the operations of inserting and removing elements, accessing them by content, and iterating over them. We will first see trees in use as an implementation of the `SortedSet` interface (a subinterface of `Set`), in the class `TreeSet` (Section 10.2.1).

Besides allowing a focus on performance, another advantage to looking at implementations separately is that it will help you to understand the constructors of the different collection classes. We will see in Section 9.2 that there are two standard kinds of constructor which almost all the collection classes have. But many also have other constructors, which are quite similar across implementations regardless of the interface being implemented. Since these are essentially for the purpose of configuring the implementations, we will discuss them along with other aspects of the implementations.

8.4 Efficiency and the O-notation

In the last section, we talked about different implementations being "good" for different operations. A good algorithm is economical in its use of two resources: time and space. Of the two, we are usually more interested in how fast an algorithm is than we are in the amount of memory it uses. It's very hard to say precisely how quickly a program will execute, as that depends on many factors—including some that are outside the province of the programmer, such as the quality of the compiled code and the speed of the hardware. Even if we ignore these and limit ourselves to thinking only about how the execution time for an algorithm depends on its data, detailed analysis can be complex. A relatively simple example is provided in Knuth's classic book *Sorting and Searching* (Addison-Wesley, 1973), where the worst-case execution time for a multi-list insertion sort is given as

$$3.5N^2 + 24.5N + 4M + 2$$

where N is the number of elements being sorted and M is the number of lists.

As a shorthand way of describing algorithm efficiency, this isn't very convenient. Clearly we need a broader brush for general use. The one most commonly used is the *O-notation* (pronounced as the "big-oh notation"). The O-notation describes how an algorithm performs "in the limit", in other words when its data set becomes really large. In the expression above, the first two terms are comparable for low values of N; indeed, for $N < 8$ the second term is larger. But as N grows the first term increasingly dominates the expression, and by the time it reaches 100 the first term is 15 times as large as the second one. Using a very broad brush, we say that the worst-case for multi-list insertion sort takes time $O(N^2)$. We don't care too much about the coefficient because it doesn't make any difference to the single most important question we want to ask about any algorithm: what happens to the running time when the data size increases, say when it doubles? For the worst-case insertion sort, the answer is that the running time goes up fourfold. That makes $O(N^2)$ pretty bad, worse than most algorithms in common use and worse than any we will meet in practical use in this chapter.

Figure 8.5 shows some commonly-found running times, together with examples of algorithms to which they apply. For example, many other running times are possible, including some that are much worse than those in the table. Many important problems can only be solved by algorithms that take $O(2^N)$—for these, when N doubles the running time is squared! For all but the smallest data sets, such algorithms are infeasibly slow.

$O(...)$	*common name*	*effect on the running time if N is doubled*	*example algorithms*
1	constant	unchanged	insertion into a hash table (Section 10.1)
$\log N$	logarithmic	increases by a constant amount	insertion into a tree (Section 10.2.1)
N	linear	doubled	linear search
$N \log N$		doubled plus a constant amount	merge sort (Section 15.1)
N^2	quadratic	increases fourfold	bubble sort

Figure 8.5: Some Common Running Times

Sometimes we have to think about situations in which the cost of an operation varies with the state of the data structure; for example, the normally-inexpensive operation of adding an element to a tree will sometimes trigger the need for a relatively expensive tree rebalancing operation. In these situations we calculate the *amortized cost* of the operation—that is, the total cost of performing it n times (where n is large) divided by n.

8.5 Contracts

In reading about software design, you are likely to come across the term *contract*, often without any accompanying explanation. In fact, software engineering gives this term a meaning very close to what people usually understand a contract to be. In everyday usage, a contract defines what two parties can expect of each other, their obligations to each other, in some transaction. If a contract specifies the service that a supplier is offering to a client, the supplier's obligations are obvious. But the client too may have obligations—besides the obligation to pay—and failing to meet them will automatically release the supplier from her obligations too. For example, airlines' conditions of carriage—for our class of tickets anyway—release the airline from the obligation to carry a passenger who has failed to turn up in time. This allows the airline to plan their service on the assumption that all the passengers they are carrying were punctual; they do not have to incur extra work to accommodate clients who have not fulfilled their side of the contract.

Contracts work just the same way in software. If the contract for a method states preconditions on its arguments (*i.e.* the obligations that a client must fulfil) the method is only required to return its contracted results when those preconditions are fulfilled. For example, binary search (Section 15.3) is a fast algorithm to find a key within an ordered list, and it fails if you apply it to an unordered list. So the contract for `Collections.binarySearch` can say "if the list is unsorted, the results are undefined", and the implementer of binary search is free to write code which given an unordered list returns random results, throws an exception, or even enters an infinite loop. In practice, this situation is relatively rare in the contracts of the core API, because instead of restricting input validity they mostly allow for error states in the preconditions and specify the exceptions that the method must throw if it gets bad input. We think this is a practice to avoid, because it restricts the flexibility of the supplier unnecessarily. All that a client should need to know is how to keep to its side of the contract; if it fails to do that, all bets are off and there should be no need to say exactly what the supplier will do.

It's good practice in Java to code to an interface rather than to a particular implementation, so as to provide the maximum flexibility in choosing implementations. For that to work, what does it imply about the behaviour of implementations? If your client code uses methods of the `List` interface, for example, and at runtime the object doing the work is actually an `ArrayList`, you need to know that the assumptions you have made about how `Lists` behave are true for `ArrayLists` also. So a class implementing an interface has to fulfil all the obligations laid down by the terms of the interface contract. (Of course, a weaker form of these obligations is already imposed by the compiler; a class claiming to implement an interface must provide concrete method definitions matching the declarations in the interface; contracts take this further by specifying the behaviour of these methods as well).

The Collections Framework separates interface and implementation obligations in an unusual way. Some API methods return collections with restricted functionality—for example, the set of keys that you can obtain from a `Map` can have elements removed but not added (Chapter 13). Others can neither have elements added or removed (for example the list view returned by `Arrays.asList`), or may be completely read-only, like collections which have been wrapped in an unmodifiable wrapper (Section 14.2.1). To accomodate this variety of behaviours in the Framework without an explosion in the number of interfaces, the designers labelled the modification methods in the `Collection` interface (and in the `Iterator` and `ListIterator` interfaces as well) as *optional operations*. If a client tries to modify a collection using an optional operation which the collection does not implement, the method must throw `UnsupportedOperationException`. The drawback to this approach is that a client programmer can no longer rely on the contract for the interface, but has to know which implementation is being used and to consult the contract for that as well. That's such a serious drawback that you will probably never be justified in subverting interfaces in this way in your own designs.

8.6 Collections and Thread-Safety

A Java program when running is executing one or more execution streams, or *threads*. A thread is a like a lightweight process, so a program simultaneously executing several threads can be thought of like a computer running several program simultaneously, but with one important difference—different threads can simultaneously access the same memory locations and other system resources. On machines with multiple processors truly concurrent thread execution can be achieved by assigning a processor to each thread. If instead there are more threads than processors—the usual case—multithreading is implemented by *time-slicing*, in which a processor executes some instructions from each thread in turn before switching to the next one. Time-slicing is often fast enough to give the illusion that each thread has its own processor, and in fact this is the best way to imagine what is happening when you are programming for multiple threads.

There are two good reasons for using multithread programming: an obvious one, in the case of multiprocessor machines, is to share the work out and get it done quicker. (This reason will become even more compelling in the next few years as chip designers turn increasingly to architectures which depend on running multiple threads on even a single processor). The second reason is that many tasks can best be thought of as separate execution streams. A good example is the garbage collector. The JVM will run one instance of the garbage collector per CPU, with each instance getting its share of processor cycles via time-slicing. That's irrelevant detail, though, to the programmer writing a garbage collector—she should think of it as an independent autonomous process running in parallel with the other threads executing on the same JVM. Using that model will simplify her task enormously.

Inevitably, the conceptual simplification that threads can bring comes at a price. Different threads simultaneously accessing the same memory location can produce some very unexpected results, unless you take care to constrain their access. For a very simple example, consider a program which uses a stack, modelled by an array `arr` and an `int` variable `i` which points at the top element of the stack. For the program to work correctly, it should ensure that however many elements are added to or removed from the stack, `i` should always point at the top element. This is called an *invariant* of the program. If both `arr` and `i` are variables of the same class, it is an invariant of that class.

Now think about what can happen if two threads simultaneously attempt to add an element to the stack. Each will execute the same code, say

```
arr[i] = ... ;  //1
i++ ;           //2
```

which is correct in a single-threaded environment but may break the invariant in some multi-thread execution sequences. For example, if thread A executes line //1, thread B executes line //1 and then line //2, and finally thread A executes line //2, the overall result will have been to add one element to the stack, but to increment the stack pointer by two. The program is now in an inconsistent state, and is likely to fail because other parts of it will depend on the invariant being true. Code like this, that maintains program consistency when executed by a single thread but may break it in a multi-threaded environment, is not *thread-safe*. By contrast it would be possible to code this program so that once thread A had started to execute line //1, all other threads were excluded until it had successfully executed line //2. This is called *synchronizing* on a *critical section* of code. The synchronized code would be thread-safe, but at a price:

forcing threads to queue up to enter the critical section one at a time will slow down the overall execution of the program. Even in a single-threaded environment synchronization has a performance cost.

This tradeoff between performance and thread-safety has been viewed differently at different times during the development of the Java collection classes. In versions before Java 2 the only collection classes provided were `Vector`, `Hashtable` and their subclasses, now regarded as legacy classes to be avoided. They were written to be thread-safe (by synchronizing most of their methods) but the resulting poor performance led the Java 2 collection designers to avoid internal synchronization in most of the classes that now constitute the Framework and which we will be discussing in this chapter.

The exceptions to this rule are the classes in `java.util.concurrent` (Sections 11.3 and 13.4) and objects produced by the synchronization wrappers described in Chapter 14. Together with the legacy collections classes, these fall into a category sometimes described as *conditionally thread-safe* (Bloch, *Effective Java*): they are thread-safe by the definition above, but sequences of operations on them may still require external synchronization. For example, the operation of adding an element to a collection if it is currently absent can only be made atomic (and therefore consistently correct) by synchronizing the code sequence which first tests for its presence and then adds it if appropriate. Only one collection class, `ConcurrentHashMap` (Section 13.4), has atomically-executed methods like `putIfAbsent` which makes it (unconditionally) thread-safe.

8.6.1 Thread Safety and Iterators

If a method call breaks the conditions of the contract, the method is *entitled* to do anything. But to help the client programmer understand the problem, it would be *helpful* to fail in an indicative way. This is especially true when concurrency is involved, as otherwise failure may only become apparent some time after thread interference has occurred, making the problem difficult to diagnose. The non-thread-safe collection classes provide a helpful mechanism which can give clear and early warning of concurrent access. This mechanism is built into the iterators; if an iterator is described as *fail-fast* in the documentation, its methods will check that the collection has not been structurally changed (broadly speaking, that elements have been added or removed) since the last iterator method call. If it detects a change it will throw a `ConcurrentModificationException`. Of course, this mechanism cannot trap all concurrent access, but it will catch the most common cases.

You may have noticed that this description allows for “concurrent” modifications to be detected even in a single-threaded environment. Any structural changes to a collection during the lifetime of its iterator can result in a `ConcurrentModificationException`. This restriction rules out some sound programs (see, for example, Section 12.1), but many more unsound ones.

Chapter 9

The `Collection` Interface

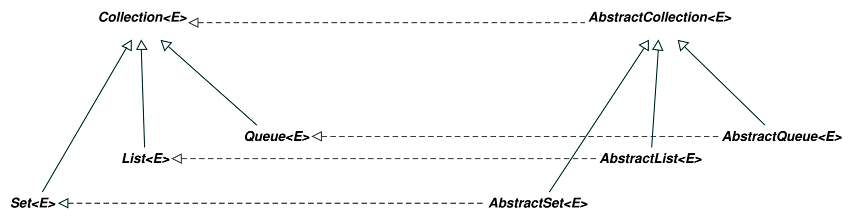


Figure 9.1: `Collection`

`Collection` (Figure 9.1) defines the core functionality that we expect of any non-keyed collection. It provides methods in four groups:

Adding elements:

```
boolean add( E o )                              // add the element o
boolean addAll( Collection<? extends E> c )     // add the contents of c
```

The boolean result returned by these methods indicates whether the collection was changed by the call. It can be false for collections, like sets, which will be unchanged if they are asked to add an element which is already present. But the method contracts specify that the elements being added must be present after execution, so if the collection refuses an element for any other reason (for example, some collections can't have `null` elements) these methods must throw an exception.

The signatures of these methods show that, as one might expect, you can only ever add elements or element collections of the parametric type.

Removing elements:

```
boolean remove( Object o )               // remove the element o
```

If the element `o` is null, `remove` removes a null from the collection if one is present. Otherwise if there is an element `e` present for which `o.equals(e)` it removes it. If not, it leaves the collection unchanged.

```
void clear()                          // remove all elements
boolean removeAll( Collection<?> c )   // remove the elements in c
boolean retainAll( Collection<?> c )   // remove the elements *not* in c
```

Where a method in this group returns a boolean function, it is true if the collection changed as a result of applying the operation.

In contrast to the methods for adding elements, these methods—and those of the next group—will accept elements or element collections of any type. We will explain this in a moment, when we look at examples of the use of these methods.

Querying the contents of a collection:

```
boolean contains(Object o)           // true if o is present
boolean containsAll(Collection<?> c) // true if all elements of c are present
boolean isEmpty()                    // true if any elements are present
int size()                           // returns the element count (or
                                     // Integer.MAX_VALUE if that is less)
```

Making a collection's contents available for further processing:

```
Iterator iterator()              // returns an Iterator over the elements
Object[] toArray()               // copies contents to an Object[]
<T> T[] toArray( T[] t )         // copies contents to a T[] (for any T)
```

The last two methods in this group convert collections into arrays. The first method will create a new array of `Object`, while the second takes an array of `T` and returns an array of the same type containing the elements of the collection. These methods are important because, although arrays should now be regarded as a legacy data type (see Section 16.3) many APIs, especially older ones which predate the Java Collections Framework, have methods with array type parameters or returns.

Seeing these methods for the first time, you might wonder why the "obvious" overload—one that returns an array of `E`—is missing. The answer lies in the Principle of Indecent Exposure, which we first saw in the discussion of generic arrays in Chapter ???. The Principle of Indecent Exposure forbids a class to expose an array of unreifiable type, and the declarations of `toArray` are designed to conform to it. The first overload is fine, because `Object` is certainly reifiable. The second is also fine—provided your program has everywhere else kept to the Principle. If so, then you will not have created (or received) arrays of non-reifiable type, and so `T` is guaranteed to be reifiable. The "obvious" overload, returning an array of type `E[]`, could not make that guarantee.

As a way of reinforcing the point, we could ask ourselves: if a method returning `E[]` existed, from where could it get the information about `E` that it would need to make the array? For that, `E` would have to be known at runtime, and erasure has removed it by then.

The array provided as argument to the second overload can have another purpose besides providing the type for the returned array. If there is room, the elements of the collection are placed in it, otherwise a new array of that type is created. The first case can be useful if you want to allow the `toArray` method to reuse an array that you supply; this can be more efficient, particularly if the method is being called repeatedly. The second case is more convenient—a common and straightforward idiom is to supply an array of zero length:

```
Collection<String> cs = ...
String[] sa = cs.toArray( new String[0] );
```

But why is *any* type allowed for T? One reason is to give the flexibility to allocate a more specific array type if the collection happens to contain elements of that type.

```
List<Object> l = Array.asList("zero","one");
String[] a = l.toArray(new String[0]);
```

Here, a list of objects happens to contain only strings, so it can be converted into a string array, in an operation analogous to the list promotion described in Section 5.2

If the list contains an object that is not a string, the error is caught at run-time rather than compile-time.

```
List<Object> l = Array.asList("zero","one",2);
String[] a = l.toArray(new String[0]);  // run-time error
```

Here the call raises ArrayStoreException, the error that occurs if you try to assign to an array with an incompatible reified type.

In general, one may want to copy a collection of a given type into an array of a more specific type (for instance, copying a list of objects into an array of strings, as above), or of a more general type (for instance, copying a list of strings into an array of object). It would never be the case that one would want to copy a collection of a given type into an array of a completely unrelated type (for instance, copying a list of integer into an array of string is always wrong). However, there is no way to specify this constraint in Java, so such errors are caught at run-time rather than compile-time.

One drawback of this design is that it does not work with arrays of primitive type.

```
List<Integer> l = Array.asList(0,1,2);
int[] a = l.toArray(new int[0]);  // compile-time error
```

This is illegal because the type parameter T in the method call must, as always, be a reference type. The call would work if we replace both occurrences of int by Integer, but often this will not do, as for performance or compatibility reasons we require an array of primitive type. In such cases, there is nothing for it but to copy the array explicitly.

```
List<Integer> l = Array.asList(0,1,2);
int[] a = new int[l.size()];
for (int i=0; i<l.size(); i++) a[i] = l.get(i);
```

The Collection Framework does not include convenience methods to convert collections to arrays of primitive type. Fortunately, this requires only a few lines of code.

9.1 Using the Methods of Collection

To illustrate the use of the collection classes, let's construct a tiny example. The two of us are forever trying to get organised; let's imagine that our latest effort involves writing our own to-do manager. A central class in this system will be the Task. Parts of the information that constitutes a Task are its priority and a String describing it. The natural ordering of Tasks, as defined by compareTo, is the natural ordering of their string representations. We have chosen string representations for the Task subclasses which mean that coding tasks will always be placed before telephoning tasks of the same priority—we are geeks, after all.

```
enum Priority  HIGH, MEDIUM, LOW ;
class Task implements Comparable<Task> {
  private Priority priority;
  protected String details;
  Task( String details, Priority priority ) {
    this.details = details;
    this.priority = priority;
  }
  public int compareTo( Task t ) {
    return toString().compareTo( t.toString() );
  }
  Priority getPriority() {
    return priority;
  }
  // definition of equals() and hashCode() omitted
}
```

We have divided the tasks to be scheduled into various subclasses:

```
class PhoneTask extends Task {
  String number;
  PhoneTask( String name, Priority priority, String number ) {
    super( name, priority );
    this.number = number;
  }
  public String toString() {
    return "ph. " + details;
  }
  // definition of equals() and hashCode() omitted
}
class CodingTask extends Task {
  CodingTask( String specReference, Priority priority ) {
    super( specReference, priority );
  }
  public String toString() {
    return "code " + details;
  }
  // definition of equals() and hashCode() omitted
}
```

Now we can define (Example 9.1) a series of tasks that to be carried out (even if in a real system they would be more likely to be retrieved from a database) For convenience, we're going to choose `ArrayList` as the implementation of `Collection` to use in this example, but we're not going to take advantage of any of the special properties of lists; we're treating `ArrayList` as an implementation of `Collection` and nothing more. As part of the retrieval process we can use the facilities provided by `Collection` to organise our tasks into various categories represented by lists, using the method `Arrays.asList` discussed in Section 1.4. The identifiers for these lists refer to the fact that the lists returned by `Arrays.asList` are unresizable:

Example 9.1 Example tasks for the Task Manager

```
PhoneTask nickPT = new PhoneTask( "Nick", Priority.MEDIUM, "987 6543" );
PhoneTask philPT = new PhoneTask( "Phil", Priority.HIGH, "123 4567" );
CodingTask blCT = new CodingTask( "bus. logic", Priority.MEDIUM );
CodingTask dbCT = new CodingTask( "db", Priority.HIGH );
CodingTask uiCT = new CodingTask( "ui", Priority.MEDIUM );
```

```
Collection<Task> urMondayTasks = Arrays.asList( philPT, dbCT );
Collection<Task> urTuesdayTasks = Arrays.asList( nickPT, blCT, uiCT );
Collection<PhoneTask> urPhoneTasks = Arrays.asList( nickPT, philPT );
Collection<CodingTask> urCodingTasks = Arrays.asList( blCT, dbCT, uiCT );
```

This isn't going to be quite good enough for our purposes, as these unresizable lists do not implement many of the optional operations of `Collection` (Section 8.5), and we want to use all the `Collection` operations on them. We have to relax our rule of not using any of the properties special to `ArrayList` in order make new resizable `Collections`.

```
Collection<Task> mondayTasks = new ArrayList<Task>( urMondayTasks );
Collection<Task> tuesdayTasks = new ArrayList<Task>( urTuesdayTasks );
Collection<PhoneTask> phoneTasks = new ArrayList<PhoneTask>( urPhoneTasks );
Collection<CodingTask> codingTasks = new ArrayList<CodingTask>( urCodingTasks );
```

Although the constructor we are using here belongs (like all constructors) to the class and not to the interface, we aren't assuming too much, because in fact amongst the concrete classes which implement sets, queues and lists, all have a constructor which takes an argument of type `Collection`.

Now we can use the methods of `Collection` to work with these categories. The following examples use the methods in the order in which they were presented above.

- adding elements — we can add new tasks to the schedule:

  ```
  mondayTasks.add( new PhoneTask( ... ) );
  ```

  or we can combine schedules together:

  ```
  Collection<Task> allTasks = new ArrayList<Task>( mondayTasks );
  allTasks.addAll( tuesdayTasks );
  ```

- removing elements— when a task is completed we can remove it from a schedule:

  ```
  boolean wasPresent = mondayTasks.remove( philPT );
  ```

  and we can clear a schedule out altogether because all its tasks have been done (yeah, right):

  ```
  mondayTasks.clear();
  ```

  The removal methods also allow us to combine entire collections in various ways. For example, to see which tasks other than phone calls are scheduled for Tuesday, we can write:

```
Collection<Task> tuesdayNonphoneTasks =
                        new ArrayList<Task>( tuesdayTasks );
tuesdayNonphoneTasks.removeAll( phoneTasks );
```

or to see which phone calls are scheduled for that day:

```
Collection<Task> tuesdayPhoneTasks =
                        new ArrayList<Task>( tuesdayTasks );
tuesdayPhoneTasks.retainAll( phoneTasks );
```

This last example can be approached differently to achieve the same result:

```
Collection<PhoneTask> phoneTuesdayTasks =
                        new ArrayList<PhoneTask>( phoneTasks );
phoneTuesdayTasks.retainAll( tuesdayTasks );
```

This example provides an explanation of the signatures of methods in this group and the next. Why do they take arguments of type `Object` or `Collection<?>` when the methods for adding to the collection restrict their arguments to its parametric type? Taking the example of `retainAll`, its contract requires the removal of those elements of this collection which do not occur in the argument collection. That gives no justification for restricting what the argument collection may contain—in the example above it can contain instances of any kind of `Task`, not just `PhoneTask`. And it is too narrow even to restrict the argument to collections of supertypes of the parametric type; we want the least restrictive type possible, which is `Collection<?>`. Similar reasoning applies to `remove`, `removeAll`, `contains` and `containsAll`.

- querying the contents of a collection—these methods allow us to check, for example, that the operations above have worked correctly. We are using `assert` here to make the system check our belief that we have programmed the previous operations correctly. For example the first statement will throw an `AssertionError` if we had not succeeded in adding `nickPT` to `tuesdayPhoneTasks`:

```
assert tuesdayPhoneTasks.contains( nickPT );
assert tuesdayTasks.containsAll( tuesdayPhoneTasks );
assert mondayTasks.isEmpty();
assert mondayTasks.size() == 0;
```

- making the collection contents available for further processing — the methods in this group provide an iterator over the collection or convert it to an array.

  Section 8.1 showed how the simplest—and most common—explicit use of iterators has been replaced in Java 5 by the *foreach* statement, which uses them implicitly. But there are uses of iteration with which *foreach* can't help. For example, supposing we want to merge two lists, of phone and of coding tasks, into a new list in which they are in their natural order. For little added effort Example 9.2 shows how we can generalise this task and merge any two collections, provided that iterators of each return their elements in their natural order.

Example 9.2 Merging collections using natural ordering

```
static <T extends Comparable<? super T>> List<T> mergeCollections(
      Collection<? extends T> c1, Collection<? extends T> c2 ) {
    List<T> mergedList = new ArrayList<T>();
    Iterator<? extends T> c1Itr = c1.iterator();
    Iterator<? extends T> c2Itr = c2.iterator();
    T c1Task = getNextTask( c1Itr );
    T c2Task = getNextTask( c2Itr );
    // each iteration will take a task from one of the iterators;
    // continue until neither iterator has any further tasks
    while( c1Task != null || c2Task != null ) {
      // use the current phone task if either the current coding
      // task is null or both are non-null and the phone task
      // precedes the coding task in the natural order
      boolean useC1Task = c2Task == null ||
        c1Task != null && c1Task.compareTo( c2Task ) < 0;
      if ( useC1Task ) {
        mergedList.add( c1Task );
        c1Task = getNextTask( c1Itr );
      } else {
        mergedList.add( c2Task );
        c2Task = getNextTask( c2Itr );
      }
    }
    return mergedList;
}
static <E> E getNextTask( Iterator<E> itr ) {
  return itr.hasNext() ? itr.next() : null ;
}
```

and we can check whether this works by:

```
assert mergeCollections( phoneTasks, codingTasks ).toString().equals (
  "[code bus. logic, code db, code ui, ph. Nick, ph. Phil]" );
```

9.2 Implementing `Collection`

The only direct implementation of the `Collection` interface in the Collections Framework is provided by the class `AbstractCollection`, which is the root of the hierarchies of sets, queues and lists. Example 9.3 shows code functionally equivalent to the standard `AbstractCollection` but favouring conciseness over efficiency to show the essence of what an implementation must provide.

Example 9.3 Outline implementation of `AbstractCollection`

```
public abstract class AbstractCollection<E> implements Collection<E> {
  protected AbstractCollection() {}
  public abstract Iterator<E> iterator();
  public abstract int size();
  public boolean isEmpty() {  return size() == 0; }
  public boolean contains( Object o ) {
    for ( E el : this ) {
      if ( o == null ? el == null : o.equals( el ) ) return true;
    }
    return false;
  }
  public Object[] toArray() { return toArray( new Object[size()] ); }
  public <T> T[] toArray(T[] a) {
    int size = size();
    if ( a.length < size ) {
      a = (T[])java.lang.reflect.Array.newInstance(
                            a.getClass().getComponentType(), size );
    }
    int i = 0;
    for ( E el : this ) a[i++] = (T)el;   // unchecked cast
    return a;
  }
  public boolean add( E o ) { throw new UnsupportedOperationException(); }
  public boolean remove( Object o ) {
    for( Iterator<E> e = iterator() ; e.hasNext() ; ) {
      E el = e.next();
      if ( o == null ? el == null : o.equals( el ) ) {
        e.remove();
        return true;
      }
    }
    return false;
  }
```

Example 9.4 Continuation of Example 9.3

```
  public boolean containsAll( Collection<?> c ) {
    boolean result = true;
    for ( Object el : c ) { result = result && contains( el ); }
    return result;
  }
  public boolean addAll( Collection<? extends E> c)  {
    boolean modified = false;
    for ( E el : c ) { modified = add( el ) || modified; }
    return modified;
  }
  public boolean removeAll(Collection<?> c) {
    boolean modified = false;
    for( Iterator<E> e = iterator() ; e.hasNext() ; ) {
      if (c.contains(e.next())) {
        e.remove();
        modified = true;
      }
    }
    return modified;
  }
  public boolean retainAll(Collection<?> c) {
    boolean modified = false;
    for( Iterator<E> e = iterator() ; e.hasNext() ; ) {
      if (!c.contains(e.next())) {
        e.remove();
        modified = true;
      }
    }
    return modified;
  }
  public void clear() {
    for( Iterator<E> e = iterator() ; e.hasNext() ; ) {
      e.next();
      e.remove();
    }
  }
}
```

The code in Example 9.3 implements all but two of the methods inherited from `Collection`. These are `iterator` and `size`, and all the other methods of `Collection` are implemented in terms of these two. The implementation of `add` that is provided, however, simply throws an `UnsupportedOperationException`, so for collections that are not to be read-only it must be overriden as well. Most of the methods of Example 9.3 are straightforward and require no comment; the exception is `toArray(T[] t)`. Recall that the argument `t` is an array which the method can use for one of two purposes: it can accomodate the collection elements if it is long enough; if not, it can provide the type information needed to create a new array which will be long enough. It is this second alternative which is implemented by the line

```
a = (T[])java.lang.reflect.Array.newInstance(
                   a.getClass().getComponentType(), size );
```

The method `newInstance` of `java.lang.reflect.Array` creates a new array given the component type and the number of elements required. Here the component type is being copied from the parameter, and the length of the new array is taken to be that of the collection to be stored in it. The casts in this method are unsafe, as you would expect given the discussion at the end of Section 9.1—any array can be provided as the argument to this method, so clearly there can be no compile-time guarantee of type-safety.

Note that this method omits one action of the standard implementation in the case that the supplied array is more than big enough to store the collection. In this situation the standard implementation places a `null` element immediately after the last stored collection element. This could be used as a sentinel to mark the end of the collection elements—but only, of course, if the method was being applied to a collection that does not accept `null` elements.

9.2.1 `AbstractCollection` subclasses

`AbstractCollection` is the root of the collections class hierarchy for sets, queues and lists. The relationship of `Collection` and `AbstractCollection` to `Set`, `Queue` and `List` and their abstract implementations is shown in Figure 9.2.

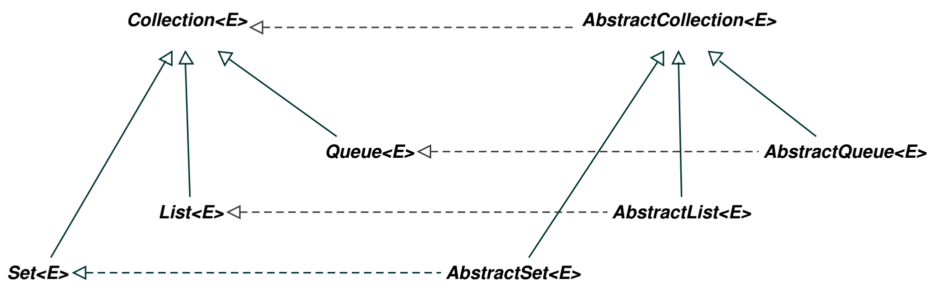


Figure 9.2: The place of `AbstractCollection` in the Java Collections Framework (extracted from the front endpaper)

Before we go on to look at these three main kinds of collection, which follow in the next three main sections, this is the moment for an explanation of the constructors that are available for `AbstractCollection` subclasses. The general rule for `Collection` implementations is that they have at least two constructors; taking as an example `HashSet` these are

```
public HashSet()
public HashSet( Collection<? extends E> c )
```

The first of these creates an empty set, and the second a set that will contain the elements of any collection of the parametric type—or one of its subtypes, of course: using this constructor has the same effect as creating an empty set with the default constructor, then adding the contents of a collection using `addAll`. Not quite all collection classes follow this rule (`ArrayBlockingQueue` cannot be created without fixing its capacity, and `SynchronousQueue` cannot hold any elements so no constructor of the second form is appropriate). In addition many collection classes have other constructors besides these two, but which ones they have depends not on the interface they implement but on the underlying implementation—these additional constructors are used to configure the implementation.

Chapter 10

Sets

A set is a collection of items which cannot contain duplicates—adding an item if it is already present in the set has no effect. The `Set` interface has the same methods as those of `Collection`, but is defined separately in order to allow the contract of `add` (and `addAll`, which is defined in terms of `add`) to be changed in this way. For example, suppose it is Monday and you have some free phone time, so that you have a chance to carry out all your telephone tasks. You can make the appropriate collection by adding all your telephone tasks to your Monday tasks. Using a set (again choosing a conveniently common implementation of `Set`) you can write

```
Set<Task> phoneAndMondayTasks = new HashSet<Task>( mondayTasks );
phoneAndMondayTasks.addAll( phoneTasks );
```

and this works because of the way that duplicate elements are handled. The task `nickPT`, which is in both `mondayTasks` and `phoneTasks`, only appears once in `phoneAndMondayTasks`, as intended—you definitely don't want to have to do all such tasks twice over!

10.1 Implementing `Set`

Section 9.1, which used the the methods of `Collection` in some examples, emphasised that they would work with any implementation of `Collection`. What if we had decided that we would use one of the `Set` implementations from the Collections Framework? We would have had to choose between the various concrete implementations that the Framework provides, which differ both in how fast they perform the basic operations of `add`, `contains` and iteration, and in the order in which their iterators return their elements. In this section we will look at these differences, then summarising the comparative performance of the different implementations in Section 10.3.

The only direct implementation of the `Set` interface in the Collections Framework is provided by the class `AbstractSet`, from which all the concrete implementations inherit. The contract for `AbstractSet` refines that of `AbstractCollection` in respect of what constitutes equality. Two sets are equal if and only if they contain the same elements—there is no dependence on ordering. Example 10.1 shows an implementation of `AbstractSet`.

The `equals` method of this class checks that the argument is itself a `Set` (a `Set` cannot be equal to any other kind of object), that the two `Sets` have the same size, and that one contains all the elements of

Example 10.1 Outline implementation of `AbstractSet`

```
public abstract class AbstractSet<E> extends AbstractCollection<E>
                                          implements Set<E>
    protected AbstractSet()
    public boolean equals(Object o)
      if (!(o instanceof Set)) return false;
      return ((Set)o).size() == size() && containsAll( (Set)o );

    public int hashCode()
      int h = 0;
      for( E el : this )
        if ( el != null ) h += el.hashCode();

      return h;
```

the other. Provided that `size` fulfils its contract of returning the number of distinct elements that a `Set` contains, this guarantees that equal `Sets` must contain the same elements.

Let's see some working code: Example 10.2 is a tiny implementation of `Set`. It provides implementations for the two methods — `size` and `iterator`—that are abstract in `AbstractSet`, and makes `add` into a supported operation, overriding the version inherited from `AbstractCollection`. It is based on arrays, in the same way as the collection implementations we looked at in Part I. For reasons discussed there, the unchecked cast at the creation of the array does not contravene the Principle of Indecent Exposure: although `array` may have a non-reified type, it is never exposed.

Example 10.2 Outline implementation of `Set`

```
class FixedArraySet<E> extends AbstractSet<E> {
  private final int MAX_SIZE = 10;
  private E[] array = (E[])new Object[MAX_SIZE];  // unchecked cast
  private int i = -1;
  public FixedArraySet() {}
  public FixedArraySet( Collection<? extends E> c ) { addAll(c); }
  public boolean add( E o ) {
    if ( ! contains( o ) ) {
      assert i < MAX_SIZE;
      array[i + 1] = o;
      i = i + 1;
      return true;
    } else {
      return false;
    }
  }
  public int size() { return i + 1 ; }
  public Iterator<E> iterator() {
    return new Iterator<E>() {
      private int j = -1;
      public boolean hasNext() { return j < i ; }
      public E next() { return array[++j]; }
      public void remove() {
        if ( j != i ) System.arraycopy( array, j + 1, array, j, i - j );
        i--; j--;
      }
    };
  }
}
```

This implementation is about the simplest you can imagine; being backed by a fixed array, it has a fixed maximum size. If assertions are enabled, attempting to add an element to a “full” `FixedArraySet` results in an `AssertionError`; otherwise an `ArrayIndexOutOfBoundsException` is thrown. This limitation makes `FixedArraySet` a toy implementation, as does its performance: `contains` has a complexity of $O(n)$ because it requires iteration through the set; since `add` calls `contains` its cost too is $O(n)$. These results are terrible compared to the standard implementations, as we shall see. The iteration operation itself, however, simply involves stepping through the backing array, so has complexity $O(1)$. This suggests that an array-backed implementation would in fact be a good choice for applications that depend on efficiently iterating through a set; `CopyOnWriteArraySet` is provided for just this reason.

There are five concrete implementations of `Set` in the Collections Framework. Figure 10.1 shows their relationship to `AbstractSet` and the `Set` subinterface `SortedSet`. In this section we will look at `HashSet`, `LinkedHashSet` and `CopyOnWriteArraySet`—`EnumSet` is specialised for use with enums and is discussed along with them in Section ???, and `SortedSet` is discussed together with its implementation `TreeSet` in Section 10.2. .

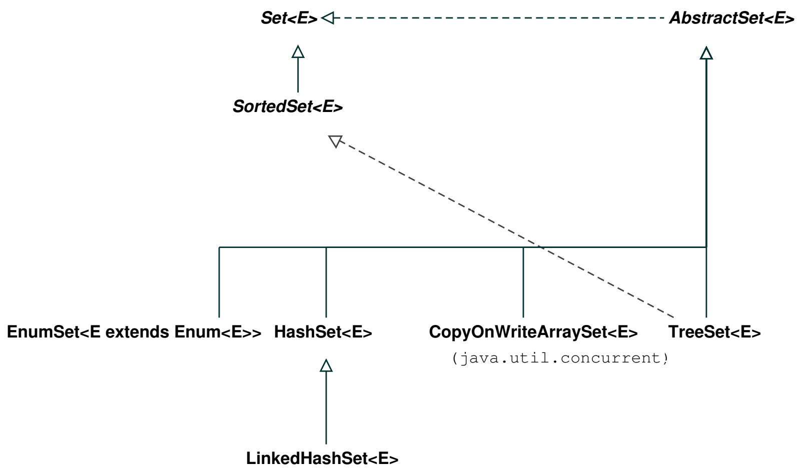


Figure 10.1: Implementations of the `Set` interface (extracted from the front endpaper)

10.1.1 `HashSet`

This class is the most commonly-used implementation of `Set`. As the name implies, it is implemented by a *hash table*, an array in which elements are stored at a position derived from their contents. Since hash tables store and retrieve elements by their content, they are well suited to implementing the operations of `Set` (the Collections Framework also uses them to for various implementations of `Map`). For example, to implement `contains(Object o)` you would want to look for the element `o` and return `true` if it were found.

The position of an element in a hash table is given by the *hash code* of its contents—a number calculated in a way that gives an even spread of results over the element values that might be stored. For example, the hash code of a string `str` is calculated as follows:

```
int hash = 0;
for ( char ch : str.toCharArray() ) {
  hash = hash * 31 + ch;
}
```

To get an index from the hash code we get its remainder after division by the table length (for historical reasons, the Collections Framework classes use bit-masking instead of division). Since that means it is the pattern of bits at the low end of the hash code that is significant, prime numbers (like 31, here) are used in calculating the hash code because multiplying by primes will not tend to shift information away from the low end—as would multiplying by a power of 2, for example.

It's obvious that the algorithm above will give equal hash values for equal strings—essential if the same hash function is to be used for storing words as for retrieving them. But can it also give equal hash values for unequal strings? That would be a problem because it would imply that the hash table would have to store more than one element at each location. A moment's thought will show that in fact you can't avoid this problem completely without having an infinitely large array to hold the table. We can minimise the problem with a good hash function, one which spreads the elements out equally in the table, but when collisions do occur we have to have a way of keeping the colliding elements at the same table location or *bucket*. This is done by storing them in a linked list, as shown in Figure 10.2[1]. We will look at linked lists in detail as one of the implementations of the `Queue` interface (Section 11.2) but for now it's enough to see that elements stored at the same bucket can still be accessed, at the cost of following a chain of cell references. In this example preserving the bottom three bits of the hash codes for the `Character` objects which box `'a'` and `'b'` give a bucket index of 2, while `'c'` gives an index of 1.

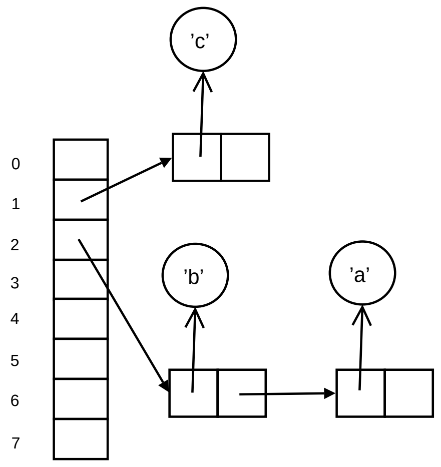


Figure 10.2: A hash table with chained overflow

As long as there are no collisions the cost of inserting or retrieving an element is constant. As the hash table fills, collisions become more likely; assuming a good hash function the probability of a collision in a lightly loaded table is proportional to its *load factor*, defined as the number of elements in the table divided by its capacity (the number of buckets). If a collision does take place, a linked list has to be created and subsequently traversed, adding an extra cost to insertion proportional to the number of elements in the list. As the load factor continues to increase the performance worsens until eventually, usually at some predetermined load factor, it becomes worthwhile to copy the elements to a new and larger hash table.

Iterating over a hash table requires each bucket to be examined to see whether it is occupied or not and therefore costs a time proportional to the capacity of the hash table plus the number of elements it contains. Since the iterator examines each bucket in turn, the order in which elements are returned depends on their hash codes. The table shown in Figure 10.2 yields its elements in order of descending table index and forward traversal of the linked lists:

```
Set s1 = new HashSet(8);
s1.add('a');
s1.add('b');
s1.add('c');
assert s1.toString().equals("[b, a, c]");
```

[1]This diagram is simplified from the real situation. Because `HashSet` is actually implemented by a specialised `HashMap`, each of the cells in the chain contains not one but two references, to a key and a value (see Section 13). Only the key is shown in this diagram.

Later in this section we will look at `LinkedHashSet`, a variant hash table implementation with an iterator which does return elements in their insertion order.

The chief attraction of a hash table implementation for sets is the constant-time performance (ideally) for the basic operations of `add`, `remove`, `contains` and `size`. Its main disadvantage is its iteration performance; since iterating through the table involves examining every bucket, its cost is proportional to the table size regardless of the size of the set it contains.

The Collections Framework class `HashSet` has some features which don't directly follow from this general description of how hash tables work. For one thing,`HashSet` delegates all its operations to a private instance of `HashMap` (Section 13.2) in which every set element is treated as a key with a corresponding standard value signifying that it is present. A second peculiarity follows from the first, because since JDK1.4 all instances of `HashMap` have a bucket count which is a power of two. So the constructors don't behave quite as you might expect with respect to the capacity. `HashSet` has the standard constructors that we introduced in Section 9.2.1, together with a further two:

```
HashSet( int initialCapacity )
HashSet( int initialCapacity, float loadFactor )
```

both of which create an empty set but allow some control over the size of the underlying table—creating one with a length of the next-largest power of 2 after the supplied capacity. The hash table-based implementations in the Collections Framework all have similar constructors.

10.1.2 `CopyOnWriteArraySet`

In functional terms `CopyOnWriteArraySet` is another straightforward implementation of the `Set` contract, but with quite different performance characteristics from `HashSet`. As the name implies, each instance of this class is essentially read-only; a change to the contents of the set results in an entirely new object being created. So `add` has complexity $O(n)$, as does `contains`, which has to be implemented by a linear search. Clearly you wouldn't use `CopyOnWriteArraySet` in a context where you were expecting many searches or insertions. But because this implementation is backed ultimately by an array, iteration over its elements is $O(1)$—faster than `HashSet`—and it has one advantage which is really compelling in some applications: it provides thread safety (Section 8.6) at no additional cost. Normally, of course, thread-safe implementations are much less efficient because a thread must get exclusive access to a collection object before it can use it in any way. But if, as in the case of `CopyOnWriteArraySet`, such change is actually impossible, then thread-safety comes for free!

When would you want to use a set with these characteristics? In fairly specialised cases; one that is quite common is in the implementation of the Observer-Observable design pattern, which requires events to be notified to a set of observers. With other set implementations an awkward complication arises if this set is changed by another thread during the notification process. By making a new copy of the array, then modifying and returning it, `CopyOnWriteArraySet` manages to avoids this complication at no extra cost for read accesses.

Since there are no configuration parameters for a `CopyOnWriteArraySet`, the constructors are just the standard ones of Section 9.2.1.

10.1.3 `LinkedHashSet`

This class inherits from `HashSet`, still implementing `Set` and refining the contract of its superclass in only one respect: it guarantees that its iterators will return their elements in the order in which they were first added. It does this by maintaining a linked list of the set elements, as shown by the curved arrows in Figure 10.3. The resulting behaviour is just what you would expect:

```
Set s2 = new LinkedHashSet(8) {{ add('a'); add('b'); add('c'); }};  //1
assert s2.toString().equals("[a, b, c]");
```

The code at line `//1` creates an anonymous local subclass of `HashSet` with an initializer which, on the creation of an object of the class, adds three elements to it. It's a concise (if somewhat tricky) way of simultaneously creating and initializing an object.

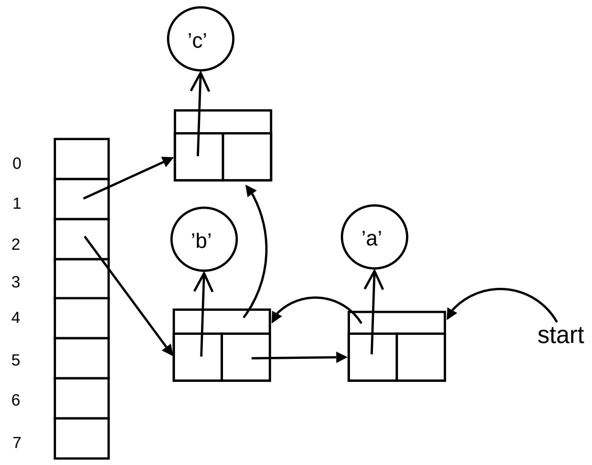


Figure 10.3: A linked hash table

The linked structure also has a useful consequence in terms of improved performance: `next` performs in constant time, as the linked list can be used to visit each element in turn. This is in contrast to `HashSet`, for which every bucket in the hash table must be visited whether it is occupied or not, but the overhead involved in maintaining the linked list means that you would only choose `LinkedHashSet` in preference to `HashSet` if the efficiency or the order of iteration were important for your application.

The constructors for `LinkedHashSet` provides the same facilities as those of `HashSet` for configuring the underlying hash table.

10.2 `SortedSet`

`Set` has one subinterface, `SortedSet` (see back endpaper) which adds to its contract a guarantee that its iterator will traverse the set in ascending element order. We discussed the different ways of defining element ordering earlier in this chapter: it can either be defined by the element class itself, if that implements `Comparable`, or it can be imposed by an external `Comparator`. `Task` implements `Comparable` (its natural ordering is the natural ordering of its string representation), so can be sorted "automatically". Merging two ordered lists (Example 9.2, page 125) which was quite tricky using parallel iterators, is trivial if we use a `SortedSet` to do the work:

```
// TreeSet is the standard implementation of SortedSet
Set<Task> naturallyOrderedTasks = new TreeSet<Task>( phoneTasks );
naturallyOrderedTasks.addAll( codingTasks );
assert naturallyOrderedTasks.toString().equals (
  "[code bus. logic, code db, code ui, phone Nick, phone Phil]" );
```

The alternative to using the natural ordering is to supply the `SortedSet` with a `Comparator` at construction time. For example, another useful ordering for tasks would be in descending order of priority:

```
Comparator<Task> priorityComparator =
  new Comparator<Task>(){
    public int compare(Task o1, Task o2) {
      int p = o1.getPriority().compareTo( o2.getPriority() );
      return p != 0 ? p : o1.toString().compareTo( o2.toString() );
    }
  };
SortedSet<Task> priorityOrderedTasks =
    new TreeSet<Task>( priorityComparator );
priorityOrderedTasks.addAll( phoneTasks );
priorityOrderedTasks.addAll( codingTasks );
assert priorityOrderedTasks.toString().equals (
  "[code db, ph. Phil, code bus. logic, code ui, ph. Nick]" );
```

Could you not simply compare the priorities of the tasks, without using the string representation as a secondary key? You could, but you might not like the result. Recall that in the discussion of `Comparable` and `Comparator` (Section 8.2) we mentioned that the methods of these interfaces should be consistent with `equals`. The reason for this is that the contract for `SortedSet` (and, as we shall see later, `SortedMap` also) states that it will use the `compareTo` method of its `Comparator`—or, if it does not have one, the `compare` method of its elements—instead of the elements' `equals` method. This means that if two elements are equal in their ordering, the set will treat them as duplicates and one will be discarded.

The extra methods defined by the `SortedSet` interface fall into three groups:

Retrieving the comparator:

```
Comparator<? super E> comparator()
```

This method returns the set's comparator if it has been given one rather than relying on its elements to define their ordering. The type `Comparator<? super E>` is used because a `SortedSet` parameterized on `E` can rely for ordering on a `Comparator` defined on any supertype of `E`. For example, a `Comparator<Number>` could be used with a `SortedSet<Integer>`.

Getting the head and tail elements:

```
E first()
E last()
```

If the set is empty, these operations throw a `NoSuchElementException`.

Finding subsequences:

```
SortedSet<E> subset( E fromElement, E toElement )
SortedSet<E> headSet( E toElement )
SortedSet<E> subset( E fromElement )
```

The arguments to these operations do not themselves have to be members of the set. The sets returned are *half-open intervals*: they are inclusive of the `fromElement`—provided it really is a set member, of course—and exclusive of the `toElement`.

We could use these methods, for example, to obtain a view of the portion of `priorityOrderedSet` corresponding to the high-priority items by using `headSet`, providing as the argument the element which we wish to form the exclusive limit of the set — a medium-priority task defined so that it will come before any other in the secondary (description) ordering:

```
Task highestMediumPriorityTask = new Task( "\0", Priority.MEDIUM );
SortedSet<Task> highPriorityTasks =
    priorityOrderedTasks.headSet( highestMediumPriorityTask );
```

Notice that the sets returned by these operations are not newly created sets but different views of the original `SortedSet`. So changes in them will be reflected in the original `SortedSet`, and vice versa.

10.2.1 `TreeSet`

We have already met this class in discussing the interface `SortedSet`, for which it is the standard implementation and which completely specifies its contract. This is the first tree implementation that we have seen, so we should take a little time now to consider how trees perform in comparison to the other implementation types used by the Collections Framework.

Trees are the data structure you would choose for an application which needed fast insertion and retrieval of individual elements but which also required that they should be held in sorted order. For example, suppose you wanted to match all the words from a set against a given prefix, a common requirement in visual applications where a dropdown should ideally show all the possible elements that match against the prefix that the user has typed. A hash table can't return its elements in sorted order and a list can't retrieve its elements quickly by their content, but a tree can do both.

In computing, a tree is a branching structure that represents hierarchy. Computing trees borrow a lot of their terminology from genealogical trees, though there are some differences; the most important is that in computing trees each node has only one parent (except the root, which has none). An important class of tree often used in computing is a *binary* tree—one in which each node can have at most two children. Figure 10.4 shows an example of a binary tree containing the words of this sentence in alphabetical order.

The most important property of this tree can be seen if you look at any non-leaf node, say the one containing the word "the": all the nodes below that on the left contain words that precede "the" alphabetically, and all those on the right words that follow it. To locate a word you would start at the root and descend level by level doing an alphabetic comparison at each level, so the cost of retrieving or inserting an element is proportional to the depth of the tree.

How deep then is a tree that contains n elements? The complete binary tree with two levels has 3 elements (that's $2^2 - 1$), the one with three levels has 7 elements ($2^3 - 1$) and so on. In general a binary tree with n complete levels will have $2^n - 1$ elements. To work backwards from the number of elements to the number of levels you have to use a *log* function—a function with the property that $2^{\log(n)} = n$. All that you really need to know about log functions is that just as n grows much more slowly than 2^n, $\log(n)$ grows much more slowly than n. So `contains` on a large tree is much faster than on a list containing

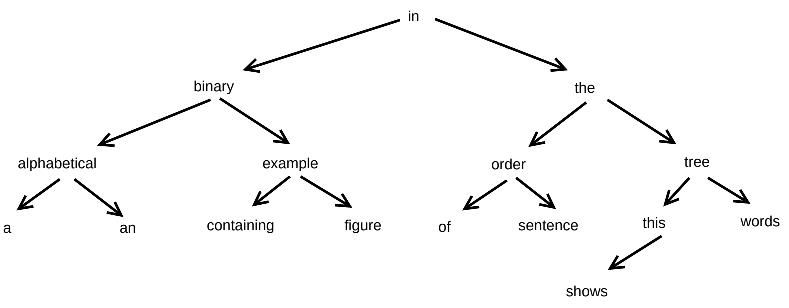


Figure 10.4: An ordered binary tree

the same elements. It's still not as good as on a hash table—whose operations can ideally work in constant time—but a tree has the big advantage over a hash table that its iterator can return its elements in sorted order.

Not all binary trees will have this nice performance, though. Figure 10.4 shows a *balanced* binary tree—one in which each node has an equal number of descendants (or as near as possible) on each side. An unbalanced tree can give much worse performance—in the worst case, as bad as a linked list (see Figure 10.5). `TreeSet` uses a data type called a *red-black tree*, which has the advantage that if it becomes unbalanced through insertion or removal of an element, it can always be rebalanced in $O(\log(n))$ time.

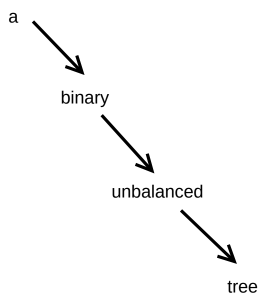


Figure 10.5: An unbalanced binary tree

The constructors for `TreeSet` include, beside the standard ones, one which allows you to supply a `Comparator` (Section 10) and one which allows you to create one from another `SortedSet`, using both the same comparator.and the same elements.

10.3 Comparison of `Set` Implementations

	`add`	`contains`	`next`	notes
`HashSet`	$O(1)$	$O(1)$	$O(n/h)$	h is the table capacity
`TreeSet`	$O(\log n)$	$O(\log n)$	$O(1)$	traversal in sorted order
`CopyOnWriteArraySet`	$O(n)$	$O(n)$	$O(1)$	thread-safe
`LinkedHashSet`	$O(1)$	$O(1)$	$O(1)$	traversal in inserted order

Chapter 11

Queues

A Queue (Figure 11.1) is a collection designed to hold elements to be processed, in the order in which processing is to take place. Different implementations of the interface embody different rules about what this order should be. Many of the implementations use the rule that tasks are to be processed in the order in which they were submitted (*First In First Out*, or *FIFO*), but other rules are possible—for example, the Collections Framework includes queue classes whose processing order is based on task priority. The

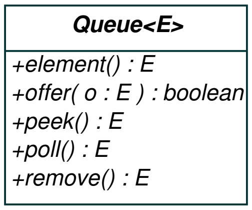


Figure 11.1: Queue

Queue interface was introduced in Java 5, mainly motivated by the need for queues in the concurrency utilities included in that release. A glance at the hierarchy of implementations shown in the front endpaper figure shows that in fact nearly all of the Queue implementations in the Collections Framework are in the package java.util.concurrent.

One classic requirement for queues in concurrent systems comes when a number of tasks have to be executed by a number of threads working in parallel. An everyday example of this situation is that of a single queue of airline passengers being handled by a line of check-in operators. Each operator works on processing a single passenger (or a group of passengers) while the remaining passengers wait in the queue. When a passenger reaches the head of the queue she waits to be assigned to the next operator to become free. There is a good deal of fine detail involved in implementing a queue like this; operators have to be prevented from simultaneously attempting to process the same passenger, empty queues have to be handled correctly, and in computer systems there has to be a way of defining queues with a maximum size, or *bound*. (This last requirement may not be at all desirable in airline terminals, but it can be very useful in systems in which there is a maximum reasonable waiting time for a task to be executed). The Queue implementations in `java.util.concurrent` look after these implementation details for you.

In addition to the operations inherited from `Collection`, `Queue` includes operations for:

Adding an element to a queue, if possible:

```
boolean offer(E o)  // inserts the given element if possible
```

This operation may routinely fail, for example because the queue has reached its capacity. `offer` accommodates this failure by returning a boolean reporting whether the element was successfully inserted (in contrast to the `add` method of `Collection`, which can report failure only by throwing an exception).

Retrieving an element from a queue: The four methods provided for this purpose differ in two respects: whether they only provide a copy or also remove the element, and whether they return null or throw an exception in the case of an empty queue.

The methods which throw an exception for an empty queue are

```
E element()  // retrieves but does not remove the head element
E remove()   // retrieves and removes the head element
```

Notice that this is a different overload from the `Collection` method `remove(Object)`. The methods which return `null` for an empty queue are

```
E peek()     // retrieves but does not remove the head element
E poll()     // retrieves and removes the head element
```

Because of the use of `null` as a return value it is inadvisable to use it as a queue element; for this reason, most implementations do not allow it.

11.1 Using the Methods of `Queue`

Let's look at examples of the use of these methods. For us, the major goal of having a to-do manager is to tell us what we should do next—we find it altogether too much of a strain to work this out for ourselves. Queues are a good mechanism for storing this information. In simple cases, we may just decide that the best policy is to do tasks in the order in which we become aware of the need for them. So new tasks should be added to the tail of the queue. For these examples we have chosen the `ConcurrentLinkedQueue`, which despite its unwieldy name is the most straightforward implementation.

```
Queue<Task> taskQueue = new ConcurrentLinkedQueue<Task>();
taskQueue.offer( nickPT );
taskQueue.offer( philPT );
```

Any time we feel ready to do a task we can take the one that has reached the head.

```
Task nextTask = taskQueue.poll();
if ( nextTask != null ) {
  // process nextTask
}
```

The choice between using `poll` and `remove` depends on whether we want to regard queue emptiness as an exceptional condition. Realistically—given the nature of this application—that might be a sensible assumption, so this could be preferable:

```
try {
  Task nextTask = taskQueue.remove();
  // process nextTask
} catch ( NoSuchElementException e ) {
  // but we *never* run out of tasks!
}
```

This scheme needs some refinement to allow for the nature of different kinds of tasks. Phone tasks can be fitted into relatively short time slots, whereas we don't like to start coding unless there is reasonably substantial time to get into the task. So if time is limited, say before the next meeting, we might like to check that the next task is of the right kind before we take it off the queue:

```
try {
  Task nextTask = taskQueue.element();
  if ( nextTask instanceof PhoneTask ) {
    // process nextTask
  }
} catch ( NoSuchElementException e ) { ... }
```

But notice that the `Queue` methods can't do much for us if the head element is not one that we want. Of course, the methods of `Collection` are still available, so we can get an iterator, but not all queue iterators guarantee to return their elements in proper sequence. So in this situation, if our to-do manager is entirely queue-based, it may be able to offer us no better alternative than to go for coffee until the meeting starts. We will see that the `List` interface can provide better facilities in this respect.

11.2 Implementing `Queue`

The concrete Collections Framework implementations of `Queue`, shown in Figure 11.2, differ widely in their behaviour. They are mostly, but not all, synchronized for use in multithreading programs; most are *blocking* queues, for which a subinterface `BlockingQueue` is provided; some support priority ordering; one—`DelayQueue`—holds elements until their delay has expired, and one—`SynchronousQueue`—is purely a synchronisation facility. In choosing between `Queue` implementations you would be influenced more by these functional differences than by their relative performances. Besides, a discussion of the efficiency of the concurrent implementations is outside the scope of this book.

11.2.1 `LinkedList`

The first class that stands out in Figure 11.2 is `LinkedList`. This isn't specialised at all: it isn't thread-safe, nor does it support blocking or priorities. In fact, it was in the Collections Framework for a long time before queues were introduced, as one of the standard implementations of `List` (Section 12.2). It was retrofitted in Java 5 to implement `Queue` as well. In its simplest form, a linked list consists of a number of cells each with two fields, one to hold an element reference and one to allow an iterator to reach the next cell. Each cell object would be an instance of a class defined like this:

```
class Cell<E> {
  private E element;
  private Cell next;
  ...
}
```

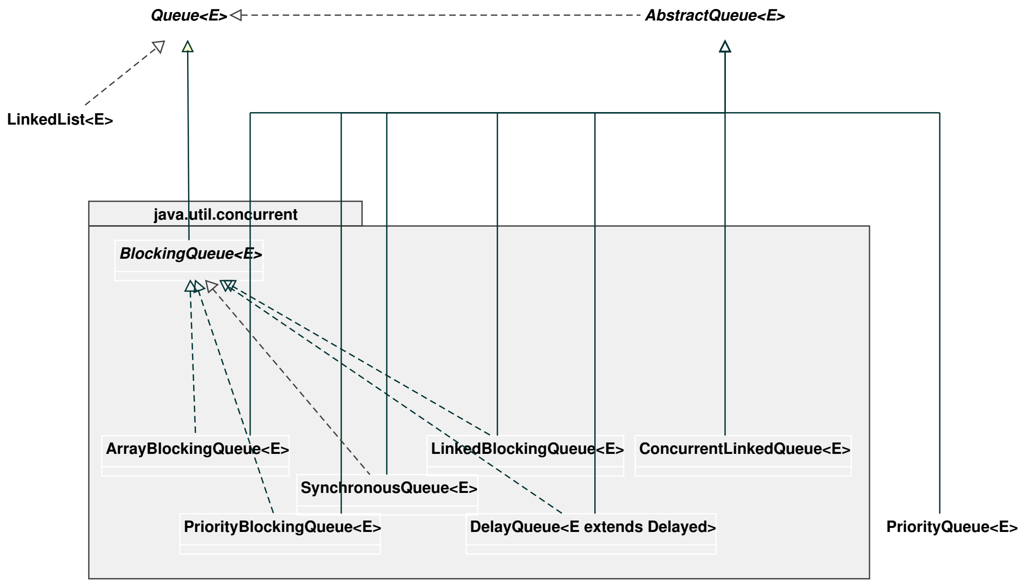


Figure 11.2: Implementations of `Queue` in the Collections Framework (extracted from front endpaper)

Figure 11.3(a) shows a simplified picture of a `LinkedList` after the following statement:

```
new LinkedList<Character>(){{ add( nickPT ); add( philPT ); }};
```

Removing an element from the list or inserting one into it is done by rearranging the cell references. For example, Figure 11.3(b) shows the effect of calling `remove` on this queue. A `LinkedList` also maintains a pointer to the last element of the queue, so that `add` and `offer` can be implemented by a similar rearrangement. One of the attractions of linked lists is that the insertion and removal operations implemented by these pointer rearrangements perform in constant time; by contrast, traversing a linked list is relatively slow because of the need to follow pointers from each cell to the next.

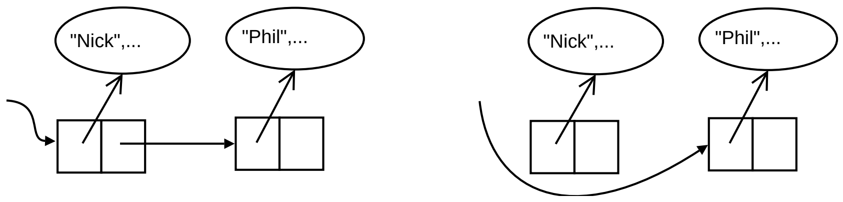


Figure 11.3: Removing an element from a `LinkedList`

You would use `LinkedList` as a `Queue` implementation in situations where thread safety isn't an issue, you don't require blocking behaviour from the queue, and where you are prepared to ensure either that your program never adds `null` elements to the queue, or else that it never uses `peek` or `poll` to determine queue emptiness (since these methods return `null` for an empty queue). The constructors for `LinkedList` are just the standard ones of Section 9.2.1.

11.2.2 AbstractQueue

Following the pattern of Collection and Set, all the other Queue implementations inherit from a skeketal AbstractQueue. The key difference between AbstractQueue and LinkedList is that AbstractQueue is written on the assumption that a subclass implementation of poll or peek will return null only if the list is empty. In a LinkedList, on the other hand, null elements are allowed.

Code functionally equivalent to the standard implementation of AbstractQueue is shown in Example 11.1. Concrete subclasses must implement the Collection methods size and iterator, together with the Queue methods offer, peek and poll.

Example 11.1 Outline implementation of AbstractQueue

```
public abstract class AbstractQueue<E>
      extends AbstractCollection<E> implements Queue<E>
  protected AbstractQueue()
  public boolean add(E o)
      if( o == null ) throw new NullPointerException();
      if( ! offer(o) ) throw new IllegalStateException("Queue full");
      return true;

  public E remove()
      E x = poll();
      if ( x == null ) throw new NoSuchElementException();
      return x;

  public E element()
      E x = peek();
      if ( x == null ) throw new NoSuchElementException();
      return x;

  public void clear()  while (poll() != null) ;
```

11.2.3 PriorityQueue

PriorityQueue is the only other implementation besides LinkedList which is not designed primarily for concurrent use; it is not thread-safe nor does it provide blocking behaviour. It gives up its elements for processing according to an ordering like that used by SortedSet—either the natural order of its elements if they implement Comparable, or the ordering imposed by a Comparator supplied when the PriorityQueue is constructed. But don't be misled by this similarity into thinking that a PriorityQueue is like a SortedSet with some added operations. For one thing, the iterator for a PriorityQueue is not guaranteed to return its elements in sorted order; you have to use the methods of the Queue interface to take advantage of the sorting. But you can use a SortedSet to make a PriorityQueue—one of the six constructors uses a PriorityQueue argument to supply the elements and ordering rule for the queue. The constructors are:

```
PriorityQueue()       // Natural ordering, default initial capacity (11)
PriorityQueue(Collection<? extends E> c)
                      // Natural ordering of elements taken from c
PriorityQueue(int initialCapacity)
                      // Natural ordering, specified initial capacity
PriorityQueue(int initialCapacity, Comparator<? super E> comparator)
                      // Comparator ordering, specified initial capacity
PriorityQueue(PriorityQueue<? extends E> c)
                      // Ordering and elements copied from c
PriorityQueue(SortedSet<? extends E> c)
                      // Ordering and elements copied from c
```

We can use `PriorityQueue` for a simple implementation of our to-do manager with the priority-based `Comparator` defined in Section 10.2. A queue using that comparator will always yield a task chosen by the description ordering from among those which currently have the highest priority.

```
Queue<Task> priorityQueue = new PriorityQueue<Task>( priorityComparator );
priorityQueue.addAll( allTasks );
...
Task nextTask = priorityQueue.poll();
```

Todo Give outline of priority heap implementation of `PriorityQueue`.

`PriorityQueue` provides $O(\log n)$ time for `offer`, `poll`, `remove()` and `add`, $O(n)$ time for `remove(Object)` and `contains`, and constant time for `peek`, `element`, and `size`.

11.2.4 `ConcurrentLinkedQueue`

The other non-blocking `AbstractQueue` subclass is `ConcurrentLinkedQueue`, an unbounded thread-safe queue based on linked nodes. It uses FIFO ordering, and is implemented using an algorithm which allows different threads to access it for adding and removing elements simultaneously without blocking. It would be a good structure to simulate the check-in queue described above.

`ConcurrentLinkedQueue` has the two standard constructors of Section 9.2.1.

11.3 BlockingQueue

Java 5 added a number of classes to the Collections Framework for use in concurrent applications. Most of these are implementations of the `Queue` subinterface `BlockingQueue` (Figure 11.4), designed primarily to be used in producer-consumer queues.

One common example of the use of producer-consumer queues is in systems which perform print spooling; client processes add print jobs to the spool queue, to be processed by one or more print service processes each of which repeatedly “consumes” the task at the head of the queue.

The key facilities which `BlockingQueue` provides to such systems are, as its name implies, enqueuing and dequeueing methods which do not return until they have executed successfully. So for example, a print server does not need to constantly poll the queue to discover whether there any print jobs waiting; it need only call the `poll` method, supplying a timeout, and the system will suspend it until either a queue element becomes available or the timeout expires. `BlockingQueue` defines seven new methods, in three groups:

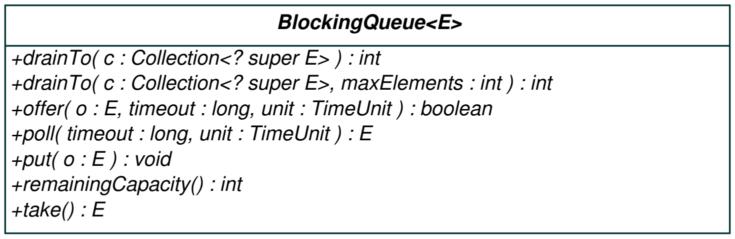

BlockingQueue<E>
+drainTo(c : Collection<? super E>) : int
+drainTo(c : Collection<? super E>, maxElements : int) : int
+offer(o : E, timeout : long, unit : TimeUnit) : boolean
+poll(timeout : long, unit : TimeUnit) : E
+put(o : E) : void
+remainingCapacity() : int
+take() : E

Figure 11.4: `BlockingQueue`

Retrieving or querying the contents of the queue:

```
int drainTo(Collection<? super E> c)
                // clears the queue into c
int drainTo(Collection<? super E> c, int maxElements)
                // clears at most specified number of elements into c
int remainingCapacity()
                // returns the number of elements that would be accepted
                // without blocking
```

The effect of `drainTo` is similar to repeated calls of `poll`, with the difference that these repeated calls would each have to obtain a lock on the queue, and that between them other threads would have the opportunity to change the contents of the queue. Since `drainTo` executes atomically it will in general be much more efficient.

Adding an element:

```
boolean offer(E o, long timeout, TimeUnit unit)
                // inserts o, waiting up to the timeout
void put(E o)
                // adds o, waiting indefinitely if necessary
```

The non-blocking overload of `offer` defined in `Queue` will return `false` if it cannot immediately insert the element. This new overload waits for a time specified using `java.util.concurrent.TimeUnit`, an `Enum` which allows timeouts to be defined in nanoseconds, microseconds, milliseconds or seconds.

Removing an element:

```
E poll(long timeout, TimeUnit unit)
                // retrieves and removes the head, waiting up to the timeout
E take()
                // retrieves and removes the head of this queue, waiting
                // indefinitely if necessary.
```

As with `offer`, the blocking overload of `poll` complements the non-blocking overload defined in `Queue`.

`BlockingQueue` implementations guarantee that their queue operations will be thread-safe and atomic. But this guarantee doesn't extend to operations inherited from `Collection`. So any thread running concurrently with others must get exclusive access to a `BlockingQueue` in order to execute any of these methods, or suffer the risk of destructive interference or—in the case of the bulk methods `addAll`, `containsAll`, `removeAll` and `retainAll`—the possibility of failing after partial completion.

11.3.1 Implementing BlockingQueue

The Collections Framework provides five implementations of `BlockingQueue`. We're going to present them here only briefly, as a detailed discussion of concurrency is outside the scope of this book.

`ArrayBlockingQueue`

This implementation is based on a *circular array*—a linear structure in which the first and last elements are logically adjacent (Figure 11.5).

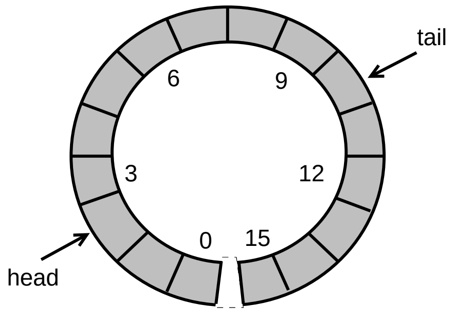


Figure 11.5: A Circular Array (only the section between head and tail-1 should be shaded)

The position labelled "head" indicates the head of the queue; each time the head element is removed from the queue, the head index is advanced. Similarly each new element is added at the tail position, resulting in that index being advanced. When either index needs to be advanced past the last element of the array, it gets the value 0. If the two indices have the same value the queue is either full or empty, so an implementation must separately keep track of the count of elements in the queue. A circular array in which the head and tail can be continuously advanced like this is better as a queue implementation than a non-circular one (*e.g.* the standard implementation of `ArrayList`, Section 12.2) in which removing the head element requires changing the position of all the remaining element so that the new head is at position 0.

Constructors for array-backed collection classes generally have a single configuration parameter, the initial length of the array. For fixed-size classes like `ArrayBlockingQueue`, this parameter is necessary in order to define the capacity of the collection. (For variable-size classes like `ArrayList`, a default initial capacity can be used, so constructors are provided that don't require the capacity). For `ArrayBlockingQueue`, the three constructors are:

```
ArrayBlockingQueue( int capacity )
ArrayBlockingQueue( int capacity, boolean fair )
ArrayBlockingQueue( int capacity, boolean fair, Collection<? extends E> c )
```

Besides configuring the backing array, the last two also require a boolean argument to control how the queue will handle multiple blocked requests. These will occur when multiple threads attempt to remove items from an empty queue or enqueue items on to a full one. When the queue becomes able to service one of these requests, which one should it choose? The alternatives are to choose the one which has been waiting longest—that is, to implement a *fair* scheduling policy—or to choose one arbitrarily. Fair scheduling sounds like the better alternative, since it avoids the possibility that an unlucky thread might be delayed indefinitely, but in practice it is rarely used because of the very large overhead that it imposes on a queue's operation.

`LinkedBlockingQueue`

If the fixed capacity of `ArrayBlockingQueue` represents a problem for your application, this class provides an alternative using the linked-list mechanism described in Section 11.2. The two standard collection constructors create a thread-safe blocking queue with a capacity of $2^{31} - 1$. A third constructor allows you to specify a lower capacity. `LinkedBlockingQueue` cannot implement a fair queuing strategy for blocked threads.

`PriorityBlockingQueue`

This implementation is a thread-safe, blocking version of `PriorityQueue` (Section 11.2). It uses the same priority ordering but supports multiple thread access. `PriorityBlockingQueue` always handles multiple blocked threads fairly.

`DelayQueue<E extends Delayed>`

This is a specialised priority queue, in which the ordering is based on the *delay time* for each element—the time remaining before the element will be ready to be taken from the queue. If all elements have a positive delay time—that is, none of their associated delay times have expired—`peek` and `poll` will return null. If one or more elements has an expired delay time, it is the one with the longest-expired delay time that will be at the head of the queue. The elements of a `DelayQueue` belong to a class which implements `java.util.concurrent.Delayed`:

```
interface Delayed extends Comparable<Delayed> {
    long getDelay(TimeUnit unit);
}
```

The `getDelay` method of a `Delayed` object returns the remaining delay associated with that object. The `compareTo` method (see Section 8.2) of `Comparable` must be defined to give results consistent with the delays of the objects being compared.

For example, in our to-do manager we are likely to need reminder tasks, to ensure that we follow up e-mails and phone messages that have gone unanswered. We could define a new class `DelayedTask` as follows:

```
class DelayedTask implements Delayed {
  public final static int MILLISECONDS_IN_DAY = 60 * 60 * 24 * 1000;
  private long endTime;     // in milliseconds after January 1, 1970
  private Task task;
  DelayedTask( Task t, int daysDelay ) {
    endTime = new Date().getTime() + daysDelay * MILLISECONDS_IN_DAY;
    task = t;
  }
  public long getDelay( TimeUnit unit ) {
    long remainingTime = endTime - new Date().getTime();
    return unit.convert( remainingTime, TimeUnit.MILLISECONDS );
  }
  public int compareTo( Delayed t ) {
    long thisDelay = getDelay( TimeUnit.MILLISECONDS );
    long otherDelay = t.getDelay( TimeUnit.MILLISECONDS );
    return ( thisDelay < otherDelay ) ? -1 : ( thisDelay > otherDelay ) ? 1 : 0;
  }
  Task getTask() { return t }
}
```

and use it to implement a reminder queue:

```
BlockingQueue<DelayedTask> reminderQueue = new DelayQueue<DelayedTask>();
reminderQueue.offer( new DelayedTask (task1, 1) );
reminderQueue.offer( new DelayedTask (task2, 2) );
...
// now get the first reminder task that is ready to be processed
DelayedTask t1 = reminderQueue.poll();
if ( t1 == null ) {
  // no reminders ready yet
} else {
  // process t1
}
```

Collection operations on a `DelayQueue` do not respect delay values, in contrast to Queue operations. This means, for example, that the two expressions

```
new HashSet<DelayedTask>(){{ addAll( delayedTaskQueue ); }}
new HashSet<DelayedTask>(){{ delayedTaskQueue.drainTo( this ); }}
```

will not be equal if there are any unexpired delays among the elements of `delayedTaskQueue`.

`SynchronousQueue`

At first sight, you might think there is little point to a queue with no internal capacity, which is a short description of `SynchronousQueue`. But in fact it can be very useful; a thread which wants to add an element to a `SynchronousQueue` must wait until another thread is ready to simultaneously take it off, and the same is true—in reverse—for a thread which wants to take an element off the queue. So the `SynchronousQueue` has the function that its name suggests, of a *rendezvous*—a mechanism for synchronizing two threads. (Don't confuse the concept of synchronous threads, which co-operate by working simultaneously, with Java's keyword `synchronized`, which prevents simultaneous execution of code by different threads).

The classic example of the use of a rendezvous is the situation in which two buffers are in use, by a writer thread which is filling one and by a reader thread which is emptying the other. When each is ready, the two buffers have to be simultaneously exchanged.

```
//writer thread writes                              //reader thread reads
//to buffer until empty                             //from buffer until full
                                                    Buffer b1 = buffer;
sQueue.offer( buffer );  --synchronizes with--   buffer = sQueue.take();
buffer = sQueue.take();  --synchronizes with--   sQueue.offer( b1 );
//writes to buffer                                  //reads from buffer
```

As far as the `Collection` methods are concerned, a `SynchronousQueue` behaves like an empty `Collection`. `Queue` and `BlockingQueue` methods behave as you would expect for a queue with zero capacity, except for `peek`: this always returns null, even when a thread is waiting to offer an element to the queue, because a `SynchronousQueue` exists to exchange elements, not to store them. `SynchronousQueue` has one empty constructor and one which takes a boolean parameter controlling whether the queue should implement a fair blocking strategy.

Chapter 12

Lists

Although we come to them after sets and queues, lists are probably the most-used Java collections in practice. A `List` (Figure 12.1) is a collection which—unlike a set—can contain duplicates, and which—unlike a queue—gives the user full visibility and control over the ordering of its elements.

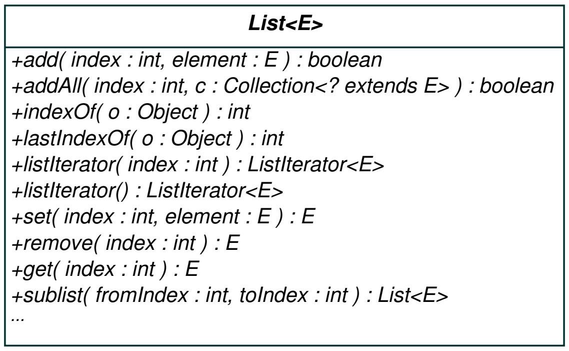

List<E>
+add(index : int, element : E) : boolean
+addAll(index : int, c : Collection<? extends E>) : boolean
+indexOf(o : Object) : int
+lastIndexOf(o : Object) : int
+listIterator(index : int) : ListIterator<E>
+listIterator() : ListIterator<E>
+set(index : int, element : E) : E
+remove(index : int) : E
+get(index : int) : E
+sublist(fromIndex : int, toIndex : int) : List<E>
...

Figure 12.1: `List`

In addition to the operations inherited from `Collection`, the `List` interface includes operations for:

Positional Access: manipulate elements based on their numerical position in the list:

```
void add( int index, E element )          // add element o at given index
boolean addAll(int index, Collection<? extends E> c )
                                          // add contents of c at given index
E get( int index )                        // return element with given index
E remove( int index )                     // remove element with given index
E set( int index, E element )             // replace element with given index
```

Search: search for a specified object in the list and return its numerical position. These methods return -1 if the object is not present.

```
int indexOf(Object o)                    // return index of first occurrence of o
int lastIndexOf(Object o)                // return index of last occurrence of o
```

Range-view: get a view of a range of the list.

```
List<E> subList(int fromIndex, int toIndex)
                                         // return a view of a portion of the list
```

`subList` works like the subsequence operations on `SortedSet` (Section 10.2): the returned list contains the list elements starting at `fromIndex` up to but not including `toIndex`. The returned list has no separate existence—it is just a view of part of the the list from which it was obtained, so changes in it are reflected in the original list, and vice versa.

List iteration: extend `Iterator` semantics to take advantage of the list's sequential nature.

```
ListIterator<E> listIterator()           // returns a ListIterator for this list
ListIterator<E> listIterator( int index) // returns a ListIterator for this list,
                                         //   starting at the given index
```

`ListIterator` is an interface which adds methods to `Iterator` to traverse a list backwards, to change list elements or add new ones, and to get the the current position of the iterator. The current position of a `ListIterator` always lies between two elements, the one that would be returned by a call to `previous` and the one that would be returned by a call to `next`. In a list of length n, there are $n + 1$ valid index values, from 0 (before the first element) to n (after the last one). To the `Iterator` methods `hasNext`, `next` and `remove`, `ListIterator` adds the following methods:

```
public interface ListIterator<E> extends Iterator<E> {
  void add(E o);           // Inserts the specified element into the list
  boolean hasPrevious();   // Returns true if this list iterator has further
                           //   elements in the reverse direction.
  int nextIndex();         // Returns the index of the element that would be
                           // returned by a subsequent call to next.
  E previous();            // Returns the previous element in the list.
  int previousIndex();     // Returns the index of the element that would be
                           // returned by a subsequent call to next
  void set(E o);           // Replaces the last element returned by next or
                           //   previous with the specified element
}
```

12.1 Using the Methods of `List`

Let's look at examples of the use of these methods. You may remember that although queues—especially `PriorityQueue`—looked like a plausible implementations for our to-do manager, they have one severe drawback: there is no way in general to inspect or manipulate any element other than the one at the head. We found one example of this drawback in the situation in which we were looking for a short task to fit into a tight time-slot, but there are many others. Lists provide low-level facilities for manipulating their contents, but at a price: we have to do more low-level work.

For example, there is no list implementation corresponding to `PriorityQueue` which will organise our tasks by priority. We will have to do it for ourselves, in this case by creating a separate task list for each priority. For the moment, we will keep the references to these separate lists in another list which we can initialise like this:

```
List<List<Task>> taskLists = new ArrayList<List<Task>>();
for( Priority p : Priority.values() ) {
  taskLists.add( new ArrayList<Task>() );
}
```

and to add a task `sampleTask` we can write

```
Priority p = sampleTask.getPriority();
taskLists.get(p.ordinal()).add(sampleTask);
```

Now we can, for example, carry out all of the phone tasks in turn, in descending order of priority.

```
for( List<Task> taskList : taskLists  ) {
  for( Iterator<Task> it = taskList.iterator() ; it.hasNext() ; ) {
    Task t = it.next();
    if( t instanceof PhoneTask ) {
      // process t
    }
  }
}
```

For the ordering of tasks within each of the fixed-priority lists, we will treat the list like a queue having the head element at index 0 and the tail element at index $n - 1$, where n is the number of outstanding tasks. `offer` can then be simulated by `add(E o)`—which appends the supplied element to the end of the list—and the methods which retrieve the head of the queue can be simulated by `remove(0)` and `get(0)`. With this understanding, let's look at examples of the use of methods from each of the groups above.

Positional Access: although in the normal way of things new tasks will be added on to the end of each of the task lists, there may be tasks that just have to go to the head of the queue:

```
List<Task> highPriorityTaskList = taskLists.get( Priority.HIGH.getOrdinal() );
highPriorityTaskList.add( 0, new PhoneTask( "home", Priority.HIGH, "321 9876" ));
```

or if—the next example is sadly all too frequent!—we have to do all the tasks left over from Monday before we can start on Tuesday's:

```
tuesdayHighPriorityTasks.addAll( 0, mondayHighPriorityTasks );
```

We often like to inspect the next task, without necessarily committing to doing it immediately (as with the `peek` method of `Queue`):

```
Task nextTask = highPriorityTaskList.get( 0 );
```

but if we are really confident that we can do it, we can simultaneously retrieve it and remove it from the list, as with the `remove` method of `Queue`. Notice that `List` has no method like `Queue.poll`—if you try to remove a non- existent method from a `list`, it will always throw `NoSuchElementException`. The indices of subsequent elements are adjusted by `remove`, so the `List` will still begin at index 0.

```
Task nextTask = highPriorityTaskList.remove( 0 );
```

But the only real answer to our problems is not to try to do so many things. As a start we're just going to drop the last activity on the low-priority list—the last one we aim to do in the day. Better still, let's reward ourselves for actually getting to that point by scheduling a more agreeable replacement "task":

```
Task dayEndTask = new Task( "go to pub", Priority.LOW );
lowPriorityTaskList.set( lowPriorityTaskList.size() - 1, dayEndTask );
```

The `set` method actually returns a reference to the object being replaced, but to really follow through on our decision that "less is more", we have to steel ourselves to throw that reference away on this occasion.

Search: We often absent-mindedly insert the same task into the system more than once, so it would be useful to be able to remove the duplicate entries. One way would be to identify the first occurrence of a task and then repeatedly remove the last occurrence as identified by `lastIndexOf`.

```
for( int i = 0 ; i < taskList.size() ; i++  ) {
  // get the next task to de-duplicate
  Task task = taskList.get(i);
  while( i != taskList.lastIndexOf( task )) {
    taskList.remove( taskList.lastIndexOf( task ) );
  }
}
```

Your first thought for this example might have been to use *foreach* or, equivalently. an explicit iterator to work through the tasks to be de-duplicated. But recall from Section 8.6.1 that if the iterator on `taskList` is fail-fast, that code would fail with `ConcurrentModificationException`. This is an example of a sound program caught up in the blanket restriction on "concurrent" modification.

Range-view: If we wanted to build co-operative working into the new task manager, we might define a method to allow us to share task lists:

```
  public void shareTasks( List<List<Task>> taskLists )
```

but we might not always want to share *all* of the task lists. Suppose, for example that we wanted to supply a list to `shareTasks` consisting of only the top two priority lists. We would need to create a view of our `List<List<Task>>` containing only those two elements, like this:

```
  shareTasks( taskList.subList( 0, 2 ) )
```

List iteration: The way that we removed duplicate tasks using `lastIndexOf` was simple but inefficient. We repeatedly found the last occurrence of an element by searching for it from the very end of the list. A list iterator will allow us to do the job more efficiently, resuming the search each time from the point at which the last duplicate element was found.

```
for( int i = 0 ; i < taskList.size() ; i++  ) {
  Task task = taskList.get(i);
  // get a ListIterator, initially pointing past the end of the list
  ListIterator<Task> iter = taskList.listIterator( taskList.size() );
  Task t1 = iter.previous();
  // now search backwards removing duplicates, and stopping when we
  // reach the original task
  while( task != t1 ) {
    if ( task.equals( t1 ) ) {
      iter.remove();
    }
    t1 = iter.previous();
  }
}
```

12.2 Implementing `List`

There are three concrete implementations of `List` in the Collections Framework, differing again, as with the implementations of `Set`, in how fast they perform the various operations defined by the interface; in this case, however, there are no functional differences between the implementations. Figure 12.2 shows how they are related. In this section we look at each implementation in turn, and in Section 12.3 we provide a performance comparison.

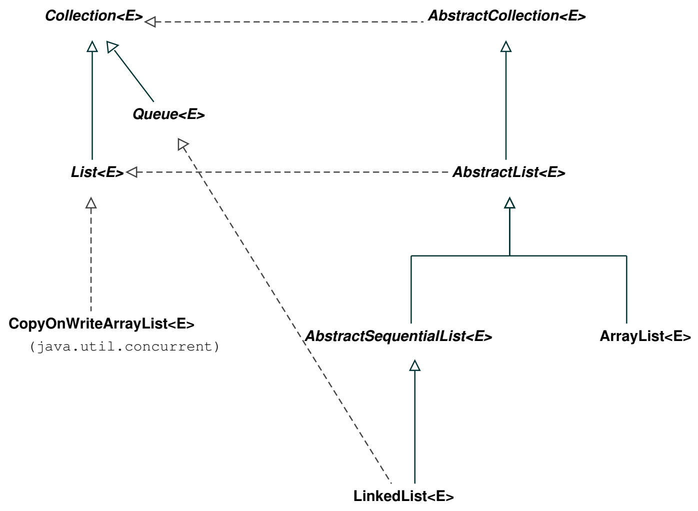


Figure 12.2: Implementations of the `List` interface (extracted from the front endpaper)

12.2.1 `AbstractList`

The Collections Framework again provides a skeletal abstract implementation, `AbstractList`, as a basis for concrete implementations. Example 12.1 shows an outline implementation of `AbstractList`, although for reasons of space it omits the implementation of `subList`.

Most of the methods on the `List` interface are implemented in `AbstractList`, although `set`, `remove` and `add` simply throw `UnsupportedOperationException`, to be overridden in subclasses which implement modifiable `List`s. The other methods are implemented in terms of `get`, which is left abstract and as a non-optional operation must be implemented by any subclass. Of the methods inherited from `AbstractCollection`, `size` is still abstract but `iterator` returns an instance of an inner class whose `next` method relies on `get`.

Example 12.1 Outline implementation of `AbstractList`

```
public abstract class AbstractList<E> extends AbstractCollection<E> implements List<E> {
  // Object method overrides
  public boolean equals(Object o) {
    if (!(o instanceof List)) return false;
    List l = (List) o;
    if ( size() != l.size() ) return false;
    for ( int i = 0 ; i < size() ; i++ ) {
      if ( get(i) == null ? l.get(i) != null : ! get(i).equals( l.get(i) ) ) { return false; }
    }
    return true;
  }
  public int hashCode() {
    int h = 1;
    for( E el : this ) h += h * 31 + ( el==null ? 0 : el.hashCode() );
    return hashCode;
  }
  // Collection method implementations
  public boolean add(E o) {
    add(size(), o);
    return true;
  }
  public Iterator<E> iterator() { return new Itr(); }
  // List method implementations
  public E set(int index, E element) { throw new UnsupportedOperationException(); }
  public E remove(int index) { throw new UnsupportedOperationException(); }
  public void add(int index, E element) { throw new UnsupportedOperationException();  }
  public boolean addAll(int index, Collection<? extends E> c) {
    for( E el : c ) { add(index++, e.next()); }
    return c.size() > 0;
  }
  public abstract E get( int index );
  public int indexOf(Object o) {
    ListIterator<E> e = listIterator();
    while( e.hasNext() ) {
      E el = e.next();
      if ( o == null ? el == null : o.equals( el ) ) { return e.previousIndex(); }
    }
    return -1;
  }
```

Example 12.2 Outline implementation of `AbstractList` (continued)

```
    public int lastIndexOf(Object o) {
      ListIterator<E> e = listIterator(size());
      while( e.hasPrevious() ) {
        E el = e.previous();
        if ( o == null ? el == null : o.equals( el ) ) { return e.nextIndex(); }
      }
      return -1;
    }
    public ListIterator<E> listIterator() {  return listIterator(0); }
    public ListIterator<E> listIterator(final int index) {
      if (index<0 || index>size()) throw new IndexOutOfBoundsException();
      return new ListItr(index);
    }
    public List<E> subList(int fromIndex, int toIndex) { /* ... */  }
    // Iterator classes
    private class Itr implements Iterator<E> {
      int cursor = 0;
      int lastRet = -1;
      public boolean hasNext() { return cursor != size(); }
      public E next() {
        try {
          E next = get(cursor);
          lastRet = cursor++;
          return next;
        } catch(IndexOutOfBoundsException e) { throw new NoSuchElementException();}
      }
      public void remove() {
        if (lastRet == -1) throw new IllegalStateException();
        AbstractList.this.remove(lastRet);
        if (lastRet < cursor) cursor--;
        lastRet = -1;
      }
    }
  }
```

Example 12.3 Outline implementation of AbstractList (continued)

```
private class ListItr extends Itr implements ListIterator<E> {
  ListItr(int index) { cursor = index; }
  public boolean hasPrevious() { return cursor != 0; }
  public E previous() {
    try {
        int i = cursor - 1;
        E previous = get(i);
        lastRet = cursor = i;
        return previous;
    } catch(IndexOutOfBoundsException e) { throw new NoSuchElementException(); }
  }
  public int nextIndex() { return cursor; }
  public int previousIndex() { return cursor-1; }
  public void set(E o) {
    if (lastRet == -1) throw new IllegalStateException();
    AbstractList.this.set(lastRet, o);
  }
  public void add(E o) {
    AbstractList.this.add(cursor++, o);
    lastRet = -1;
  }
}
```

So for direct subclasses of AbstractList the efficiency of iteration depends on the efficiency of random access. That's appropriate for the first class we will look at, which uses arrays—of all linear data structures the fastest for random access—to provide an implementation of List. It's exactly the wrong thing to do for classes dependent on sequential access, though, so AbstractSequentialList switches the dependency around. List classes themselves will always know whether they are a sequential or random implementation, of course, but generic algorithms (Chapter 15), which often have a sequential and a random variant, need a hint in order to be able to adopt the right one. That hint is provided by the RandomAccess, a marker interface (one with no methods) which is implemented by all the List implementations which are backed by arrays.

A very simple example of the use of AbstractList is given by the class Triples (Example 12.4), which implements all the non-optional methods of the List interface. Triples behaves as an immutable List view which appears to be a list of three identical elements, although in fact only one is physically present. The same idea is used in the Collections class to implement the method nCopies (Section 14.1).

12.2.2 ArrayList

Arrays are provided as part of the Java language and have a very convenient syntax, but their key disadvantage—that once created they cannot be resized—makes them increasingly less popular than List implementations which (if resizable at all) are indefinitely extensible. The most commonly-used implementation of List is in fact ArrayList—that is, a List backed by an array.

Example 12.4 `Triples`

```
public class Triples<E> extends AbstractList<E>
        implements RandomAccess, Serializable {
    private E element;
    public Triples(E o) { element = o; }
    public int size() { return 3; }
    public boolean contains(Object obj) {
        return obj == null ? element == null : obj.equals( element );
    }
    public E get( int index ) {
        if ( index < 0 || index > 2 )
            throw new IndexOutOfBoundsException(
                    "Index: "+ index + " incorrect for Triple");
        return element;
    }
}
```

The standard implementation of `ArrayList` stores the List elements in contiguous array locations, with the first element always stored at index 0 in the array. It requires an array at least large enough (with sufficient *capacity*) to contain the elements, together with a way of keeping track of the number of "occupied" locations (the size of the `List`). If an `ArrayList` has grown to the point where its size is equal to its capacity, attempting to add another element will require it to replace the backing array with a larger one capable of holding the old contents and the new element, and with a margin for further expansion (the standard implementation actually uses a new array double the length of the old one).

The performance of `ArrayList` reflects array performance for "random-access" operations: `set` and `get` take constant time. And `ArrayList` inherits `AbstractList`'s iterator, whose `next` method is defined in terms of `get`, so that too takes constant time. The downside of an array implementation is in inserting or removing elements at arbitrary positions, because that may require adjusting the position of other elements. For example, Figure 12.3(a) shows a new `ArrayList` after three elements have been added by means of the following statement:

```
new ArrayList<Character>(){{ add('a'); add('b'); add('c'); }};
```

If we now want to remove the element at index 1 of an array, the implementation must preserve the order of the remaining elements and also ensure that the occupied region of the array is still to start at index 0. So the element at index 2 must be moved to index 1, that at index 3 to index 2, and so on. Figure 12.3(b) shows our sample `ArrayList` after this operation has been carried out. Since every element must be moved in turn, the time complexity of this operation is proportional to the size of the list, although it is worth noting that because this operation can usually be implemented in hardware, the constant factor is low.

Even so the alert reader, recalling the discussion of the circular array used to implement `ArrayBlockingQueue` (Section 11.3.1), may wonder why a circular array was not chosen for the implementation of `ArrayList` too. It is true that the `add` and `remove` methods of a circular array only

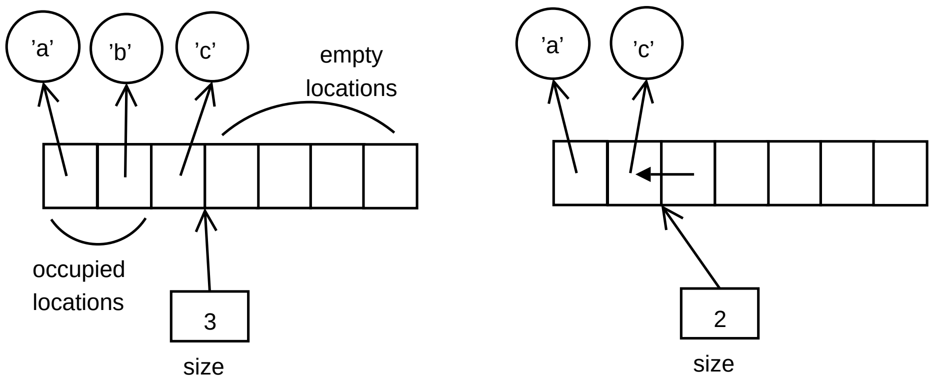


Figure 12.3: Removing an element from an `ArrayList`

show a much better performance when they are called with an index argument of 0, but this is such a common case and the overhead of using a circular array is so small, that the question remains. An outline implementation of a circular array list was presented by Heinz Kabutz, in *The Java Specialists' Newsletter*, (http://www.javaspecialists.co.za/archive/Issue027.html).

As we mentioned in the discussion of `ArrayBlockingQueue` (Section 11.2), variable-size array-backed collection classes can have one configuration parameter—the initial length of the array. So besides the standard Collection Framework constructors, `ArrayList` has one which allows you to choose the value of the initial capacity to be large enough to accommodate the elements of the collection without frequent create-copy operations. If you choose one of the others, the standard implementation uses an initial capacity of 10.

12.2.3 `AbstractSequentialList`

In discussing `AbstractList` we mentioned that the efficiency of iteration depends on the efficiency of the `get` method. This is appropriate for a `List` implementation like `ArrayList`, which is backed by a data structure that implements `get` efficiently on random index values. A class like `LinkedList`, on the other hand (which we met as a `Queue` implementation in Section 11.2) can only access its data sequentially, so implementing iteration in terms of `get` is exactly the wrong way round. `AbstractSequentialList` is designed for situations like this; it overrides the `AbstractList` implementation of `listIterator` with an abstract declaration, and replaces the abstract `get` method of `AbstractList` with a concrete definition in terms of iteration over the list.

To implement a sequential list using this class you need to override the `listIterator` and `size` methods and, if the list is to be modifiable, implement the `add`, `set` and `remove` methods of the iterator.

12.2.4 `LinkedList`

We learnt most of what we need to know about `LinkedList` when we considered it as a `Queue` implementation (Section 11.2). The only feature that was not relevant to discuss in the context of queues is the double linking (Figure 12.4) that makes `hasPrevious` and `previous` as efficient as `hasNext` and `next`. `LinkedList` can add and remove elements in constant time, but random access must be implemented through iteration.

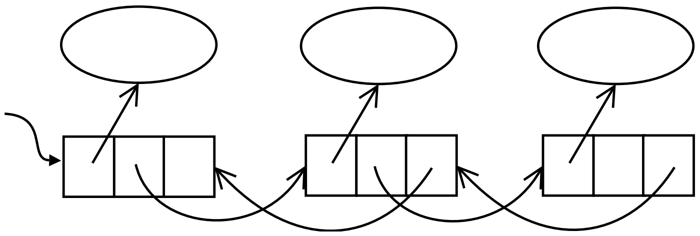


Figure 12.4: A Doubly-Linked List

12.2.5 `CopyOnWriteArrayList`

In Section 10.1 we met `CopyOnWriteArraySet`, a set implementation designed to provide thread-safety together with very fast read access. `CopyOnWriteArrayList` is a `List` implementation with the same design aims. This combination of thread safety with fast read access is useful in some concurrent programs, especially when a collection of observer objects needs to receive frequent event notifications. The cost is that the collection has to be immutable, and that therefore a new one must be created if any changes are made. This cost may not be too high to pay if changes in the set of observers occur only rarely.

`CopyOnWriteArraySet` in fact delegates all its operations to an instance of `CopyOnWriteArrayList`. There are two differences between these two classes; one is that the `add` and `addAll` methods of `CopyOnWriteArraySet` are implemented in terms of methods `addIfAbsent` and `addAllAbsent` provided by `CopyOnWriteArrayList` to ensure that the set contains no duplicates. The other difference is that `CopyOnWriteArrayList` has an extra constructor— in addition to the two standard ones (Section 9.2.1)—that allows it to be created using the elements of a supplied array as its initial contents.

12.3 Comparison of `List` Implementations

As always, the O-formulae in Figure 12.5 don't tell the whole story, not least because of the constant factors they omit. For example, although element addition and removal in an `ArrayList` take linear time, the element copying operation can usually be implemented in hardware and is therefore very fast. By contrast, although calling `next` in a `LinkedList` takes constant time, a pointer has to be followed so the constant is quite large.

	`get`	`add`	`contains`	`next`	notes
`ArrayList`	$O(1)$	$O(n)$	$O(n)$	$O(1)$	
`LinkedList`	$O(n)$	$O(1)$	$O(n)$	$O(1)$	
`CopyOnWriteArrayList`	$O(1)$	$O(n)$	$O(n)$	$O(1)$	

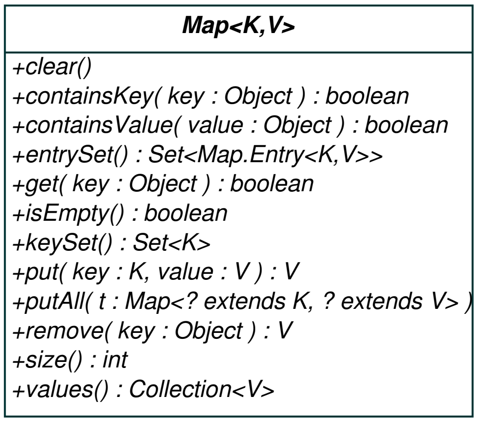
Figure 12.5: Comparative performance of different `List` implementations

The cost of `add`ing an element to an `ArrayList` includes an allowance for the cost of expanding the list when the array is full. This is an example of an amortized cost (Section 8.4): the cost of transferring the contents of the old array to the new one is linear in the number of elements in the old array. This cost has to be incurred repeatedly for each element as the array continues to grow in size, but if each resize doubles the size of the array (or multiplies it by any constant factor), the frequency of resizing continually decreases. Overall, the total cost of resizing the `ArrayList` for N elements is $O(N)$, so the amortized cost is $O(1)$.

Chapter 13

Maps

`Map` is the last of the major Collection Framework interfaces in this chapter, and the only one which does not inherit from `Collection`. It defines the operations which are supported by a set of key-to-value associations in which the keys are unique. These operations are shown in Figure 13.1 and fall into five groups, broadly parallel to three of the operation groups of `Collection`—adding elements, removing elements, and querying collection contents. The fourth operation group of `Collection` contains `iterator` and `toArray`, for which `Map` has no corresponding methods.



Map<K,V>

+clear()
+containsKey(key : Object) : boolean
+containsValue(value : Object) : boolean
+entrySet() : Set<Map.Entry<K,V>>
+get(key : Object) : boolean
+isEmpty() : boolean
+keySet() : Set<K>
+put(key : K, value : V) : V
+putAll(t : Map<? extends K, ? extends V>)
+remove(key : Object) : V
+size() : int
+values() : Collection<V>

Figure 13.1: The `Map` interface (extracted from the back endpaper)

Adding associations:

```
V put( K key, V value )                // adds or replaces a key-value association.
                                       // Returns the old value (may be null) if the
                                       // key was present, otherwise returns null
void putAll(Map<? extends K,           // adds each of the key-value associations in
                    ? extends V> t)    // the supplied map into the receiver
```

Removing associations:

```
void clear()                 // removes all associations from this map
V remove(Object key)         // removes the association, if any, with the given
                             // key; returns the value with which it was associated,
                             // or null
```

You may recall from Section 9.1 that the `remove` method of `Collection` also has a parameter of type `Object`. Since the contract for `remove` provides for it to do nothing if the key is not present in the map, there can be no harm in allowing a non-generic parameter here.

The operations in this and the previous group, to add and remove associations, are optional; calling them on an unmodifiable map will not change the map and may result in an `UnsupportedOperationException`.

Querying for the existence of presence of specific keys, values, or associations:

```
boolean containsKey(Object k)   // returns true if k is present as a key. k may be null
                                // for maps that allow null keys.
boolean containsValue(Object v) // returns true if v is present as a value. v may be null.
V get(Object k)                 // returns the value corresponding to k, or null if k is
                                // not present as a key.
```

Retrieving the keys, values or associations in bulk:

```
Set<Map.Entry<K, V>> entrySet() // returns a set view of the key-value associations.
Set<K> keySet()                 // returns a set view of the keys
Collection<V> values()          // returns a set view of the values
```

The collections returned by these methods are backed by the map, so any changes to them are reflected in the map itself. In fact, only limited changes can be made: elements can be removed from them, either directly or via an iterator, but cannot be added. Removing a key removes the single corresponding key-value association; removing a value, on the other hand, removes only one of the associations mapping to it; the value may still be present as part of an association with a different key.

Querying the association count

```
int size()               // returns the number of key-value associations
boolean isEmpty()        // returns true if there are no key-value associations
```

As with the `size` method of `Collection`, the largest element count that can be reported is `Integer.MAX_VALUE`.

13.1 Using the Methods of `Map`

In the last section we modelled the outstanding tasks with a `List<List<Task>>`, placing the list of lowest-priority tasks at index 0, the next-lowest at position 1, and so on. This feels clumsy, especially when we see that to retrieve a task list we have to write

```
List<Task> highPriorityTaskList = taskLists.get( Priority.HIGH.getOrdinal() );
```

Now that we have Maps, we can improve on using a list of lists by instead mapping priorities to tasklists:

```
Map<Priority,List<Task>> taskMap = new HashMap<Priority,List<Task>>();
for( Priority p : Priority.values() ) {
    taskMap.put( p, new ArrayList<Task>() );
}
```

And to get the highest-priority task list as above we can now write, rather more neatly,

```
List<Task> highPriorityTaskList = taskMap.get( Priority.HIGH );
```

To see the use of some of the other methods of Map, let's extend the example a little to allow for the possibility that some of these tasks might actually earn us some money by being billable. One way of representing this would be by defining a class Client:

```
class Client {...}
Client c1 = new Client(...);
```

and creating a mapping from task details to client objects:

```
Map<String,Client> billingMap = new HashMap<String,Client>();
billingMap.put( "code ui", c1 );
```

How can we ensure that the system can still handle non-billable tasks? There is a choice: we can either simply not add the name of a non-billable task into the billingMap or, alternatively, we can map it to null. Let's take the first alternative, so that as part of the code for processing a task t we can write

```
String taskDetails = t.toString();
if ( billingMap.containsKey( taskDetails ) {
  Client client = billingMap.get( taskDetails );
  client.bill(...);
}
```

When finally we have finished all the work we were contracted to do by our client Acme Corp., the mappings which associate task details with Acme can be removed:

```
Client acme = ...
Collection<Client> clients = billingMap.values();
for( Iterator<Client> iter = clients.iterator() ; iter.hasNext() ; ) {
    if (iter.next().equals( acme ))  {
        iter.remove();
    }
}
```

13.2 Implementing Map

The implementations that the Collections Framework provides for Map are shown in Figure 13.2. As with the other main interfaces, the Framework provides a skeletal implementation (AbstractMap) for subclassing, together with seven concrete implementations. We shall discuss HashMap, LinkedHashMap,

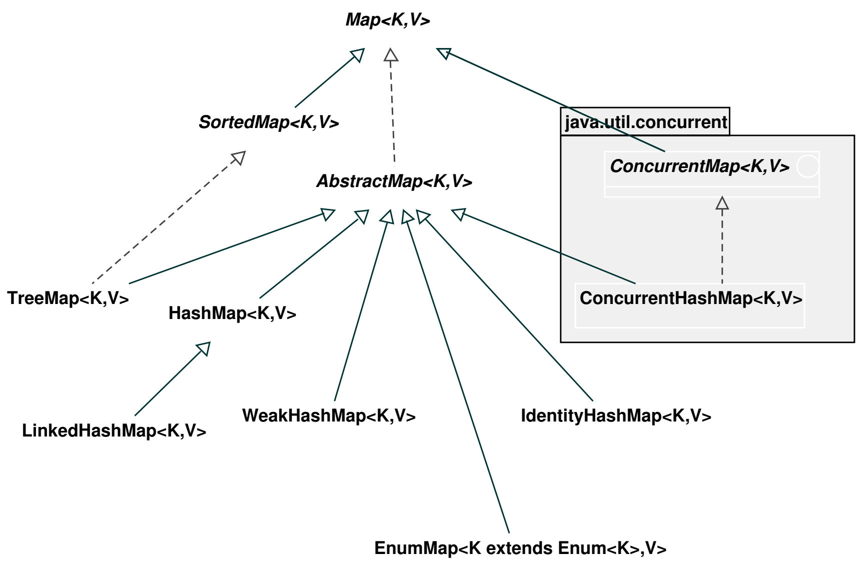


Figure 13.2: The structure of `Map` implementations in the Collections Framework (extracted from the front endpaper)

`WeakHashMap` and `IdentityHashMap` here; `EnumMap` is specialised for use with enums and is discussed with them in Section ???, and `SortedMap` and `ConcurrentMap` are discussed along with their implementations in the sections following this one.

For constructors, the general rule for `Map` implementations is parallel to that for `Collection` subclasses. Every general-purpose implementation (excluding `EnumMap`, that is) has at least two constructors; taking `HashMap` as an example they are

```
public HashMap()
public HashMap( Map<? extends K,? extends V> m )
```

The first of these creates an empty map, and the second a map that will contain the key-value mappings contained in the supplied map `m`. The keys and values of `m` must have types which are the same as of the keys and values, respectively, of the map being created—or subtypes of them, of course. Using this second constructor has the same effect as creating an empty map with the default constructor, then adding the contents of `m` using `putAll`. In addition to these two, the standard implementations have other constructors for configuration purposes.

Iterators returned by most of the standard `Map` implementations are fail-fast in the same way as `List` iterators are (Section 12.1). The exceptions are `ConcurrentHashMap` and `EnumMap`, whose iterators avoid throwing `ConcurrentModificationException`; as a result, the view of the map that they reflect may not include modifications that have occurred since they were created.

13.2.1 `AbstractMap`

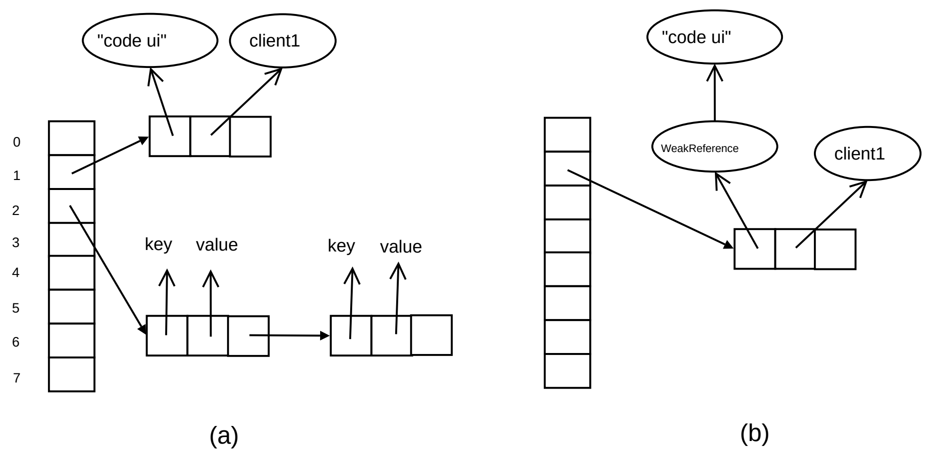


Figure 13.3: `HashMap` and `WeakHashMap`

[tbs: outline implementation of `AbstractMap`]. Most of the methods on the `Map` interface are implemented in `AbstractMap`, although `clear`, `put`, `putAll`, and `remove` simply throw `UnsupportedOperationException`, to be overridden in subclasses which implement modifiable `Map`s. The other methods are implemented in terms of `entrySet`, which is left abstract and as a non-optional operation must be implemented by any subclass.

Notice how much this limits the practical usefulness of `AbstractMap`; if all access to its data has to be through a set of key-value mappings, then operations like `get` can only be implemented by iterating through that set to find the right key. We know from the discussion of `HashSet` (Section 10.1) that hash tables can give much better performance than that, so it is not surprising to find that almost all the standard implementations of `Map` are hash-based and therefore override virtually all the methods of `AbstractMap`.

13.2.2 `HashMap`

In discussing set implementation in Section 10.1 we mentioned that `HashSet` delegates all its operations to a private instance of `HashMap`. Figure 13.3(a) repeats Figure 10.2 without the simplification that removed the value elements from the map (all elements in a `HashSet` are stored as keys with the same constant value). The discussion in Section 10.1 of hash tables and their performance applies equally to `HashMap`. In particular, without key collisions it provides constant-time performance for `put` and `get`. Iteration over a collection of keys or values requires time proportional to its capacity plus the number of key-value mappings that it contains.

The extra constructors for `HashMap` are also like those of `HashSet`, allowing the programmer to specify the initial capacity and, optionally, the load factor at which the table will be rehashed.

13.2.3 `LinkedHashMap`

Like `LinkedHashSet`, `LinkedHashMap` refines the contract of its parent class `HashMap`, by guaranteeing that its iterators will return their elements in the order in which they were first added. In the same way, iteration over a `LinkedHashMap` takes time proportional only to the number of elements in the map, not its capacity. Its constructors are like those of `HashMap`.

13.2.4 `WeakHashMap`

An ordinary `Map` keeps ordinary ("strong") references to all the objects it contains. That means that even when a key has become unreachable it cannot be garbage collected. Normally, that's exactly what we want; in the previous section, where we mapped task details to client objects, we wouldn't want a mapping to disappear just because we weren't holding a reference to the string object that we had put into the `HashMap` to represent the task details. To look up the value associated with a supplied key, the `HashMap` will look for a key which matches (in the sense of `equals`) the supplied one—they don't have to be physically the same object.

But suppose that the objects of the key class are unique—that is, no two objects of that class ever are equal in the sense of `equals`. For example, each object might contain a unique serial number. In this case, once we no longer have a reference—from outside the map—to an object being used as a key, we can never look it up again, because we can never recreate it. So the map might as well get rid of the key-value pair and, in fact, there may be a strong advantage in doing so if the map is large and memory is in short supply. That is the purpose that `WeakHashMap` accomplishes.

Internally `WeakHashMap` holds references to its key objects through references of the class `java.lang.ref.WeakReference` (Figure 13.3(b)). A `WeakReference` introduces an extra level of indirection in reaching an object. For example, to make a weak reference to an object `myObj` you would write

```
WeakReference wref = new WeakReference( myObj );
```

and at a later time recover a strong reference to it using the `get` method of `WeakReference`:

```
Object myObjRecoveredRef = wref.get();
```

If the reference variable `myObj` is nulled or goes out of scope and there are no other strong references to the the object it referred to, then the existence of the weak reference will not by itself prevent the garbage collecter from reclaiming the object. So the recovered reference value `myObjRecoveredRef` may (or may not) be null.

`WeakHashMap` uses `WeakReferences` to refer to its keys. Before most operations on a `WeakHashMap` are executed, the map checks which keys have been reclaimed (the `WeakReference` mechanism allows you to tell the garbage collector to leave you information each time it reclaims a weakly referenced object). For each key which the garbage collector has reclaimed, the `WeakHashMap` removes the entire key-value association.

What is a `WeakHashMap` good for? Imagine you were using a map to associate extra information with objects and you had a high turnover of objects; you would need to remember to remove each key object to prevent the map growing indefinitely and causing a memory leak. If the map were a `WeakHashMap`, though, neglecting to remove unwanted keys wouldn't be a serious problem, as they would eventually be removed by the garbage collector. So one use is to guard against memory leaks.

A more general application is in those situations where you don't mind information disappearing if memory is low, like a cache. Here `WeakHashMap` is useful whether or not the keys are unique, because you can always recreate a key if necessary to see if the corresponding value is still in the cache. `WeakHashMap` isn't perfect for this purpose—because it weakly references the map's keys rather than its values (usually much larger), it gives the garbage collecter the wrong information about the gains that it can make. A second drawback is that weak references are *too* weak; the garbage collector is liable to reclaim a weakly reachable object at any time, and the programmer cannot influence this in any way. (A sister class of WeakReference, java.lang.ref.SoftReference, is treated differently: the garbage collector should postpone reclaiming these until it is under severe memory pressure. Heinz Kabutz has written a SoftReference-based map using generics; see http://www.javaspecialists.co.za/archive/Issue098.html).

WeakHashMap performs similarly to HashMap though more slowly because of the overheads of the extra level of indirection for keys. The cost of clearing out unwanted key-value associations before each operation is proportional to the number of associations that need to be removed.

13.2.5 IdentityHashMap

An IdentityHashMap differs from an ordinary HashMap in that two keys are considered equal only if they are physically the same object: identity, rather than equals, is used for key comparison (in contravention of the contract for Map). This is a specialised class, commonly used in operations like serialization, in which a graph has to be traversed and information stored about each node. The algorithm used for traversing the graph must be able to check, for each node it encounters, whether that node has already been seen; otherwise graph cycles could be followed indefinitely. An ordinary map can only detect duplication by equality, and of course a graph could contain two different but equal nodes. An IdentityHashMap, by contrast, will report a node as being present only if that same node has previously been put into the map.

The standard implementation of IdentityHashMap handles collision differently from the chaining method shown in Figure 10.2 and used by all the other variants of HashSet and HashMap. This implementation uses a technique called *linear probing*, in which the key and value references are stored directly in adjacent locations in the table itself, rather than in cells referenced from it. In the standard implementation, collisions are handled by simply looking stepping along the table until the first free pair of locations is found. Figure 13.4 shows three stages in filling an IdentityHashMap with a capacity of 8. In (a) we are storing a key-value pair whose key hashes to 0, and in (b) a pair whose key hashes to 4. The third key, added in (c), also hashes to 4, so the algorithm steps along the table until it finds an unused location. In this case the first one it tries, with index 6, is free and can be used. Deletions are more tricky than with chaining; if key2 and value2 were removed from the table in Figure 10.2 key3 and value3 would have to be moved down to take their place.

There are three constructors for IdentityHashMap:

```
public IdentityHashMap()
public IdentityHashMap( Map<? extends K,? extends V> m )
public IdentityHashMap(int expectedMaxSize )
```

The first two are the standard constructors that every general-purpose Map implementation has. The third takes the place of the two constructors which in other HashMaps allow the user to control the initial capacity of the table and the load factor at which it will be rehashed. IdentityHashMap doesn't allow this, fixing it instead (at .67 in the standard implementation) in order to protect the user from herself: whereas the cost of finding a key in a table using chaining is proportional to the load factor l, in a table using linear probing it is proportional to $1/(1-l)$. So avoiding high load factors is too important to be left to the programmer!

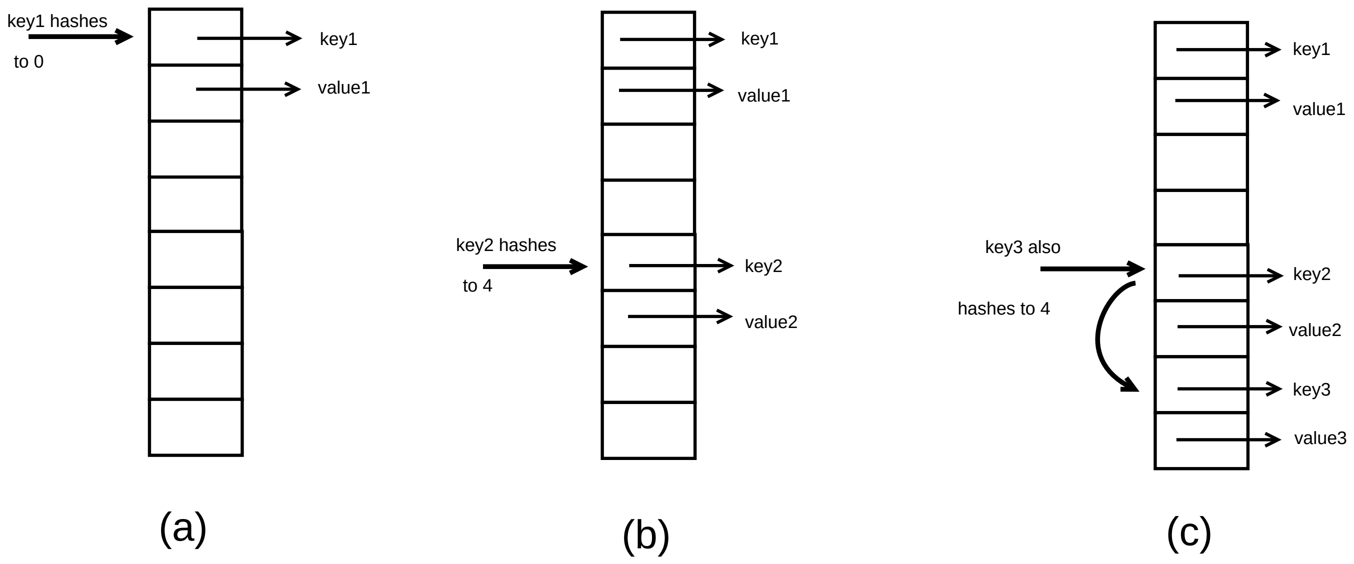


Figure 13.4: Resolving Collisions by Linear Probing

Out of all the Collection Framework hash implementations why does `IdentityHashMap` alone use linear probing when all the others use chaining? Again, it is to protect the programmer from herself. Linear probing has a better performance than chaining under some circumstances, but its performance deteriorates dramatically if the hashing algorithm being used doesn't distribute keys evenly over the table. `IdentityHashMap` is the only implementation for which the user can't supply the hashing algorithm—it uses `System.identityHashCode`, which gives the same hash code as would be returned by the `Object` method `hashCode`, whether or not that method is overridden in the object being hashed.

13.3 `SortedMap`

Like `SortedSet`, the subinterface `SortedMap` (see back endpaper) adds to its contract a guarantee that its iterators will traverse the map in ascending key order. Its definition is closely similar to that of `SortedSet`, with methods like `firstKey` and `headMap` corresponding to the `SortedSet` methods `first` and `headSet`. Again you can either use the natural ordering of the keys or impose a `Comparator` on them, but in any case the `compare` method must be consistent with `equals`, as the `SortedMap` will use it in its place.

The extra methods defined by the `SortedMap` interface fall into three groups:

Retrieving the comparator:

```
Comparator<? super K> comparator()
```

This method returns the map's key comparator if it has been given one rather than relying on the natural ordering of the keys.

Getting the head and tail elements:

```
K firstKey()
K lastKey()
```

If the set is empty, these operations throw a `NoSuchElementException`.

Finding subsequences:

```
SortedMap<K,V> subMap( K fromKey, K toKey )
SortedMap<K,V> headMap( K toKey )
SortedMap<K,V> tailMap( K fromKey )
```

These operations work like the corresponding operations in `SortedSet`: the arguments do not themselves have to be actual keys, and the sets returned include the `fromKey` if it is present in the map, and do not include the `toKey`.

To see how a `SortedMap` could be useful, consider the problem we looked at earlier, of processing all the tasks in the system, in descending order of priority. This was straightforward using a list of tasklists, because lists retain their ordering:

```
for( List<Task> taskList : taskLists  ) {
  for( Task t : taskList ) { ... }
}
```

but using an ordinary map we would have to get the ordering from the enum in which the priorities were defined:

```
for( Priority p : Priority.values() ) {
  List<task> taskList = taskMap.get( p );
  for( Task t : taskList ) { ... }
}
```

`SortedMap` can improve on this. We could use the ordering of the enum to construct it:

```
SortedMap<Priority,List<Task>> taskMap = new TreeMap<Priority,List<Task>>();
for( Priority p : Priority.values() ) {
    taskMap.put( p, new ArrayList<Task>() );
}
```

and then for the processing we could work through it using the same ordering, which it retains:

```
for( List<Task> taskList : taskMap.keySet() ) {
  for( Task t : taskList ) { ... }
}
```

13.3.1 TreeMap

SortedMap is implemented in the Collections Framework by TreeMap. We have already met trees as a data structure for storing elements in order when we discussed TreeSet (Section 10.2.1). In fact TreeSet is just a TreeMap with a standard value for all keys, so the explanation of the mechanism and performance of red-black trees given in Section 10.2.1 applies equally here.

The constructors for TreeMap include, beside the standard ones, one which allows you to supply a Comparator and one which allows you to create one from another SortedMap, using both the same comparator and the same mappings.

13.4 ConcurrentMap

Maps are often used in high-performance server applications, for example as cache implementations. Such applications are typically multi-threaded, to take advantage of multiple-CPU architectures, so there is a need for thread-safe Map implementations. In one sense this is quite easy to achieve; we will see in the next section that the Collections Framework provides a mechanism for wrapping a collection in a thread-safe wrapper so that only one thread at a time may operate on it. The drawback to this for an application running many threads is that the synchronized map becomes a bottleneck for which the threads have to queue, so removing one of the main advantages of having multiple threads. And even that may not be enough: suppose that an application wants to avoid ever overwriting an existing value in the map. Simply writing

```
if ( ! map.containsKey( key )) {    //1
  map.put( key, value );            //2
}
```

won't always work even if the map is synchronized, because immediately after line //1 is executed a different thread might intervene to put the key, and now our thread will go on in line //2 to overwrite the value that it has placed there. This is an example of conditional thread-safety, discussed in Section 8.6.

The interface ConcurrentMap addresses this problem by providing declarations for methods which are only conditionally thread-safe in the synchronized versions of ordinary Map implementations. There are four of these methods, which must perform atomically in ConcurrentMap implementations:

```
V putIfAbsent(K key, V value)
          // associate key with value only if key is not currently present
boolean remove(Object key, Object value)
          // remove key only if it is currently mapped to value.
V replace(K key, V value)
          // replace entry for key only if it is currently present
boolean replace(K key, V oldValue, V newValue)
          // Replace entry for key only if it is currently mapped to oldValue
```

13.4.1 ConcurrentHashMap

ConcurrentHashMap provides an implementation of ConcurrentMap and also offers a solution to the problem of reconciling throughput with thread safety. Retrievals usually do not block, even while the table is being updated (to allow for this, the contract states that the results of retrievals will reflect the most recently completed update operations holding upon their onset). Updates also can often proceed without blocking, because a `ConcurrentHashMap` consists of not one but a set of tables, called *segments*, each of which can be independently locked. If the number of segments is large enough relative to the number of threads accessing the table, there will be often be no more than one update in progress per segment at any time. It is never necessary—or even possible, for the client programmer—to lock the table as a whole.

ConcurrentHashMap is a useful implementation of `Map` in any application where it is not necessary to lock the entire table. But it is really indispensable in highly concurrent contexts, where it performs far better than a synchronized `HashMap`

13.5 Comparison of `Map` Implementations

Todo: provide table of map implementations

Chapter 14

Special Implementations

14.1 Pre-populated Collections

The `Collections` class provides convenient ways of creating some kinds of collections prepopulated with zero or more references to the same object. The simplest possible collections are empty:

```
static <T> List<T> emptyList()     // Returns the empty list (immutable)
static <K,V> Map<K,V> emptyMap()   // Returns the empty map (immutable)
static <T> Set<T> emptySet()       // Returns the empty set (immutable)
```

Empty collections can be useful in implementing methods to return collections of values; they can be used to signify that there were no values to return. If you have much occasion to use them, they can save you a good deal of unnecessary object creation. These methods replace the static `Collections` fields `EMPTY_SET`, `EMPTY_LIST`, and `EMPTY_MAP`, which were commonly used before Java 5 but are less useful now because their raw types generate unchecked warnings whenever they are used.

The `Collections` class also provides you with ways of creating collections objects containing only a single member:

```
static <T> Set<T> singleton(T o)
          // Returns an immutable set containing only the specified object.
static <T> List<T> singletonList(T o)
          // Returns an immutable list containing only the specified object.
static <K,V> Map<K,V> singletonMap(K key, V value)
          // Returns an immutable map, mapping only the key K to the value V.
```

Again these can be useful in providing a single input value to a method that is written to accept a Collection of values.

Finally, it is possible to create a list containing an arbitrary number of object references:

```
static <T> List<T> nCopies(int n, T o)
          // Returns an immutable list containing n references to the object o
```

Although the list produced by nCopies is itself immutable—in fact, like the class `Triples` we showed in Section 12.2.1, it physically contains only a single element—the view that it provides can be used as the basis for building further collections, for example by providing it as the argument to a constructor or an `addAll` method.

14.2 Decorated Collections

The `Collections` class provides ways of modifying the behaviour of standard collections classes in three ways—by making them unmodifiable or dynamically type-safe, or by synchronizing them. It does this by creating specialised objects which mostly delegate their real work to the standard collection object, but modify the way in which it is done. This is an example of the use of the *decorator pattern* (Gamma et. al., *Design Patterns*), which provides an alternative to inheritance when that would result in an explosion of closely related subclasses. Implementing the decorator pattern normally involves creating a decorator class which implements the modifications needed to the behaviour of the collection; here, the decorator classes are hidden inner classes of the `Collections` class, which instead of exposing them provides factory methods to return instances of them.

14.2.1 Unmodifiable Collections

An unmodifiable collection will throw `UnsupportedOperationException` in response to any attempt to change its structure or the elements which compose it. This can be useful when you want to allow clients read access to an internal data structure. Passing the structure in an unmodifiable wrapper will prevent a client from changing it—provided, of course, that it does not contain references to modifiable objects. (If that is in fact the case, you may have to protect your internal data structure by providing clients instead with a *defensive copy* made for the purpose).

There are six unmodifiable wrapper factory methods, corresponding to the six major interfaces of the Collections Framework:

```
public static <T> Collection<T>
    unmodifiableCollection(Collection<? extends T> c);
public static <T> Set<T> unmodifiableSet(Set<? extends T> s);
public static <T> List<T>
    unmodifiableList(List<? extends T> list);
public static <K, V> Map<K, V>
    unmodifiableMap(Map<? extends K, ? extends V> m);
public static <T> SortedSet<T>
    unmodifiableSortedSet(SortedSet<? extends T> s);
public static <K, V> SortedMap<K, V>
    unmodifiableSortedMap(SortedMap<K, ? extends V> m);
```

14.2.2 Synchronized Collections

As we explained in Section 8.6, most of the Framework classes are not thread-safe—by design—in order to avoid the overhead of unnecessary synchronization as incurred by the legacy classes `Vector` and `Hashtable`. But there are occasions when you do need to program multiple threads to have access to the same collection, and these synchronized wrappers are provided by the `Collections` class for such situations. Again there is a factory method for each of the major Framework interfaces:

```
public static <T> Collection<T>
    synchronizedCollection(Collection<T> c);
public static <T> Set<T> synchronizedSet(Set<T> s);
public static <T> List<T> synchronizedList(List<T> list);
public static <K, V> Map<K, V> synchronizedMap(Map<K, V> m);
public static <T> SortedSet<T>
    synchronizedSortedSet(SortedSet<T> s);
public static <K, V> SortedMap<K, V>
    synchronizedSortedMap(SortedMap<K, V> m);
```

The decorator classes that provide these synchronized views are conditionally thread-safe (Section 8.6); although each of their operations is guaranteed to be atomic, you may need to synchronize multiple method calls in order to obtain consistent behaviour. In particular, iterators must be created and used entirely within a code block synchronized on the collection, otherwise the result will at best be failure with `ConcurrentModificationException`. This is very coarse-grained synchronization; if your application makes heavy use of synchronized collections, its effective concurrency will be greatly reduced.

14.2.3 Checked Collections

Unchecked warnings are a signal to us to take special care to avoid run-time type violations. But it may be beyond our power to ensure that they never occur; for example, after we have passed a typed collection reference to an ungenerified library method, we can't be sure that it has added only correctly typed elements to the collection. Rather than lose confidence in the collection's type safety we can pass in a checked wrapper, which will explicitly cast every element added to the collection into the type supplied when it is created. Checked wrappers are supplied for the main interfaces:

```
public static <E> Collection
    checkedCollection(Collection<E> c, Class<E> elementType)
public static <E> List
    checkedList(List<E> c, Class<E> elementType)
public static <E> Set
    checkedSet(Set<E> c, Class<E> elementType)
public static <E> SortedSet
    checkedSortedSet(SortedSet<E> c, Class<E> elementType)
public static <K, V> Map
    checkedMap(Map<K, V> c, Class<K> keyType, Class<V> valueType)
public static <K, V> SortedMap
    checkedSortedMap(SortedMap<K, V> c, Class<K> keyType,Class<V> valueType)
```

A related use of the checked wrappers is to ensure that a client has not smuggled a wrongly typed element into a typed collection you are being supplied with. Suppose, for example, that a library class receives a collection of objects representing secure channels which it will use for transmitting information. To ensure that every element really has the right type, it could create a new empty checked collection, and add all the elements of the suspect collection to it.

```
public void transmitSensitiveInformation( List<SecureChannel> list ) {
  List<SecureChannel> secureList =
    Collections.checkedList( new ArrayList<SecureChannel>(), SecureChannel.class );
  secureList.addAll( list );
  ...
  // from now on use only secureList
}
```

Like the other decorated collections, checked wrappers have fairly severe limitations. Since the check takes place at runtime, only runtime type information can be used, so the only types that can be fully checked are reifiable ones. By contrast, the best that you can do by way of run-time type checking for a collection of elements of a parameterised type is to create a wrapper that will check that each added element belongs to the corresponding raw type. (This partial solution is made even more unsatisfactory by the fact that the checked wrapper will itself be a collection of the raw type, so you won't be able to use it without getting unchecked warnings).

Todo: add section on run-time type checking

Chapter 15

Generic Algorithms

The Collections Framework supplies a useful set of generic algorithms in the class `Collections`. Generic algorithms represent reusable functionality; they can be applied to `List`s (or in some cases to `Collection`s) of any type. Generifying the types of these methods has led to some fairly complicated declarations, so each section discusses the declarations briefly after presenting them. Part III gives a fuller motivation for these and other design decisions taken in generifying the Collections Framework.

The generic algorithms fall into four major categories: changing element order in lists, finding extreme values in a collection, searching a list, and systematically changing the contents of a list.

15.1 Changing the Order of `List` Elements

`swap` is the simplest of these methods; it executes in constant time. `sort` is the most complex; it transfers the elements into an array, applies a merge sort to them in time $O(n \log n)$, and then returns them to the `List`. The remaining methods all execute in time $O(n)$, although the coefficient will depend on whether the `List` should be processed using sequential or random access. Each method selects between two algorithms based both on whether the `List` implements the interface `RandomAccess`, and also on its size—for small sequential access lists the random access algorithms are faster.

```
static void reverse(List<?> list)
          // reverses the order of the elements
static void rotate(List<?> list, int distance)
          // rotates the elements of the list; the element at index
          // i is moved to index (distance + i) % list.size()
static void shuffle(List<?> list)
          // randomly permutes the list elements
static void shuffle(List<?> list, Random rnd)
          // randomly permutes the list using the randomness source rnd
static <T extends Comparable<? super T>> void sort(List<T> list)
          // sorts the supplied list using natural ordering
static <T> void sort(List<T> list, Comparator<? super T> c)
          // sorts the supplied list T using the supplied ordering
static void swap(List<?> list, int i, int j)
          // swaps the elements at the specified positions
```

15.2 Finding Extreme Values in a `Collection`

`min` and `max` are supplied for this purpose, each with two overloads—one using natural ordering of the elements, and one accepting a `Comparator` to impose an ordering. They execute in $O(n)$ time.

```
static <T extends Object & Comparable<? super T>> T
      max(Collection<? extends T> coll)
            // returns the maximum element using natural ordering
static <T> T max(Collection<? extends T> coll, Comparator<? super T> comp)
            // returns the maximum element using the supplied comparator
static <T extends Object & Comparable<? super T>> T
      min(Collection<? extends T> coll)
            // returns the minimum element using natural ordering
static <T> T min(Collection<? extends T> coll, Comparator<? super T> comp)
            // returns the maximum element using the supplied comparator
```

Why are these declarations so unwieldy? In the case of the overloads that use natural ordering, the reason for the complexity of the return type lies in the need to maintain compatibility with code that makes calls to the older, non-generified, version. The older version returns `Object`, so that has to be the return type of the erasure of the generified version too.

In the case of the overloads that accept a `Comparator`, the types are chosen to make it clear that the method can be applied to any `Collection`-`Comparator` pair where the type parameter of the `Collection` is a subtype of the type parameter of the `Comparator` (see ????).

15.3 Finding Specific Values in a `List`

Methods in this group locate elements or groups of elements in a `List`, again choosing between alternative algorithms on the basis of the list's size and whether it implements `RandomAccess`.

Todo: Explain binary search, give O for sequential access list

```
static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key)
          // searches for key using binary search
static <T> int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)
          // searches for key using binary search
static int indexOfSubList(List<?> source, List<?> target)
          // finds the first sublist of source which matches target
static int lastIndexOfSubList(List<?> source, List<?> target)
          // finds the last sublist of source which matches target
```

The signature of the first `binarySearch` overload says that you can use it to search for a key of type `T` in a list of objects which can have any type—provided they can be compared with objects of type `T` or a supertype of `T`. The second is like the `Comparator` overloads of `min` and `max`, except that in this case the type parameter of the `Collection` must be a subtype of the type of the key, which in turn must be a subtype of the type parameter of the `Comparator`.

The signatures of `indexOfSubList` and `lastIndexOfSubList` allow the source and target lists to be of any kind (remember that the two wildcards signify unrelated types). No assumptions need to be made about the types of the list elements; they are simply going to be compared using `equals`.

15.4 Changing the Contents of a `List`

These methods change some or all of the elements of a list. `copy` transfers elements from the source list into an initial sublist of the destination list (which has to be long enough to accomodate them), leaving any remaining elements of the destination list unchanged. `replaceAll` replaces every occurrence of one value in a list with another—where either the old or new value can be null—returning `true` if any replacements were made. `fill` replaces every element of a list with a specified object.

```
static <T> void copy(List<? super T> dest, List<? extends T> src)
          // copies all of the elements from one list into another
static <T> void fill(List<? super T> list, T obj)
          // replaces every element of list with obj
static <T> boolean replaceAll(List<T> list, T oldVal, T newVal)
          // replaces all occurrences of oldVal in list with newVal
```

The signatures of these methods can all be explained using the Get and Put Principle. `copy` was discussed in Part I. It gets elements from the source list and put them into the destination, so the types of these lists are respectively `? extends T` and `? super T`. This fits with the intuition that the type of the source list elements should be a subtype of the destination list. For `fill`, the Get and Put Principle dictates that you should use `extends` if you are putting values in to a parameterised collection, and for `replaceAll`, it states that if you are putting values into and getting values out of the same structure you should not use wildcards at all.

Part III

Using Generics Effectively

Chapter 16

Style and Technique in Generic Programming

16.1 Avoid Unchecked Warnings

One of the greatest advantages of using generics is that the programmer can omit many casts that were previously necessary, safe in the knowledge that exceptions that could previously have been raised by these casts can never now occur. But this guarantee is fenced with a proviso: for it to apply, compilation must have produced no unchecked warnings. By default, unchecked warnings are only reported in a general way by the compiler: without specifically enabling them the compiler will only report something like

```
Note: MyClass.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Supplying the option `-Xlint:unchecked` will produce detailed information about the sources of type insecurity in a program. All Java 5 compilations should use this this option.

Of course, the presence of unchecked warnings doesn't actually tell you that a program is not typesafe; it tells you is that the compiler has been unable to determine the opposite, so you can no longer rely on the guarantee that you will see no casting exceptions at runtime. If you are really confident that you understand your program better than the compiler does, you can afford to ignore or suppress warnings (see Sections 17.2 and 16.2). Such situations are very much the exception, however: unless your program interfaces with legacy code, unchecked warnings should almost always be taken as an indication that a program is unsound. Warnings can arise from a variety of common errors.

For example, a common beginner's mistake is to forget to provide a type parameter when calling the constructor of a generic type. This will result in a raw type.

```
// omitting type parameter from instance creation - not recommended
List l = new ArrayList();
// ...
l.add("abc");   // unchecked call
```

This can be a difficult mistake to track at first, because the unchecked warning is attached not to the use of the unparameterized constructor, but to later calls of methods on the raw type.

```
MyClass.java:19: warning: [unchecked] unchecked call to add(E)
as a member of the raw type java.util.List
```

An “unchecked call” warning is issued when you call a method on a raw instance of a parameterised type, as in this case. When it is the argument that is provided raw in place of an expected parameterised one, a different warning is given; in that case the compiler warns of “unchecked method invocation”. This can appear in the one context in which raw types may be unavoidable, when they are returned from methods of ungenerified legacy code. For example the Hibernate method `Query.list` returns the results of a database query as a raw `List` of results, which you might then want to sort in natural order:

```
   List trackList = query.list();
   Collections.sort( trackList );    // unchecked method invocation
```

Of course, the compiler will report that `Collections.sort` expects an instantiated `List` type:

```
HibernateTest.java:62: warning: [unchecked] unchecked method invocation:
<T>sort(java.util.List<T>) in java.util.Collections is applied to
(java.util.List)
```

In this case the unchecked warning cannot be entirely avoided, but the occurrences should be localised as far as possible, as recommended in Section 17.2.

Because unchecked warnings are just an indication of the compiler’s failure to prove type-safety they can sometimes appear in surprising situations, including those for which there is in fact complete type information. For example, in the following code the result of the call to `getList` can be unambiguously typed as `List<String>`:

```
   // mixing raw and generic types - not recommended
   class Mixed<T> {
     public List<String> getList() { /*...*/ }
     public static void main(String[] args) {
       Mixed raw = new Mixed();
       List<String> listString = raw.getList(); // unchecked conversion
     }
   }
```

but the compiler issues a warning:

```
Mixed.java:6: warning: [unchecked] unchecked conversion
found   : java.util.List
required: java.util.List<java.lang.String>
```

This happens because calculating the type of a non-inherited member of the raw type `Mixed` is too complicated in general (even though it may be obvious in a simple case like the one above); the most reliable way for the compiler to handle it is to erase the types of the members of `Mixed` completely. Although in this case the warning is spurious, it should not be ignored; the use of the raw type will cause problems elsewhere, and eliminating it will also remove the spurious warning.

A second cause of unchecked warnings is casting to a parameterized type or a type parameter. Because such casts, which take place at run-time, do not have access to compile-time type information, they can never be determined by the compiler to be type-safe. There are only a few occasions, listed in Section 16.2, when unsafe casts are unavoidable. The most common case in which they can and should be avoided is again in the context of interfacing with legacy code. A beginner’s ”solution” to the problem set by the Hibernate example above is to cast the returned `List` type:

```
// unsafe unchecked cast - not recommended
List<Track> trackList = (List<Track>)query.list();
Collections.sort( trackList );
```

This avoids the warning on the call of `sort` at the cost of an unchecked warning about the cast. The correct solution is again to apply the methods of Section 17.2.

A third situation in which unchecked warnings can be generated was described in Section 5.8, on the use of varargs to create an array of non-reifiable type:

```
// creating array of non-reifiable type - not recommended
class VarArgsProblem {
  public <T> List<T>[] createArray ( List<T>... lsa ) { return lsa; }
  public static void main(String[] args) {
    List<String>[] lsa =
          new VarArgsProblem().createArray(new ArrayList<String>());
  }
}
```

This causes the warning

```
VarArgsProblem.java:5: warning: [unchecked] unchecked generic array
creation of type java.util.List<java.lang.String>[] for varargs parameter
```

If you recall from Section 5.8 the problems that can arise from generic array creation, you will probably think that this warning should in fact be an error. The bottom line is, don't use varargs with non-reified types!

16.2 Make Careful Use of `@SuppressWarnings`

Section 16.1 advised that unchecked warnings should usually be treated as a reliable guide to type insecurities in generic code. There are a few occasions, however, when you can be really confident that you do know more than the compiler, and that you can safely ignore its warnings. The most common cause of unchecked warnings is of course the interaction of generified clients with legacy library code. Chapter 4 described ways of avoiding warnings from compilation of client code in this situation, but if these are not practicable then unchecked warnings may be a fact of life for you—at least for a while, until your libraries have been generified. One way to handle these warnings is to provide wrappers for legacy code as described in Section 17.2; that technique is a special case of the advice of this section: don't allow generation of unchecked warnings to become the norm for compilations of your code. If unchecked warnings are really unavoidable, use the annotation `@SuppressWarnings` to prevent yourself—and others—from becoming used to seeing them.

There are some other situations in which unchecked warnings are unavoidable. They include cloning (Section 16.5), list promotion (Section 5.2), and the creation of arrays of generic types either for internal use as in the implementation of `ArrayList` (Section 5.7) or for export as in `Collection.toArray` (Section 5.5). In this section we'll use the `clone` method as an example. For the reasons explained in Section 16.5, in a parameterized type is bound to contain an unchecked cast in a line like

```
CloneableClass<T> cloneResult =
    (CloneableClass<T>) super.clone(); // unchecked cast
```

To suppress this warning you should annotate the `clone` method with `@SuppressWarnings`:

```
@SuppressWarnings("unchecked")
public CloneableClass<T> clone() {
  // ...
}
```

Note that at the time of writing, `@SuppressWarnings` is not supported and it seems possible that it may not be supported until Java 6 is released. In any case it is a rather blunt instrument; annotations can be attached only to declarations, meaning that you cannot use `@SuppressWarnings` to specify individual lines of code that you know to be safe. Annotation by hand is therefore an important supplement to `@SuppressWarnings`:

```
@SuppressWarnings("unchecked")
public CloneableClass<T> clone() {
  // ...
  CloneableClass<T> cloneResult =
          (CloneableClass<T>) super.clone(); //unchecked cast
  // ...
}
```

We recommend that the `//unchecked...` annotation should be placed on the code line that the compiler will warn about (rather than in an explanation on the preceding line, say). This is an important aid to maintenance: if `@SuppressWarnings` is not operating (either because of its non-implementation in Java 5 or, later, because a maintenance programmer has removed it temporarily to allow checking of changes to the program), the warnings seen by the maintenance programmer look like this:

```
CloneableClass.java:37: warning: [unchecked] unchecked cast
found   : java.lang.Object
required: items.SuppressWarnings1<T>
        CloneableClass<T> cloneResult =
            (CloneableClass<T>) super.clone();  //unchecked cast
```

which immediately conveys the information that this warning is expected. For greatest clarity, you should also give the specific type of the unchecked warning—cast, call, method invocation or conversion—although where brevity is needed to help shorten line lengths, `//unchecked` alone can be used.

16.3 Favour Collections Over Arrays

Arrays are traditionally the data structure of choice in Java—as in most other programming languages designed during the last half-century—so for many programmers the advice that we offer, to avoid arrays, may be quite challenging. So it is worth emphasizing: **arrays in Java are now inferior to collections in nearly every respect**. Moreover, their covariant typing is incompatible with the invariant scheme used for generics, leading to the necessity of rules like the Principle of Indecent Exposure and the possibility of mistakes caused by contravening them.

Let's go over the main points of comparison.

- *Flexibility* Comparing arrays with collections is not comparing like with like. An array is a linear data structure, natively providing only the ability to get or set elements. Collections, by contrast, natively implement the various operations of the `Set`, `List`, and `Queue` interfaces, with numerous implementations tailored to different requirements. True, the usability of arrays is increased by the convenience class `Arrays`, but even there the set of operations that it provides is much smaller than that provided by the corresponding `Collections` class.

- *Resizeability* A major disadvantage of arrays is the necessity for the programmer to handle array-full conditions explicitly. For most collections, except those which whose size is fixed by design, the only limit in practice is that imposed by heap memory.

- *Typing Scheme* The covariant typing used for arrays was introduced only to support strongly-typed declarations for the polymorphic methods of the class `Arrays`. Its disadvantages include the runtime overheads of array type checking and the associated possibility of runtime `ArrayStoreException`s. The invariant type scheme of generics is more flexible and powerful, and overall a great improvement on it.

- *Syntax* Before Java 5, the iteration syntax for arrays was preferable to the necessity of explicitly using an `Iterator`. Now, with foreach, iteration syntax is uniform. The one remaining syntax advantage held by arrays is that the suffix notation is still preferable to using the `get` and `set` methods of `List`.

- *Primitive Types* Arrays retain an advantage for handling collections of primitive types. Collections can only handle primitives by boxing each value, with a substantial performance overhead, and the possibility of `ArrayStoreException` at run-time doesn't arise with primitive values.

- *Efficiency* Compared to `List`s, arrays perform some operations more efficiently—for example, `ArrayList` accesses incur an extra bounds check against the size of the list, so occasions may arise when you need to use arrays for maximum performance. But these will be uncommon, and the usual rule about optimization will apply—don't do it, at least not until you can test performance gains against a working version of your system.

16.4 Favour Wildcards over Type Parameters for Generic Methods

When you are considering the signature of methods with generic types, there is often a choice between using wildcards and type parameters. Which should you use? There is often a simple way to decide; compare, for example, two different overloads of the method `addObjectValue` in `javax.imageio.metadata.IIOMetadataFormatImpl`. Before generification the declarations were

```
protected void addObjectValue(String elementName,
      Class classType, boolean required, Object defaultValue)
protected void addObjectValue(String elementName,
      Class classType, int arrayMinLength, int arrayMaxLength)
```

The use of the raw `Class` type is an obvious pointer towards generification. To see precisely what type parameter `Class` should take, however, you have to look at the rest of the method signature. Both overloads allow a reference to be stored in the receiver, with the given element name; the first accepts a reference to any `Object` together with default value for it. the second allows a reference to an array of the supplied class (within length constraints supplied by the other parameters).

Considering the first overload carefully you see that the types of the arguments are related: the default value should belong to a subtype of the class type. So its generification should be

```
protected <T> void addObjectValue(String elementName,
      Class<T> classType, boolean required, T defaultValue)
```

where the type parameter `T` ties the argument types together in the right way.

There is no such relationship between the parameters to the second overload. Without wildcards its generic form would be

```
protected <T> void addObjectValue(String elementName,
       Class<T> classType, int arrayMinLength, int arrayMaxLength)
```

A programmer seeing the type of the `Class` argument must check the rest of the declaration for other occurrences. But since there are none, we can save her a valuable moment by replacing `T` in the declaration with a wildcard; seeing the `classType` declaration in this variant she will immediately know that there are no constraints on it at all.

```
protected void addObjectValue(String elementName,
       Class<?> classType, int arrayMinLength, int arrayMaxLength)
```

Declarations containing more than one parameterized type also need to be examined carefully to see whether wildcards or type parameters are more appropriate. We have seen an example where the decision was relatively easy; now let's look at a judgement call.

The class `javax.swing.SpinnerDateModel` represents a sequence of dates to be shown on a visual spinner control. One of its constructors is declared as:

```
public SpinnerDateModel(Date value,
    Comparable start, Comparable end, int calendarField)
```

where value is the currently chosen element of the sequence, the value of `calendarField` dictates the type of the value—for example, day, month or year—and `start` and `end` represent the first and last dates of the sequence. `Comparable` was used as the type of these parameters rather than `Date`, because the procedure of constructing `Date` objects with specific values is quite roundabout, requiring you to first create an instance of a `Calendar`, then calling its `getTime` method. It is simpler and clearer to define an specialised class:

```
class FixedStartDate implements Comparable<Date> {
  long t = ...  // start date in milliseconds after midnight, January 1 1970
  public int compareTo(Date d) {
    return (t < d.getTime() ? -1 : (t == d.getTime() ? 0 : 1));
  }
}
```

How should `SpinnerDateModel` be generified? The question here is whether `start` and `end` should be constrained to have the same type. If we think they should, the declaration of the constructor should be

```
public <T extends Comparable<? super Date>> SpinnerDateModel(
      Date value, T start, T end, int calendarField )
```

Note this unusual example of a parameterized constructor. On the other hand, if we think that it would be reasonable to provide an instance of `FixedStartDate` as a start date, say, and an ordinary `Date` object as the end date, then we should write the declaration as

```
public SpinnerDateModel(Date value, Comparable<? super Date> start,
      Comparable<? super Date> end, int calendarField )
```

In fact, `SpinnerDateModel` is not yet generified in Java 5, but it is likely that the alternative using wildcards will be chosen, both because it is less unnecessarily restrictive and also because once again it is clearer and easier to read. You have already met (Sections 2.3 and 15.3) the `Collections` method `copy`, whose pre-generics declaration was

```
public static void copy(List dest, List src){/*...*/}
```

Various choices involving combinations of wildcards and type parameters were available to generify this declaration. The simples relationship that can be imposed between the source and the destination lists is that their elements must have the same type:

```
public static <T> void copy(List<T> dest, List<T> src){/*...*/}
```

but this is more restrictive than necessary; the destination list need only be able to accommodate any supertype of the source list elements. This is in line with an important principle of generification; use bounded types where possible. Using type parameters to express that relationship, the declaration becomes:

```
public static <T, S extends T> void copy(List<T> dest,
      List<S> src){/*...*/}
```

We can simplify this by noting that, as in the first example in this section, where a type (`S` in this case) occurs only once it can always be replaced by a wildcard. Eliminating `S` improves the readability of the declaration as before:

```
public static <T> void copy(List<T> dest,
      List<? extends T> src){/*...*/}
```

We saw before, though, that the declaration of copy is different from this; it is actually

```
public static <T> void copy(List<? super T> dest,
      List<? extends T> src){/*...*/}
```

Why is this (more verbose) declaration better? Because it is more expressive: the Get and Put Principle tells the client programmer that `copy` intends only to get elements from the source list, and only to put elements into the destination list. This expression of intention is of course not a substitute for the contract; in fact it is not even a guarantee that `copy` will keep to these intentions—see Section 16.6—but it does immediately provides the client programmer with useful information which would take longer to extract from the contract.

16.5 Use Generics for Typesafe Covariant Hierarchies

Hierarchies of types often run in parallel. A class or interface `A` has a method `M` with an argument of type `X`; `A` has a subtype `B`, which overrides `M` to return `Y`, which is a subtype of `X` (as in Figure 16.1). This common situation cannot be specified in a typesafe way without generics. For example, the AWT contains some layout managers (those implementing `LayoutManager2`) which can position each visual component in a container according to a supplied constraint:

$$\begin{tikzcd}
A \arrow[r, "uses"] & X \\
B \arrow[u] \arrow[r, "uses"] & Y \arrow[u]
\end{tikzcd}$$

Figure 16.1: Simple Covariant Hierarchy

```
public interface LayoutManager2 extends LayoutManager {...
  void addLayoutComponent(Component comp, Object constraints);
  // ...
}
```

The simplest constraints are `int` constants used, for example, to represent the five regions of a `BorderLayout`. A more complex layout manager, `GridBagLayout`, requires a more complex set of constraints for each component, encapsulated in a `GridBagConstraints` object. `GridBagLayout.addLayoutComponent` must check that the supplied constraint has the right type, and if so cast it:

```
public class GridBagLayout implements LayoutManager2, Serializable {
  public void addLayoutComponent (Component comp, Object constraints) {
    if (constraints instanceof GridBagConstraints) {
      setConstraints(comp, (GridBagConstraints)constraints);
    } else if (constraints != null) {
      throw new IllegalArgumentException( /* ... */ );
    }
  }
  // ...
}
```

If you face a similar situation, you can use generics to avoid this unpleasant problem. If the supertype is parameterized on the type of the method argument (A, B, etc?), the signature of the method in the instantiated subtype contains exactly the right type arguments:

```
public interface LayoutManager2<C> extends LayoutManager {
  void addLayoutComponent(Component comp, C constraints);
  // ...
}
public class GridBagLayout implements LayoutManager2<GridBagConstraints>, Serializable {
  public void addLayoutComponent (Component comp, GridBagConstraints constraints) {
    // no instance test necessary
    setConstraints(comp, constraints);
  }
  // ...
}
```

It's too late for `LayoutManager2` implementations to be changed in this way, unfortunately, because they would not maintain compatibility with old subclasses (see Section 17.4). The interface itself could be changed for future implementations, though.

Java 5 actually introduced covariance for method overriding in two different ways: for parameter types through the use of generics, as above, but also—by means of a language change independent of generics—for return types. The classic need for these is in the `clone` method which, although it returns a reference to the type of the receiver, is declared in `Object` as

```
protected Object clone();
```

and whose clients therefore usually need to cast its result to get a reference of the right type. With Java 5, `clone` implementations can remove the need for this cast by returning the right type. Although existing implementations can't change to take advantage of this without breaking their clients (see Section 17.4), new ones can. For example:

```
class IntPair implements Cloneable {
  private int first, second;
  public IntPair clone() throws CloneNotSupportedException {
    IntPair clone = (IntPair) super.clone();
    clone.first = first;
    clone.second = second;
    return clone;
  }
}
```

And this works for parameterized classes too, so long as you want only a shallow clone—one which can share its fields with its clone-origin:

```
class ShallowCloneablePair<T> implements Cloneable {
  private T first, second;
  public ShallowCloneablePair<T> clone()
        throws CloneNotSupportedException {
    ShallowCloneablePair<T> clone =
        (ShallowCloneablePair<T>) super.clone();    // unchecked cast
    clone.first = first;
    clone.second = second;
    return clone;
  }
}
```

The cast now gives an unchecked warning, but that is unavoidable in cloning parameterized types. You can suppress it (see Section 16.2) with confidence, since you know—as the compiler cannot know—that it is safe to assign this type to the clone, and that doing so will allow the `clone` method to have the most precise return type possible.

Deep cloning is more of a problem. For this each mutable field has to be cloned, but since you can't call the `clone` method of an arbitrary type `T` directly (because `clone` has protected access in `Object`, the erasure of `T`), you must instead discover and call `clone` by reflection

```
class DeepCloneablePair<T> implements Cloneable {
  private T first, second;
  public DeepCloneablePair<T> clone() throws CloneNotSupportedException {
    DeepCloneablePair<T> pt = (DeepCloneablePair<T>) super.clone();
    Class<?> firstClass = first.getClass();
    try {
      Method method = firstClass.getMethod("clone", new Class[0]);
      pt.first = (T)method.invoke(first, new Object[0]); // unchecked cast
    } catch (Exception e) {
      throw new CloneNotSupportedException(firstClass.getName());
    }
    // ... repeat for second field
    return pt;
  }
}
```

This code will fail if the structure to be deeply cloned contains any non-cloneable types. In the case of immutable components, these failures are unnecessary because it is safe to share them. Although we can't test for immutability in an arbitrary type, we could at least eliminate the cause of the commonest unnecessary failures, `Number` subtypes and `String`, by testing for these types before trying to use reflection:

```
if ( firstClass == java.lang.String.class ||
      java.lang.Number.class.isAssignableFrom(firstClass ) &&
          // a client could have extended Number with a mutable subclass
          firstClass.getPackage() == Package.getPackage( "java.lang")){
  pt.first = first;
} else {
  // ... try to discover clone method by reflection, as above
}
```

By now you must be wondering whether it wouldn't be better to abandon `clone` and `Cloneable`, and start over. It certainly would be if we really could start over completely, especially with the help of generics. We could define an ordinary interface with a public method (both named to be distinct from current usage):

```
interface Kloneable<T> {
  public T klone() throws CloneNotSupportedException;
}
```

and implementing the method would be both clean and typesafe—only structures whose components are kloneable are themselves kloneable:

```
class DeepKloneablePair<T extends Kloneable<T>>
      implements Kloneable<DeepKloneablePair<T>> {
  private T first, second;
  public DeepKloneablePair<T> klone() throws CloneNotSupportedException {
    DeepKloneablePair<T> pt = new DeepKloneablePair<T>();
    pt.first = (T)first.klone();
    pt.second = (T)second.klone();
    return pt;
  }
}
```

A more realistic example is a kloneable collection:

```
class KloneableArrayList<E extends Kloneable<E>> extends ArrayList<E>
    implements Kloneable<KloneableArrayList<E>> {
  public KloneableArrayList() { super(); }
  public KloneableArrayList(Collection<? extends E> c) { super(c); }
  public KloneableArrayList(int i) { super(i); }
  public KloneableArrayList<E> klone() throws CloneNotSupportedException {
    KloneableArrayList<E> list = new KloneableArrayList<E>();
    for (E elt : this) list.add(elt.klone());
    return list;
  }
}
```

Kloneability cannot easily be grafted on to existing code: you have to choose the superclass of new kloneable classes with care, as a kloneable class must be able to initialise the superclass fields of its clone, either directly or via mutators. And you may have difficulty if the components of your kloneable class belong to existing final classes unless these too implement `Kloneable`.

16.6 Don't Use Generics to Enforce Immutability

There are various reasons to favour immutable class designs (see Bloch) but, despite exhaustive discussion in the developer forums, Java provides no way of enforcing immutability. It is therefore tempting to seize on any new possibility for enforcement, but in the case of generics this is a temptation to resist; the Get and Put Principle provides a useful indication of the programmer's intentions but only the weakest of safeguards against mutation. Consider a class designed on this principle, exposing a data structure that the programmer intends should be preserved from outside interference (since `Integer` is final, the intention is unambiguous):

```
public class MyClass {
  private List<Integer> intList = ...
  public List<? extends Integer> getIntList() {
    return intList;
  }
  ...
}
```

Client code, if it is well-typed, cannot get a more precise type than `List<? extends Integer>` by calling `getIntList`:

```
List<? extends Integer> listRef = new MyClass().getIntList();
```

so the Get and Put Principle tells us that it cannot supply `Integer` references to the list by calling any method on `listRef` with a `T`-typed argument—for example, `add` or `set`. That certainly gives an indication of the intentions of the class designer. This technique is used sometimes in the core API; for example, `java.awt.Font` includes the method declaration

```
public Map<TextAttribute,?> getAttributes()
```

where the attribute values of the returned map can belong to a variety of different classes whose only common superclass is `Object`. The returned map is a copy of the font's internal attribute map, so putting values in it will normally be an error, and the declaration expresses that fact by making it harder for well-typed code to write to the map (whereas legacy clients can simply treat the returned reference as a raw type).

Harder, but by no means impossible. For example, the Get and Put Principle never prevents a client from supplying as an argument the one value that which belongs to every type, namely `null`. Methods other than `add` or `set` can change values in a collection without needing `T`-typed arguments, for example `clear` and `remove`. In the case of `List` you can even permute the elements without knowing their type; `Collections.reverse` does this (see Section 2.7) by means of wildcard capture.

For real—if limited—immutability, turn to the unmodifiable collections provided by the `Collections` class (Sections 14.2.1). Attempts to call any method that might modify these collections causes a runtime failure with `UnsupportedOperationException`. Of course, for unmodifiable collections to be truly immutable, they must either have exclusive access to their components or the components must themselves be immutable.

Chapter 17

Evolution

In this chapter we discuss issues that arise in relation to legacy code. We cover both the situation in which you are converting it to use generics, and the one in which your generic code must interact with legacy code you can't generify.

17.1 Use Checked Collections

The guarantee that generics offer in respect of collections is that a typed collection will never contain objects of an inappropriate type. But the implementation of this guarantee depends on compiler checks that all operations on a parameterised collection are well-typed. In practice, there will be situations in which compiler checking can't be done, most often because generified code has to interact with legacy code using raw types. In these situations, you can use the checked collections provided in `java.util.Collections` as wrappers around your collection objects. Checked collections expect to be used in an environment that they don't trust; since their client might call their `add`, `addAll` or `set` method with inappropriate arguments, they do a run-time check before accepting the values provided.

For example, suppose that we have a legacy library which has methods which can add values of arbitrary type to a supplied collection, and can return collections of arbitrary type. Generic code using this library should use one of the techniques we suggest for evolution: stubs or generified signatures (Chapter 4), or wrapper classes (Section 17.2). However, these techniques all involve significant work; let's suppose that hasn't been done, and consider the consequences. In simplified form the legacy code might look like this:

```
class LegacyLibrary {
  public static void listAdd( List l ) {
    l.add(0);
  }
  public static List listCreate() {
    return new ArrayList(){{ add(0); }}
  }
}
```

Compiling this in Java 5 would produce unchecked warnings that in principle should alert the client programmer to the risks of calling it from generic code. In practice compiling legacy code produces so many unchecked warnings that no-one will ever examine them all carefully. So incautious generic client code using `LegacyLibrary` will get no indication of problems until it tries to use the returned collection.

```
class TrustingModernClient {
  public static void main( String... args ) {
    List<String> ls = new ArrayList<String>();
    LegacyLibrary.listAdd(ls);    // no warning
    // some time later...
    String s = ls.get(0);         // ClassCastException
  }
}
```

By contrast, a wary client which provides a checked collection to the legacy class will fail appropriately (at the line marked "runtime error" below) with a `ClassCastException` containing the message "Attempt to insert class java.lang.Integer element into collection with element type class java.lang.String".

```
class WaryModernClient {
  public static void main( String... args ) {
    List<String> ls = Collections.checkedList(
        new ArrayList<String>(), String.class );
    LegacyLibrary.listAdd(ls);    // runtime error
  }
}
```

A variation on this idiom can be useful when you are writing generic code which, instead of creating a collection to supply to an untrusted environment, must instead accept a collection from one. This might be a collection returned from a legacy library (as with the method `LegacyLibrary.listCreate` above), or it might be one supplied from a client which has—unintentionally or maliciously—inserted an incorrectly typed element into the collection. You can use checked collections here too, by creating a new checked empty collection and adding the elements of the suspect collection to it. From then on you can use the checked collection, or if you don't want to incur the performance overhead of checking all further additions to it, you can transfer its contents back into a well-typed non-checked collection.

```
List<String> checkedList =
      Collections.checkedList( new ArrayList<String>(), String.class);
checkedList.addAll( LegacyLibrary.listCreate() );   // unchecked conversion
// from now on use checkedList or, to avoid further run-time checking:
List<String> ls = new ArrayList<String>( checkedList );
```

If you really have to avoid the overhead of checked collections altogether, then at the cost of some extra code you can use the promotion technique of Section 5.2, in which you check the type of each member and if all checks are successful you then cast the collection of supertype references to a list of references of the subtype.

Checked collections have their limitations. The checks use `Class.isInstance` to determine whether a value is correctly typed with respect to the element class for the collection. These checks take place at run-time, so for checked collections non-reifiable types present a problem: for example, suppose that instead of the collection of `String` objects above, you want to work with a collection of type `ArrayList<String>`. The only class token that you can supply to the `checkedList` method is for the raw type `ArrayList`. The first argument must therefore be cast to match it, and the call becomes:

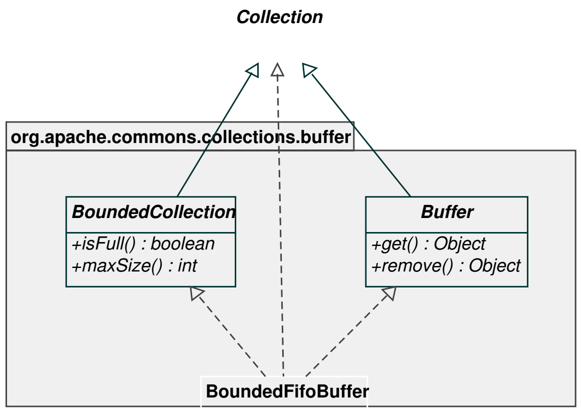


Figure 17.1: Commons Collections class `BoundedFifoBuffer`

```
List<ArrayList<String>> las = Collections.checkedList(
    (List) new ArrayList<ArrayList<String>>(), ArrayList.class );
```

which is unsatisfactory on two counts: first, that type-correctness will be checked only for the top-level component of the list—an `ArrayList` of `Integer` can be added without being detected—and second that supplying a raw type as the first argument to `checkedList` produces an unchecked warning. The reason that the target of the cast must be the raw type is straightforward: the type of the first argument must be a subtype of the LHS, and the obvious candidate as cast target, `List<ArrayList>`, is not such a subtype because although `ArrayList` is a subtype of `ArrayList<String>`, generic types are not covariant.

17.2 Provide Wrappers to Legacy Code

We have now seen many contexts in which interfacing with non-generified legacy code results in unchecked warnings about type insecurities at the interface. In line with the aim of limiting the programmer's exposure to unchecked warnings, it is good practice to confine the warnings resulting from interaction with legacy code to wrapper classes for which they can be safely suppressed. Here is an example, drawn from the popular Apache Commons Collections, an API which builds on the Collections Framework to provide many new features including interfaces for bags, bidirectional maps and map iterators, besides new implementations for many of the standard interfaces. Commons Collections version 3.1, released in 2004, was not generified. In the package `org.apache.commons.collections.buffer` it contains the class `BoundedFifoBuffer`, which implements a first-in-first-out buffer with a fixed maximum size. This class implements two interfaces, `BoundedCollection` and `Buffer`, in the same package (see Figure 17.1).

A generic wrapper for `BoundedFifoBuffer` is shown in Example 17.1. The first thing to notice about this code is that it also has been necessary to supply generified versions of both of the interfaces it implements, even though only `GenericBuffer` (the generified form of `Buffer`) uses its type parameter. But `BoundedCollection` also extends `Collection`, so if it was used as a raw type

BoundedFifoBuffer would inherit raw method declarations from Collection, in conflict with the generic ones inherited from GenericBuffer.

The last eight method definitions in the wrapper class produce unchecked warnings (as expected) for two main reasons: unchecked calls are flagged where methods of the ungenerified class are being supplied with parameterized types, with no compile-time constraint on how they treat them (toArray, containsAll, addAll, removeAll, and retainAll); unchecked casts are flagged when values being returned from these methods are being cast to generic types (or arrays of generic types) again with no compile-time security, in this case that the casts will succeed (toArray, get, and remove).

A third cause of unchecked warnings is fundamentally similar to casting; sometimes casting from a raw collection type is unnecessary because the erased types are the same. This is flagged as an unsafe conversion (*e.g.* iterator above). An unsafe conversion can always be written as an unsafe cast by inserting a suitable cast:

```
public Iterator<E> iterator() {
  return (Iterator<E>)buffer.iterator();        // unchecked cast
}
```

and the argument for doing so is that it clarifies what's happening.

Wrapper classes have advantages and disadvantages compared to the stub classes introduced in Chapter 4. They have a performance overhead, however small. To remove them from the codebase when they become redundant you will have to change the code that uses them. On the other hand, they do not require maintenance of the different compilation classpaths needed when compiling against stubs, and they allow for extra function to be added to library method calls—for example, you may want to insert runtime typechecking until the library is generified.

Example 17.1 Generic wrapper for `BoundedFifoBuffer`

```
interface GenericBuffer<E> extends Collection<E> {
  public E get();
  public E remove();
}

interface GenericBoundedCollection<E>  extends Collection<E> {
  public boolean isFull();
  public int maxSize();
}

@SuppressWarnings("unchecked")
class GenericFifoBuffer<E> implements GenericBoundedCollection<E>,
        GenericBuffer<E>, Collection<E>, Serializable {

  private BoundedFifoBuffer buffer;

  public GenericFifoBuffer(BoundedFifoBuffer boundedFifoBuffer) {
    this.buffer = boundedFifoBuffer;
  }

  // GenericBoundedCollection<E>
  public boolean isFull() { return buffer.isFull(); }
  public int maxSize() { return buffer.maxSize(); }

  // Collection<E>
  public int size() { return buffer.size(); }
  public boolean isEmpty() { return buffer.isEmpty(); }
  public boolean contains(Object o) { return buffer.contains(o); }
  public boolean add(E o) { return buffer.add(o); }
  public boolean remove(Object o) { return buffer.remove(o); }
  public void clear() { buffer.clear(); }
  public Object[] toArray() { return buffer.toArray(); }

  public <T> T[] toArray(T[] a) {
    return (T[])buffer.toArray(a);  // unchecked cast
  }
  public Iterator<E> iterator() {
    return buffer.iterator();       // unchecked conversion
  }
  public boolean containsAll(Collection<?> c) {
    return buffer.containsAll(c);   // unchecked call
  }
  public boolean addAll(Collection<? extends E> c) {
    return buffer.addAll(c);        // unchecked call
  }
  public boolean removeAll(Collection<?> c) {
    return buffer.removeAll(c);     // unchecked call
  }
  public boolean retainAll(Collection<?> c) {
    return buffer.retainAll(c);     // unchecked call
  }
  // GenericBuffer<E>
  public E get() {
    return (E)buffer.get();         // unchecked cast
  }
  public E remove() {
    return (E)buffer.remove();      // unchecked cast
  }
```

17.3 Choose Appropriate Types for Generification

When you are converting legacy code to use generics, you will want to be sure that the benefit you get, or that the clients of your code get, is a just reward for the effort of generification. The reward—greater type safety or expressiveness in the signatures—will accrue only if the class or interface uses a particular unknown type consistently. A good example is the class `java.lang.ThreadLocal`, generified for Java 5. A `ThreadLocal` object simulates an instance variable, with set and get methods, but providing a separate copy for each thread that accesses it. Before Java 5 its declaration was

```
public class ThreadLocal {
  protected Object initialValue() { /*...*/ }
  public Object get() { /*...*/ }
  public void set(Object value) { /*...*/ }
}
```

This class is a container for a single value, so converting it to a generic declaration is like converting a collection classes without the potential complications of heterogenous types. Its Java 5 declaration simply replaces `Object` everywhere in the signature with a more precise type (and also adds an extra method `remove`, to reduce the `ThreadLocal`'s space requirement):

```
public class ThreadLocal<T> {
  protected T initialValue() { /*...*/ }
  public T get() { /*...*/ }
  public void set(T value) { /*...*/ }
  public void remove() { /*...*/ }
}
```

Generifying one type can often lead to generification of other related types, particularly in the case of inheritance. `ThreadLocal` has a subclass `InheritableThreadLocal` to provide inheritance of values from parent thread to child thread. Inheritable thread-local variables are automatically passed from parent to child threads when a child is created. A method `childValue` is provided to allow the parent values to be changed before being passed to the child. Generification of this subclass is again straightforward:

```
public class InheritableThreadLocal<T> extends ThreadLocal<T> {
  protected T childValue( T parentValue ) { /*...*/ }
}
```

Propagation of type parameterization can also occur through aggregation. For example, `java.lang.ref.Reference` is the base class for reference objects. (It is the superclass of `WeakReference` and `SoftReference`, described in Section 13.2.4). A reference object is a container for a single value, with constructors which supply that value and a `get` method to access it. Generifying `Reference` and its subclasses proceeds straightforwardly along the lines described above; for example, one constructor and the `get` method for `WeakReference<T>` are generified as

```
  public WeakReference( T referent ) { /*...*/ }
  public T get() { /*...*/ }
```

but the other constructor needs a little more thought. It takes a referent and a `ReferenceQueue` argument, which the `WeakReference` holds for later use for the garbage collector. After the garbage collecter has reclaimed the referent it adds the `WeakReference` itself on to the specified queue so that the application can detect what has happened and do any necessary post-processing.

Before generification `ReferenceQueue` had the signature

```
public class ReferenceQueue {
  public Reference poll() { /*...*/ }
  public Reference remove(long timeout)
          throws IllegalArgumentException, InterruptedException {
    // ...
  }
  public Reference remove() throws InterruptedException { /*...*/ }
}
```

What should the relationship be between the type of a `ReferenceQueue` and the type of the `References` it contains? One possibility would be to allow queues to contain any kind of references at all, but the designers were guided here by the principle that in a situation like this you should generify as much as possible. If `ReferenceQueue` had not been generified, its methods would have had to return `Reference<?>`, and such returned values would be of limited use; in particular, calling the `get` method on them would give us values that we could use only at type `Object`. The generification actually chosen returns `Reference` values parameterised on an upper-bounded wildcard so that, for example, a queue instantiated on `Number` could contain only references to `Number` subtypes, and `get` called on returned `Reference` values would return `Number` references. The Java 5 declaration of `ReferenceQueue` is:

```
public class ReferenceQueue<T> {
  public Reference<? extends T> poll() { /*...*/ }
  public Reference<? extends T> remove(long timeout)
          throws IllegalArgumentException, InterruptedException {
    // ...
  }
  public Reference<? extends T> remove() throws InterruptedException { /*...*/ }
}
```

Now we only have to return to `WeakReference` (and the other two `Reference` subclasses) and adjust the signature of the second constructor. To fit with the constraint on ReferenceQueue, this constructor should accept a `ReferenceQueue` instantiated on any supertype of the type argument to `WeakReference`:

```
  public WeakReference( T referent, ReferenceQueue<? super T> q ) { /*...*/ }
  public T get() { /*...*/ }
```

17.4 Maintain Binary Compatibility when Generifying Existing Classes

Since compatibility was a principal reason for choosing the design of Java generics, you might hope that generifying existing code would not normally introduce incompatibilities with pre-generic code, either as source or binary. In most cases you would not be disappointed; for example, no problems were caused in the generification of `java.lang.ref` described in the preceding section. Not every situation is so straightforward, though; in this section we will look at some areas that can cause difficulty.

If newly-generified code interoperates with existing binaries to produce exactly the same results as the its pre-generic version, we say that the two versions are *binary compatible*. The strongest guarantee of binary compatibility is provided by ensuring that the compiled code for the two versions is identical; this property will hold if the signature of the pre-generic version is the same as the erasure of the generified one (is a *sub-signature* of it). Usually this is a natural consequence of generification but occasionally, especially with poorly-designed APIs, you have to use some ingenuity.

One well-known problem of this kind is the generification of the `Collections` class. We saw earlier (Section 3.2) that the pre-generic signature of the `max` method

```
public static Object max(Collection coll)
```

needed to be changed to ensure that the collection argument is parameterised on a type that implements a suitably parameterised version of `Comparable`. That gives the generic signature

```
public static <T extends Comparable<? super T>>
            T max(Collection<? extends T> coll)
```

but the erasure of this signature as it stands is

```
public static Comparable max(Collection coll)
```

Clients of the `Collections` class compiled against the pre-generics version will fail with a `NoSuchMethodError` if they call this method. Multiple bounds (Section 3.6) can be used to overcome this problem. Introducing a leftmost bound of `Object` does not alter the type of the return value but changes the erased signature to that of the pre-generic version:

```
public static <T extends Object & Comparable<? super T>>
            T max(Collection<? extends T> coll)
```

A complicating factor in achieving binary compatibility can be introduction of bridge methods (Section 3.7). A common example of this problem is found in inheritance from superclasses or superinterfaces which have acquired a generic type parameter. `Comparable` is a good example; classes and interfaces inheriting from it had to be well designed to avoid problems with generification. Most of the core classes did in fact avoid these problems; for example, `java.io.File` had two overloads of `compareTo` before generification:

```
public class File implements Comparable {
  public int compareTo( Object o ) { /* ... */ }
  public int compareTo( File f ) { /* ... */ }
  // ...
}
```

The contract for the first `compareTo` overload specified that if its argument was a `File` it would delegate to the second one, otherwise throwing a `ClassCastException`. This is exactly the behaviour of the bridge method that replaced it after generification:

```
public class File implements Comparable<File> {
  public int compareTo( Object o ) { /* ... */ }  // now a bridge method
  public int compareTo( File f ) { /* ... */ }
  // ...
}
```

Clients and subclasses which call or override the first overload will now fail a Java 5 compilation unless the `-source 1.4` flag is used to ensure that the bridge method is treated as an ordinary manually-generated method. But because the signature of the pre-generic version is a subsignature of the generified version, the bytecode for the two is the same, and binary compatibility is assured.

Now consider an interface whose pre-generic design was not so forward-looking, `javax.naming.Name`.

```
public interface Name extends Comparable {
  public int compareTo( Object o ) { /* ... */ }
  // ...
}
```

Routine generification of `Name` would have given the signature

```
public interface Name extends Comparable<Name> {
  public int compareTo( Name o ) { /* ... */ }
  // ...
}
```

but then its subclass `CompositeName` would have the signature

```
public class CompositeName implements Name {
  public int compareTo( Object o ) { /* ... */ }  // bridge method
  public int compareTo( Name o ) { /* ... */ }    // newly introduced
  // ...
}
```

Non-generic subclasses of `CompositeName` would only have overriden `compareTo(Object)`, so any client calling `compareTo` on such a subclass supplying an argument of type `Name` will find that generification has altered its behaviour. The only solution is to choose a less ambitious generification,

```
public interface Name extends Comparable<Object> { /* ... */ }
```

whose erasure is the same the signature of the raw type.

To avoid problems of this kind, be very cautious in generifying a class that inherits from a parameterised superclass or interface. Instantiating such a superclass or interface with any concrete type can lead to compatibility problems unless either your class is final or its subclasses are under your control.

One problem for compatibility, not always related to generics but introduced in Java 5, is covariance of method return types (Section 16.5). In Java 5 an overriding method is permitted to change the type of its return to a subtype of that returned by the method being overridden. An obvious application of this is in defining `clone`, which until now has been compelled to return `Object`. For example, in updating the Collections Framework for Java 5, it would have been legal to change the declaration of `HashSet` to

```
class HashSet {
  public HashSet clone() { /*...*/ }
  // ...
}
```

saving the client a cast. In fact neither `HashSet` nor any of the core classes have been changed in this way (just as you should not change any of your nonfinal library classes) because existing subclasses may have overridden `clone` with a return type of `Object`:

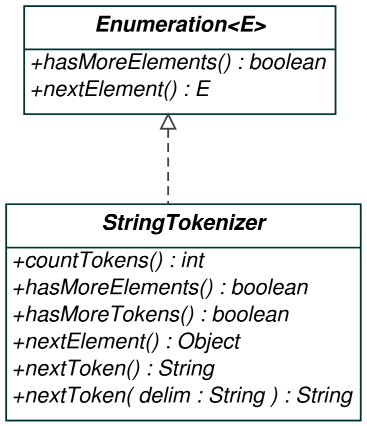

Enumeration<E>
+hasMoreElements() : boolean
+nextElement() : E
StringTokenizer
+countTokens() : int
+hasMoreElements() : boolean
+hasMoreTokens() : boolean
+nextElement() : Object
+nextToken() : String
+nextToken(delim : String) : String

Figure 17.2: Generifying `StringTokenizer`

```
class HashSetSubClass extends HashSet {
  public Object clone() { /*...*/ }
  // ...
}
```

and this will now be incompatible with the changed definition of `HashSet`.

On some occasions this can interfere with generification too: for example the class `java.util.StringTokenizer` implements the legacy interface `java.util.Enumeration` (Figure 17.2). The obvious instantiation to `Enumeration<String>` would give `StringTokenizer.nextElement` a return type of `String`, but that change would break any existing subclasses in which `nextElement`'s return type is `Object`. By contrast, the modern replacement for `StringTokenizer`, the class `java.util.Scanner`, was introduced in Java 5 and so can implement `Iterator<String>` with no compatibility problems.