CHAPTER 7

Collections and Generics

As we start to use our growing knowledge of objects to handle more and more interesting problems, one recurring question will emerge. How do we store the data we're manipulating in the course of solving those problems? We'll definitely use variables of all the different types, but we'll also need bigger, fancier storage options. The arrays we discussed back in "Arrays" on page 114 are a start, but arrays have some limitations. In this chapter we will see how to get efficient, flexible access to large amounts of data. That's where the Java Collections API that we tackle in the next section comes in. We'll also see how to deal with the various types of data we want to store in these big containers like we do with invididual values in variables. That's where generics come in. We'll get to those in "Type Limitations" on page 203.

Collections

Collections are data structures that are fundamental to all types of programming. Whenever we need to refer to a group of objects, we have some kind of collection. At the core language level, Java supports collections in the form of arrays. But arrays are static, and because they have a fixed length, they are awkward for groups of things that grow and shrink over the lifetime of an application. Arrays also do not represent abstract relationships between objects well. In the early days, the Java platform had only two basic classes to address these needs: the `java.util.Vector` class, which represents a dynamic list of objects, and the `java.util.Hashtable` class, which holds a map of key/value pairs. Today, Java has a more comprehensive approach to collections called the Collections Framework. The older classes still exist, but they have been retrofitted into the framework (with some eccentricities) and are generally no longer used.

Though conceptually simple, collections are one of the most powerful parts of any programming language. Collections implement data structures that lie at the heart of managing complex problems. A great deal of basic computer science is devoted to describing the most efficient ways to implement certain types of algorithms over collections. Having these tools at your disposal and understanding how to use them can make your code both much smaller and faster. It can also save you from reinventing the wheel.

The original Collections Framework had two major drawbacks. The first was that collections were by necessity untyped and worked only with undifferentiated `Objects` instead of specific types like `Dates` and `Strings`. This meant that you had to perform a type cast every time you took an object out of a collection. This flew in the face of Java's compile-time type safety. But in practice, this was less a problem than it was just plain cumbersome and tedious. The second issue was that, for practical reasons, collections could work only with objects and not with primitive types. This meant that any time you wanted to put a number or other primitive type into a collection, you had to store it in a wrapper class first and unpack it later upon retrieving it. The combination of these factors made code working with collections less readable and more dangerous to boot.

This all changed with the introduction of generic types and autoboxing of primitive values. First, the introduction of generic types (again, more on this in "Type Limitations" on page 203) has made it possible for truly type-safe collections to be under the control of the programmer. Second, the introduction of autoboxing and unboxing of primitive types means that you can generally treat objects and primitives as equals where collections are concerned. The combination of these new features can significantly reduce the amount of code you write and add safety. As we'll see, all of the collections classes now take advantage of these features.

The Collections Framework is based around a handful of interfaces in the `java.util` package. These interfaces are divided into two hierarchies. The first hierarchy descends from the `Collection` interface. This interface (and its descendants) represents a container that holds other objects. The second, separate hierarchy is based on the `Map` interface, which represents a group of key/value pairs where the key can be used to retrieve the value in an efficient way.

The Collection Interface

The mother of all collections is an interface appropriately named `Collection`. It serves as a container that holds other objects, its *elements*. It doesn't specify exactly how the objects are organized; it doesn't say, for example, whether duplicate objects are allowed or whether the objects are ordered in any way. These kinds of details are left to child interfaces. Nevertheless, the `Collection` interface defines some basic operations common to all collections:

`public boolean add( element )`

Adds the supplied object to this collection. If the operation succeeds, this method returns `true`. If the object already exists in this collection and the collection does not permit duplicates, `false` is returned. Furthermore, some collections are read-only. Those collections throw an `UnsupportedOperationException` if this method is called.

`public boolean remove( element )`

Removes the specified object from this collection. Like the `add()` method, this method returns `true` if the object is removed from the collection. If the object doesn't exist in this collection, `false` is returned. Read-only collections throw an `UnsupportedOperationException` if this method is called.

`public boolean contains( element )`

Returns `true` if the collection contains the specified object.

`public int size()`

Returns the number of elements in this collection.

`public boolean isEmpty()`

Returns `true` if this collection has no elements.

`public Iterator iterator()`

Examines all the elements in this collection. This method returns an `Iterator`, which is an object you can use to step through the collection's elements. We'll talk more about iterators in the next section.

Additionally, the methods `addAll()`, `removeAll()`, and `containsAll()` accept another `Collection` and add, remove, or test for all of the elements of the supplied collection.

Collection Types

The `Collection` interface has three child interfaces. `Set` represents a collection in which duplicate elements are not allowed. `List` is a collection whose elements have a specific order. The `Queue` interface is a buffer for objects with a notion of a "head" element that's next in line for processing.

Set

`Set` has no methods besides the ones it inherits from `Collection`. It simply enforces its no-duplicates rule. If you try to add an element that already exists in a `Set`, the `add()` method simply returns `false`. `SortedSet` maintains elements in a prescribed order; like a sorted list that can contain no duplicates. You can retrieve subsets (which are also sorted) using the `subSet()`, `headSet()`, and `tailSet()` methods. These

methods accept one or a pair of elements that mark the boundaries. The `first()`, `last()`, and `comparator()` methods provide access to the first element, the last element, and the object used to compare elements (more on this in "A Closer Look: The sort() Method" on page 218).

Java 7 added `NavigableSet`, which extends `SortedSet` and adds methods for finding the closest match greater or lesser than a target value within the sort order of the `Set`. This interface can be implemented efficiently using techniques such as skip lists, which make finding ordered elements fast.

List

The next child interface of `Collection` is `List`. The `List` is an ordered collection, similar to an array but with methods for manipulating the position of elements in the list:

`public boolean add( E element )`
: Adds the specified element to the end of the list.

`public void add( int index , E element )`
: Inserts the given object at the supplied position in the list. If the position is less than zero or greater than the list length, an `IndexOutOfBoundsException` will be thrown. The element that was previously at the supplied position, and all elements after it, are moved up one index position.

`public void remove( int index )`
: Removes the element at the specified position. All subsequent elements move down one index position.

`public E get( int index )`
: Returns the element at the given position.

`public Object set( int index , E element )`
: Changes the element at the given position to the specified object. There must already be an object at the index or else an `IndexOutOfBoundsException` is thrown.

The type `E` in these methods refers to the parameterized element type of the `List` class. `Collection`, `Set`, and `List` are all interface types. This is an example of the Generics feature we hinted at in the introduction to this chapter, and we'll look at concrete implementations of these shortly.

Queue

A `Queue` is a collection that acts like a buffer for elements. The queue maintains the insertion order of items placed into it and has the notion of a "head" item. Queues may be first in, first out (FIFO) or last in, first out (LIFO), depending on the implementation:

`public boolean offer( E element )`, `public boolean add( E element )`

The `offer()` method attempts to place the element into the queue, returning `true` if successful. Different `Queue` types may have different limits or restrictions on element types (including capacity). This method differs from the `add()` method inherited from `Collection` in that it returns a Boolean value instead of throwing an exception to indicate that the element cannot be accepted.

`public E poll()`, `public E remove()`

The `poll()` method removes the element at the head of the queue and returns it. This method differs from the `Collection` method `remove()` in that if the queue is empty, `null` is returned instead of throwing an exception.

`public E peek()`

Returns the head element *without* removing it from the queue. If the queue is empty, `null` is returned.

The Map Interface

The Collections Framework also includes the `java.util.Map`, which is a collection of key/value pairs. Other names for map are "dictionary" or "associative array." Maps store and retrieve elements with key values; they are very useful for things like caches or minimalist databases. When you store a value in a map, you associate a key object with a value. When you need to look up the value, the map retrieves it using the key.

With generics, a `Map` type is parameterized with two types: one for the keys and one for the values. The following snippet uses a `HashMap`, which is an efficient but unordered type of map implementation that we'll discuss later:

```
Map<String, Date> dateMap = new HashMap<String, Date>();
dateMap.put( "today", new Date() );
Date today = dateMap.get( "today" );
```

In legacy code, maps simply map `Object` types to `Object` types and require the appropriate cast to retrieve values.

The basic operations on `Map` are straightforward. In the following methods, the type `K` refers to the key parameter type, and the type `V` refers to the value parameter type:

`public V put( K key , V value )`

Adds the specified key/value pair to the map. If the map already contains a value for the specified key, the old value is replaced and returned as the result.

`public V get( K key )`

Retrieves the value corresponding to key from the map.

`public V remove( K key )`

Removes the value corresponding to key from the map. The value removed is returned.

`public int size()`

Returns the number of key/value pairs in this map.

You can retrieve all the keys or values in the map using the following methods:

`public Set keySet()`

This method returns a `Set` that contains all the keys in this map.

`public Collection values()`

Use this method to retrieve all the values in this map. The returned `Collection` can contain duplicate elements.

`public Set entrySet()`

This method returns a `Set` that contains all the key/value pairs (as `Map.Entry` objects) in this map.

`Map` has one child interface, `SortedMap`. A `SortedMap` maintains its key/value pairs sorted in a particular order according to the key values. It provides the `subMap()`, `headMap()`, and `tailMap()` methods for retrieving sorted map subsets. Like `SortedSet`, it also provides a `comparator()` method, which returns an object that determines how the map keys are sorted. We'll talk more about that in "A Closer Look: The sort() Method" on page 218. Java 7 added a `NavigableMap` with functionality parallel to that of `NavigableSet`; namely, it adds methods to search the sorted elements for an element greater or lesser than a target value.

Finally, we should make it clear that although related, `Map` is not literally a type of `Collection` (`Map` does not extend the `Collection` interface). You might wonder why. All of the methods of the `Collection` interface would appear to make sense for `Map`, except for `iterator()`. A `Map`, again, has two sets of objects: keys and values, and separate iterators for each. This is why a `Map` does not implement a `Collection`. If you do want a `Collection`-like view of a `Map` with both keys and values, you can use the `entrySet()` method.

One more note about maps: some map implementations (including Java's standard `HashMap`) allow `null` to be used as a key or value, but others may not.

Type Limitations

Generics are about abstraction. Generics let you create classes and methods that work in the same way on different types of objects. The term *generic* comes from the idea that we'd like to be able to write general algorithms that can be broadly reused for many types of objects rather than having to adapt our code to fit each circumstance. This concept is not new; it is the impetus behind object-oriented programming itself. Java generics do not so much add new capabilities to the language as they make reusable Java code easier to write and easier to read.

Generics take reuse to the next level by making the *type* of the objects with which we work an explicit parameter of the generic code. For this reason, generics are also referred to as *parameterized types*. In the case of a generic class, the developer specifies a type as a parameter (an argument) whenever they use the generic type. The class is parameterized by the supplied type to which the code adapts itself.

In other languages, generics are sometimes referred to as *templates*, which is more of an implementation term. Templates are like intermediate classes, waiting for their type parameters so that they can be used. Java takes a different path, which has both benefits and drawbacks that we'll describe in detail in this chapter.

There is much to say about Java generics. Some of the fine points may seem a bit obscure at first, but don't get discouraged. The vast majority of what you'll do with generics—using existing classes such as `List` and `Set`, for example—is easy and intuitive. Designing and creating your own generics requires a more careful understanding and will come with a little patience and tinkering.

Indeed, we begin our discussion in that intuitive space with the most compelling case for generics: the container classes and collections we just covered. Next, we take a step back and look at the good, bad, and ugly of how Java generics work. We conclude by looking at a couple of real-world generic classes in the Java API.

Containers: Building a Better Mousetrap

In an object-oriented programming language like Java, polymorphism means that objects are always to some degree interchangeable. Any child of a type of object can serve in place of its parent type and, ultimately, every object is a child of `java.lang.Object`, the object-oriented "Eve," so to speak. It is natural, therefore, for the most general types of *containers* in Java to work with the type `Object` so that they can hold just about anything. By containers, we mean classes that hold instances of other classes in some way. The Java Collections API we looked at in the previous section is the best example of containers. `List`, to recap, holds an ordered collection of elements of type `Object`. And `Map` holds an association of key/value pairs, with the keys and values also being of the most general type, `Object`. With a little help from wrappers for primitive types, this arrangement has served us well. But (not to get too

Zen on you) in a sense, a "collection of any type" is also a "collection of no type," and working with `Objects` pushes a great deal of responsibility onto the user of the container.

It's kind of like a costume party for objects where everybody is wearing the same mask and disappears into the crowd of the collection. Once objects are dressed as the `Object` type, the compiler can no longer see the real types and loses track of them. It's up to the user to pierce the anonymity of the objects later by using a type cast. And like attempting to yank off a partygoer's fake beard, you'd better have the cast correct or you'll get an unwelcome surprise.

```
Date date = new Date();
List list = new ArrayList();
list.add( date );
...
Date firstElement = (Date)list.get(0); // Is the cast correct?  Maybe.
```

The `List` interface has an `add()` method that accepts any type of `Object`. Here, we assigned an instance of `ArrayList`, which is simply an implementation of the `List` interface, and added a `Date` object. Is the cast in this example correct? It depends on what happens in the elided "..." period of time. Indeed, the Java compiler knows this type of activity is fraught and currently issues warnings when you add elements to a simple `ArrayList` as above. We can see this with a little *jshell* detour. After importing from the `java.util` and `javax.swing` packages, try creating an `ArrayList` and add a few disparate elements:

```
jshell> import java.util.ArrayList;

jshell> import javax.swing.JLabel;

jshell> ArrayList things = new ArrayList();
things ==> []

jshell> things.add("Hi there");
|  Warning:
|  unchecked call to add(E) as a member of the raw type java.util.ArrayList
|  things.add("Hi there");
|  ^--------------------^
$3 ==> true

jshell> things.add(new JLabel("Hi there"));
|  Warning:
|  unchecked call to add(E) as a member of the raw type java.util.ArrayList
|  things.add(new JLabel("Hi there"));
|  ^---------------------------------^
$5 ==> true

jshell> things
things ==> [Hi there, javax.swing.JLabel[...,text=Hi there,...]]
```

You can see the warning is the same no matter what type of object we `add()`. In the last step where we display the contents of `things`, both the plain `String` object and the `JLabel` object are happily in the list. The compiler is not worried about disparate types being used; it is helpfully warning you that it will not know whether casts such as the `(Date)` cast above will work at runtime.

Can Containers Be Fixed?

It's natural to ask if there is a way to make this situation better. What if we know that we are only going to put `Dates` into our list? Can't we just make our own list that only accepts `Date` objects, get rid of the cast, and let the compiler help us again? The answer, surprisingly perhaps, is no. At least, not in a very satisfying way.

Our first instinct may be to try to "override" the methods of `ArrayList` in a subclass. But of course, rewriting the `add()` method in a subclass would not actually override anything; it would add a new *overloaded* method:

```
public void add( Object o ) { ... } // still here
public void add( Date d ) { ... }   // overloaded method
```

The resulting object still accepts any kind of object—it just invokes different methods to get there.

Moving along, we might take on a bigger task. For example, we might write our own ~~`Date`~~`List` class that does not extend `ArrayList`, but rather delegates the guts of its methods to the `ArrayList` implementation. With a fair amount of tedious work, that would get us an object that does everything a `List` does but that works with `Dates` in a way that both the compiler and the runtime environment can understand and enforce. However, we've now shot ourselves in the foot because our container is no longer an implementation of `List` and we can't use it interoperably with all of the utilities that deal with collections, such as `Collections.sort()`, or add it to another collection with the `Collection addAll()` method.

To generalize, the problem is that instead of refining the behavior of our objects, what we really want to do is to change their contract with the user. We want to adapt their API to a more specific type and polymorphism doesn't allow that. It would seem that we are stuck with `Objects` for our collections. And this is where generics come in.

Enter Generics

As we noted when introducing the type limitations in the previous section, generics are an enhancement to the syntax of classes that allow us to specialize the class for a given type or set of types. A generic class requires one or more *type parameters* wherever we refer to the class type and uses them to customize itself.

If you look at the source or Javadoc for the `List` class, for example, you'll see it defines something like this:

```
public class List< E > {
    ...
    public void add( E element ) { ... }
    public E get( int i ) { ... }
}
```

The identifier `E` between the angle brackets (`<>`) is a *type parameter*.[1] It indicates that the class `List` is generic and requires a Java type as an argument to make it complete. The name `E` is arbitrary, but there are conventions that we'll see as we go on. In this case, the type variable `E` represents the type of elements we want to store in the list. The `List` class refers to the type variable within its body and methods as if it were a real type, to be substituted later. The type variable may be used to declare instance variables, arguments to methods, and the return type of methods. In this case, `E` is used as the type for the elements we'll be adding via the `add()` method and the return type of the `get()` method. Let's see how to use it.

The same angle bracket syntax supplies the type parameter when we want to use the `List` type:

```
List<String> listOfStrings;
```

In this snippet, we declared a variable called `listOfStrings` using the generic type `List` with a type parameter of `String`. `String` refers to the `String` class, but we could have a specialized `List` with any Java class type. For example:

```
List<Date> dates;
List<java.math.BigDecimal> decimals;
List<Foo> foos;
```

Completing the type by supplying its type parameter is called *instantiating the type*. It is also sometimes called *invoking the type*, by analogy with invoking a method and supplying its arguments. Whereas with a regular Java type, we simply refer to the type by name, a generic type must be instantiated with parameters wherever it is used.[2] Specifically, this means that we must instantiate the type everywhere types can appear as the declared type of a variable (as shown in this code snippet), as the type of a method argument, as the return type of a method, or in an object allocation expression using the new keyword.

---

1 You may also see the term *type variable* used. The Java Language Specification mostly uses "parameter" so that's what we try to stick with, but you may see both names used in the wild.

2 That is, unless you want to use a generic type in a nongeneric way. We'll talk about "raw" types later in this chapter.

Returning to our `listOfStrings`, what we have now is effectively a `List` in which the type `String` has been substituted for the type variable `E` in the class body:

```
public class List< String > {
   ...
   public void add( String element ) { ... }
   public String get( int i ) { ... }
}
```

We have specialized the `List` class to work with elements of type `String` and *only* elements of type `String`. This method signature is no longer capable of accepting an arbitrary `Object` type.

`List` is just an interface. To use the variable, we'll need to create an instance of some actual implementation of `List`. As we did in our introduction, we'll use `ArrayList`. As before, `ArrayList` is a class that implements the `List` interface, but in this case, both `List` and `ArrayList` are generic classes. As such, they require type parameters to instantiate them where they are used. Of course, we'll create our `ArrayList` to hold `String` elements to match our `List` of `Strings`:

```
List<String> listOfStrings = new ArrayList<String>
// Or shorthand in Java 7.0 and later
List<String> listOfStrings = new ArrayList<>();
```

As always, the `new` keyword takes a Java type and parentheses with possible arguments for the class's constructor. In this case, the type is `ArrayList<String>`—the generic `ArrayList` type instantiated with the `String` type.

Declaring variables as shown in the first line of the preceding example is a bit cumbersome because it requires us to type the generic parameter type twice (once on the left side in the variable type and once on the right in the initialing expression). And in complicated cases, the generic types can get very lengthy and nested within one another. Starting with Java 7, the compiler is smart enough to infer the type of the initializing expression from the type of the variable to which you are assigning it. This is called *generic type inference* and boils down to the fact that you can use shorthand on the right side of your variable declarations by leaving out the contents of the `<>` notation, as shown in the example's second version.

We can now use our specialized `List` with strings. The compiler prevents us from even trying to put anything other than a `String` object (or a subtype of `String` if there were any) into the list and allows us to fetch them with the `get()` method without requiring any cast:

```
jshell> ArrayList<String> listOfStrings = new ArrayList<>();
listOfStrings ==> []

jshell> listOfStrings.add("Hey!");
$8 ==> true
```

```
jshell> listOfStrings.add(new JLabel("Hey there"));
|  Error:
|  incompatible types: javax.swing.JLabel cannot be converted to java.lang.String
|  listOfStrings.add(new JLabel("Hey there"));
|                    ^--------------------^

jshell> String s = strings.get(0);
s ==> "Hey!"
```

Let's take another example from the Collections API. The `Map` interface provides a dictionary-like mapping that associates key objects with value objects. Keys and values do not have to be of the same type. The generic `Map` interface requires two type parameters: one for the key type and one for the value type. The Javadoc looks like this:

```
public class Map< K, V > {
    ...
    public V put( K key, V value ) { ... } // returns any old value
    public V get( K key ) { ... }
}
```

We can make a `Map` that stores `Employee` objects by `Integer` "employee ID" numbers like this:

```
Map< Integer, Employee > employees = new HashMap< Integer, Employee >();
Integer bobsId = 314; // hooray for autoboxing!
Employee bob = new Employee("Bob", ... );

employees.put( bobsId, bob );
Employee employee = employees.get( bobsId );
```

Here, we used `HashMap`, which is a generic class that implements the `Map` interface, and instantiated both types with the type parameters `Integer` and `Employee`. The `Map` now works only with keys of type `Integer` and holds values of type `Employee`.

The reason we used `Integer` here to hold our number is that the type parameters to a generic class must be class types. We can't parameterize a generic class with a primitive type, such as `int` or `boolean`. Fortunately, autoboxing of primitives in Java (see "Wrappers for Primitive Types" on page 141) makes it almost appear as if we can by allowing us to use primitive types as though they were wrapper types.

Dozens of other APIs beyond collections use generics to let you adapt them to specific types. We'll talk about them as they occur throughout the book.

Talking About Types

Before we move on to more important things, we should say a few words about the way we describe a particular parameterization of a generic class. Because the most common and compelling case for generics is for container-like objects, it's common

to think in terms of a generic type "holding" a parameter type. In our example, we called our `List<String>` a "list of strings" because, sure enough, that's what it was. Similarly, we might have called our employee map a "Map of employee IDs to Employee objects." However, these descriptions focus a little more on what the classes *do* than on the type itself. Take instead a single object container called `Trap< E >` that could be instantiated on an object of type `Mouse` or of type `Bear`; that is, `Trap<Mouse>` or `Trap<Bear>`. Our instinct is to call the new type a "mouse trap" or "bear trap." Similarly, we could have thought of our list of strings as a new type: "string list," or our employee map as a new "integer employee object map" type. You may use whatever verbiage you prefer, but these latter descriptions focus more on the notion of the generic as a *type* and may help you keep the terms straight when we discuss how generic types are related in the type system. There we'll see that the container terminology turns out to be a little counterintuitive.

In the following section, we'll continue our discussion of generic types in Java from a different perspective. We've seen a little of what they can do; now we need to talk about how they do it.

"There Is No Spoon"

In the movie *The Matrix*,[3] the hero Neo is offered a choice. Take the blue pill and remain in the world of fantasy, or take the red pill and see things as they really are. In dealing with generics in Java, we are faced with a similar ontological dilemma. We can go only so far in any discussion of generics before we are forced to confront the reality of how they are implemented. Our fantasy world is one created by the compiler to make our lives writing code easier to accept. Our reality (though not quite the dystopian nightmare in the movie) is a harsher place, filled with unseen dangers and questions. Why don't casts and tests work properly with generics? Why can't I implement what appear to be two different generic interfaces in one class? Why is it that I can declare an array of generic types, even though there is no way in Java to create such an array?!? We'll answer these questions and more in this chapter, and you won't even have to wait for the sequel. You'll be bending spoons (well, types) in no time. Let's get started.

The design goals for Java generics were formidable: add a radical new syntax to the language that safely introduces parameterized types with no impact on performance and, oh, by the way, make it backward compatible with all existing Java code and don't change the compiled classes in any serious way. It's actually quite amazing that

3 For those of you who might like some context for the title of this section, here is where it comes from: Boy: Do not try and bend the spoon. That's impossible. Instead, only try to realize the truth. Neo: What truth? Boy: There is no spoon. Neo: There is no spoon? Boy: Then you'll see that it is not the spoon that bends, it is only yourself. —The Wachowskis. *The Matrix*. 136 minutes. Warner Brothers, 1999.

these conditions could be satisfied at all and no surprise that it took a while. But as always, compromises were required, which led to some headaches.

Erasure

To accomplish this feat, Java employs a technique called *erasure*, which relates to the idea that since most everything we do with generics applies statically at compile time, generic information does not need to be carried over into the compiled classes. The generic nature of the classes, enforced by the compiler, can be "erased" in the compiled classes, which allows us to maintain compatibility with nongeneric code. While Java does retain information about the generic features of classes in the compiled form, this information is used mainly by the compiler. The Java runtime does not know anything about generics at all.

Let's take a look at a compiled generic class: our friend, `List`. We can do this easily with the *javap* command:

```
% javap java.util.List

public interface java.util.List extends java.util.Collection{
    ...
    public abstract boolean add(java.lang.Object);
    public abstract java.lang.Object get(int);
```

The result looks exactly like it did prior to Java generics, as you can confirm with any older version of the JDK. Notably, the type of elements used with the `add()` and `get()` methods is `Object`. Now, you might think that this is just a ruse and that when the actual type is instantiated, Java will create a new version of the class internally. But that's not the case. This is the one and only `List` class, and it is the actual runtime type used by all parameterizations of `List`; for example, `List<Date>` and `List<String>`, as we can confirm:

```
List<Date> dateList  = new ArrayList<Date>();
System.out.println( dateList instanceof List ); // true!
```

But our generic `dateList` clearly does not implement the `List` methods just discussed:

```
dateList.add( new Object() ); // Compile-time Error!
```

This illustrates the somewhat schizophrenic nature of Java generics. The compiler believes in them, but the runtime says they are an illusion. What if we try something a little more sane and simply check that our `dateList` is a `List<Date>`:

```
System.out.println( dateList instanceof List<Date> ); // Compile-time Error!
// Illegal, generic type for instanceof
```

This time the compiler simply puts its foot down and says, "No." You can't test for a generic type in an `instanceof` operation. Since there are no actual differentiable

classes for different parameterizations of `List` at runtime, there is no way for the `instanceof` operator to tell the difference between one incarnation of `List` and another. All of the generic safety checking was done at compile time and now we're just dealing with a single actual `List` type.

What has really happened is that the compiler has erased all of the angle bracket syntax and replaced the type variables in our `List` class with a type that can work at runtime with any allowed type: in this case, `Object`. We would seem to be back where we started, except that the compiler still has the knowledge to enforce our usage of the generics in the code at compile time and can, therefore, handle the cast for us. If you decompile a class using a `List<Date>` (the *javap* command with the *-c* option shows you the bytecode, if you dare), you will see that the compiled code actually contains the cast to `Date`, even though we didn't write it ourselves.

We can now answer one of the questions we posed at the beginning of the section: "Why can't I implement what appear to be two different generic interfaces in one class?" We can't have a class that implements two different generic `List` instantiations because they are really the same type at runtime and there is no way to tell them apart:

```
public abstract class DualList implements List<String>, List<Date> { }
// Error: java.util.List cannot be inherited with different arguments:
//    <java.lang.String> and <java.util.Date>
```

Fortunately, there are always workarounds. In this case, for example, you can use a common superclass or create multiple classes. The alternatives may not be as elegant as you'd like, but you can almost always land on a clean answer even if it is a little verbose.

Raw Types

Although the compiler treats different parameterizations of a generic type as different types (with different APIs) at compile time, we have seen that only one real type exists at runtime. For example, the class of `List<Date>` and `List<String>` shares the plain old Java class `List`. `List` is called the *raw type* of the generic class. Every generic has a raw type. It is the degenerate, "plain" Java form from which all of the generic type information has been removed and the type variables replaced by a general Java type like `Object`.[4]

4 When generics were added in Java 5.0, things were carefully arranged such that the raw type of all of the generic classes worked out to be exactly the same as the earlier, nongeneric types. So the raw type of a `List` in Java 5.0 is the same as the old, nongeneric `List` type that had been around since JDK 1.2. Since the vast majority of current Java code at the time did not use generics, this type equivalency and compatibility was very important.

It is still possible to use raw types in Java just as before generics were added to the language. The only difference is that the Java compiler generates a warning wherever they are used in an "unsafe" way. Outside *jshell*, the compiler still notices these problems:

```
// nongeneric Java code using the raw type
List list = new ArrayList(); // assignment ok
list.add("foo"); // Compiler warning on usage of raw type
```

This snippet uses the raw `List` type just as old-fashioned Java code prior to Java 5 would have. The difference is that now the Java compiler issues an *unchecked warning* about the code if we attempt to insert an object into the list:

```
% javac MyClass.java
Note: MyClass.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

The compiler instructs us to use the `-Xlint:unchecked` option to get more specific information about the locations of unsafe operations:

```
% javac -Xlint:unchecked MyClass.java
warning: [unchecked] unchecked call to add(E) as a member of the raw type
         java.util.
List:   list.add("foo");
```

Note that creating and assigning the raw `ArrayList` does not generate a warning. It is only when we try to use an "unsafe" method (one that refers to a type variable) that we get the warning. This means that it's still OK to use older-style, nongeneric Java APIs that work with raw types. We only get warnings when we do something unsafe in our own code.

One more thing about erasure before we move on. In the previous examples, the type variables were replaced by the `Object` type, which could represent any type applicable to the type variable `E`. Later, we'll see that this is not always the case. We can place limitations or *bounds* on the parameter types, and, when we do, the compiler can be more restrictive about the erasure of the type, for example:

```
class Bounded< E extends Date > {
    public void addElement( E element ) { ... }
}
```

This parameter type declaration says that the element type `E` must be a subtype of the `Date` type. In this case, the erasure of the `addElement()` method is therefore more restrictive than `Object`, and the compiler uses `Date`:

```
public void addElement( Date element ) { ... }
```

`Date` is called the *upper bound* of this type, meaning that it is the top of the object hierarchy here and the type can be instantiated only on type `Date` or on "lower" (more derived) types.

Now that we have a handle on what generic types really are, we can go into a little more detail about how they behave.

Parameterized Type Relationships

We know now that parameterized types share a common, raw type. This is why our parameterized `List<Date>` is just a `List` at runtime. In fact, we can assign any instantiation of `List` to the raw type if we want:

```
List list = new ArrayList<Date>();
```

We can even go the other way and assign a raw type to a specific instantiation of the generic type:

```
List<Date> dates = new ArrayList(); // unchecked warning
```

This statement generates an unchecked warning on the assignment, but thereafter the compiler trusts that the list contained only `Dates` prior to the assignment. It is also permissible, albeit pointless, to perform a cast in this statement. We'll talk about casting to generic types shortly in "Casts" on page 215.

Whatever the runtime types, the compiler is running the show and does not let us assign things that are clearly incompatible:

```
List<Date> dates = new ArrayList<String>(); // Compile-time Error!
```

Of course, the `ArrayList<String>` does not implement the methods of `List<Date>` conjured by the compiler, so these types are incompatible.

But what about more interesting type relationships? The `List` interface, for example, is a subtype of the more general `Collection` interface. Is a particular instantiation of the generic `List` also assignable to some instantiation of the generic `Collection`? Does it depend on the type parameters and their relationships? Clearly, a `List<Date>` is not a `Collection<String>`. But is a `List<Date>` a `Collection<Date>`? Can a `List<Date>` be a `Collection<Object>`?

We'll just blurt out the answer here first, then walk through it and explain. The rule is that for the simple types of generic instantiations we've discussed so far, *inheritance applies only to the "base" generic type and not to the parameter types*. Furthermore, assignability applies only when the two generic types are instantiated on *exactly the same parameter type*. In other words, there is still one-dimensional inheritance, following the base generic class type, but with the additional restriction that the parameter types must be identical.

For example, recalling that a `List` is a type of `Collection`, we can assign instantiations of `List` to instantiations of `Collection` when the type parameter is exactly the same:

```
Collection<Date> cd;
List<Date> ld = new ArrayList<Date>();
cd = ld; // Ok!
```

This code snippet says that a `List<Date>` is a `Collection<Date>`—pretty intuitive. But trying the same logic on a variation in the parameter types fails:

```
List<Object> lo;
List<Date> ld = new ArrayList<Date>();
lo = ld; // Compile-time Error!  Incompatible types.
```

Although our intuition tells us that the `Dates` in that `List` could all live happily as `Objects` in a `List`, the assignment is an error. We'll explain precisely why in the next section, but for now just note that the type parameters are not exactly the same and that there is no inheritance relationship among parameter types in generics. This is a case where thinking of the instantiation in terms of types and not in terms of what they do helps. These are not really a "list of dates" and a "list of objects," but more like a `DateList` and an `ObjectList`, the relationship of which is not immediately obvious.

Try to pick out what's OK and what's not OK in the following example:

```
Collection<Number> cn;
List<Integer> li = new ArrayList<Integer>();
cn = li; // Compile-time Error!  Incompatible types.
```

It is possible for an instantiation of `List` to be an instantiation of `Collection`, but only if the parameter types are exactly the same. Inheritance doesn't follow the parameter types and this example fails.

One more thing: earlier we mentioned that this rule applies to the simple types of instantiations we've discussed so far in this chapter. What other types are there? Well, the kinds of instantiations we've seen so far where we plug in an actual Java type as a parameter are called *concrete type instantiations*. Later, we'll talk about *wildcard instantiations*, which are akin to mathematical set operations on types. We'll see that it's possible to make more exotic instantiations of generics where the type relationships are actually two-dimensional, depending both on the base type and the parameterization. But don't worry: this doesn't come up very often and is not as scary as it sounds.

Why Isn't a List<Date> a List<Object>?

It's a reasonable question. Even with our brains thinking of arbitrary `DateList` and `ObjectList` types, we can still ask why they couldn't be assignable. Why shouldn't we be able to assign our `List<Date>` to a `List<Object>` and work with the `Date` elements as `Object` types?

The reason gets back to the heart of the rationale for generics that we discussed in the introduction: changing APIs. In the simplest case, supposing an `ObjectList` type

extends a `DateList` type, the `DateList` would have all of the methods of `ObjectList` and we could still insert `Objects` into it. Now, you might object that generics let us change the APIs, so that doesn't apply anymore. That's true, but there is a bigger problem. If we could assign our `DateList` to an `ObjectList` variable, we would have to be able to use `Object` methods to insert elements of types other than `Date` into it. We could *alias* (provide an alternate, broader type) the `DateList` as an `ObjectList` and try to trick it into accepting some other type:

```
DateList dateList = new DateList();
ObjectList objectList = dateList; // Can't really do this
objectList.add( new Foo() ); // should be runtime error!
```

We'd expect to get a runtime error when the actual `DateList` implementation was presented with the wrong type of object. And therein lies the problem. Java generics have no runtime representation. Even if this functionality were useful, there is no way with the current scheme for Java to know what to do at runtime. Another way to look at it is that this feature is simply dangerous because it allows for an error at runtime that couldn't be caught at compile time. In general, we'd like to catch type errors at compile time.

You might think Java could guarantee that your code is type safe if it compiles with no unchecked warnings by disallowing these assignments. Unfortunately it can't, but it doesn't have to do with generics; it has to do with arrays. If this all sounds familiar to you, it's because we mentioned it previously in relation to Java arrays. Array types have an inheritance relationship that allows this kind of aliasing to occur:

```
Date [] dates = new Date[10];
Object [] objects = dates;
objects[0] = "not a date"; // Runtime ArrayStoreException!
```

However, arrays have runtime representations as different classes and they check themselves at runtime, throwing an `ArrayStoreException` in just this case. So in theory, Java code is not guaranteed type safe by the compiler if you use arrays in this way.

Casts

We've now talked about relationships between generic types and even between generic types and raw types. But we haven't really explored the concept of casts in the world of generics yet. No cast was necessary when we interchanged generics with their raw types. Instead, we just crossed a line that triggers unchecked warnings from the compiler:

```
List list = new ArrayList<Date>();
List<Date> dl = list;  // unchecked warning
```

Normally, we use a cast in Java to work with two types that could be assignable. For example, we could attempt to cast an `Object` to a `Date` because it is plausible that the `Object` is a `Date` value. The cast then performs the check at runtime to see if we are correct. Casting between unrelated types is a compile-time error. For example, we can't even try to cast an `Integer` to a `String`. Those types have no inheritance relationship. What about casts between compatible generic types?

```
Collection<Date> cd = new ArrayList<Date>();
List<Date> ld = (List<Date>)cd; // Ok!
```

This code snippet shows a valid cast from a more general `Collection<Date>` to a `List<Date>`. The cast is plausible here because a `Collection<Date>` is assignable from and could actually be a `List<Date>`. Similarly, the following cast catches our mistake where we have aliased a `TreeSet<Date>` as a `Collection<Date>` and tried to cast it to a `List<Date>`:

```
Collection<Date> cd = new TreeSet<Date>();
List<Date> ld = (List<Date>)cd; // Runtime ClassCastException!
ld.add( new Date() );
```

There is one case where casts are not effective with generics, however, and that is when we are trying to differentiate the types based on their parameter types:

```
Object o = new ArrayList<String>();
List<Date> ld = (List<Date>)o; // unchecked warning, ineffective
Date d = ld.get(0); // unsafe at runtime, implicit cast may fail
```

Here, we aliased an `ArrayList<String>` as a plain `Object`. Next, we cast it to a `List<Date>`. Unfortunately, Java does not know the difference between a `List<String>` and a `List<Date>` at runtime, so the cast is fruitless. The compiler warns us of this by generating an unchecked warning at the location of the cast; we should be aware that when we try to use the cast object later, we might find out that it is incorrect. Casts on generic types are ineffective at runtime because of erasure and the lack of type information.

Converting Between Collections and Arrays

Converting between collections and arrays is easy. For convenience, the elements of a collection can be retrieved as an array using the following methods:

```
public Object[] toArray()
public <E> E[] toArray( E[] a )
```

The first method returns a plain `Object` array. With the second form, we can be more specific and get back an array of the correct element type. If we supply an array of sufficient size, it will be filled in with the values. But if the array is too short (e.g., zero length), a new array of the *same type but the required length* will be created and returned to us. So you can just pass in an empty array of the correct type like this:

```
Collection<String> myCollection = ...;
String [] myStrings = myCollection.toArray( new String[0] );
```

(This trick is a little awkward and it would be nice if Java let us specify the type explicitly using a `Class` reference, but for some reason, this isn't the case.) Going the other way, you can convert an array of objects to a `List` collection with the static `asList()` method of the `java.util.Arrays` class:

```
String [] myStrings = ...;    List list = Arrays.asList( myStrings );
```

Iterator

An *iterator* is an object that lets you step through a sequence of values. This kind of operation comes up so often that it is given a standard interface: `java.util.Iterator`. The `Iterator` interface has only two primary methods:

`public E next()`
: This method returns the next element (an element of generic type E) of the associated collection.

`public boolean hasNext()`
: This method returns `true` if you have not yet stepped through all the `Collection`'s elements. In other words, it returns `true` if you can call `next()` to get the next element.

The following example shows how you could use an `Iterator` to print out every element of a collection:

```
public void printElements(Collection c, PrintStream out) {
    Iterator iterator = c.iterator();
    while ( iterator.hasNext() ) {
        out.println( iterator.next() );
    }
}
```

In addition to the traversal methods, `Iterator` provides the ability to remove an element from a collection:

`public void remove()`
: This method removes the most recent object returned from `next()` from the associated `Collection`.

Not all iterators implement `remove()`. It doesn't make sense to be able to remove an element from a read-only collection, for example. If element removal is not allowed, an `UnsupportedOperationException` is thrown from this method. If you call `remove()` before first calling `next()`, or if you call `remove()` twice in a row, you'll get an `IllegalStateException`.

for loop over collections

A form of the `for` loop, described in "The for loop" on page 105, can operate over all `Iterable` types, which means it can iterate over all types of `Collection` objects as that interface extends `Iterable`. For example, we can now step over all of the elements of a typed collection of `Date` objects like so:

```
Collection<Date> col = ...
for( Date date : col )
    System.out.println( date );
```

This feature of the Java built-in `for` loop is called the "enhanced" `for` loop (as opposed to the pregenerics, numeric-only `for` loop). The enhanced `for` loop applies only to `Collection` type collections, not `Maps`. `Maps` are another type of beast that really contain two distinct sets of objects (keys and values), so it's not obvious what your intentions would be in such a loop. But because looping over a map does seem reasonable, you can use the `Map` methods `keySet()` or `values()` (or even `entrySet()` if you really wanted each key/value pair as a single entity) to get the right collection from your map that **does** work with this enhanced `for` loop.

A Closer Look: The sort() Method

Poking around in the `java.util.Collections` class, we find all kinds of static utility methods for working with collections. Among them is this goody—the static generic method `sort()`:

```
<T extends Comparable<? super T>> void sort( List<T> list ) { ... }
```

Another nut for us to crack. Let's focus on the last part of the bound:

```
Comparable<? super T>
```

This is a wildcard instantiation of the `Comparable` interface, so we can read the `extends` as `implements` if it helps. `Comparable` holds a `compareTo()` method for some parameter type. A `Comparable<String>` means that the `compareTo()` method takes type `String`. Therefore, `Comparable<? super T>` is the set of instantiations of `Comparable` on `T` and all of its superclasses. A `Comparable<T>` suffices and, at the other end, so does a `Comparable<Object>`. What this means in English is that the elements must be comparable to their own type or some supertype of their own type for the `sort()` method to make use of them. This is sufficient to ensure that the elements can all be compared to one another, but not as restrictive as saying that they must all implement the `compareTo()` method themselves. Some of the elements may inherit the `Comparable` interface from a parent class that knows how to compare only to a supertype of `T`, and that is exactly what is allowed here.