16.2 Collections Overview

A **collection** is a data structure—actually, an object—that can hold references to other objects. Usually, collections contain references to objects of any type that has the *is-a* relationship with the collection's element type. The collections-framework interfaces declare the operations to be performed generically on various types of collections. Figure 16.1 lists some of the collections-framework interfaces. Several implementations of these interfaces are provided within the framework. You may also provide your own implementations.

Interface	**Description**
`Collection`	The root interface in the collections hierarchy from which interfaces `Set`, `Queue` and `List` are derived.
`Set`	A collection that does *not* contain duplicates.
`List`	An ordered collection that *can* contain duplicate elements.
`Map`	A collection that associates keys to values and *cannot* contain duplicate keys. `Map` does not derive from `Collection`.
`Queue`	Typically a *first-in, first-out* collection that models a *waiting line*; other orders can be specified.

Fig. 16.1

Some collections-framework interfaces.

`Object`-Based Collections

The collections-framework classes and interfaces are members of package `java.util`. In early Java versions, the collections framework classes stored and manipulated *only* `Object` references, enabling you to store *any* object in a collection, because all classes directly or indirectly derive from class `Object`. Programs normally need to process *specific* types of objects. As a result, the `Object` references obtained from a collection needed to be *downcast* to an appropriate type to allow the program to process the objects correctly. As we discussed in Chapter 10, downcasting generally should be avoided.

Generic Collections

To eliminate this problem, the collections framework was enhanced with the *generics* capabilities that we introduced with generic `ArrayList`s in Chapter 7 and that we discuss in more detail in Chapter 20, Generic Classes and Methods: A Deeper Look. Generics enable you to specify the *exact type* that will be stored in a collection and give you the benefits of *compile-time type checking*—the compiler issues error messages if you use inappropriate types in your collections. Once you specify the type stored in a generic collection, any reference you retrieve from the collection will have that type. This eliminates the need for explicit type casts that can throw `ClassCastException`s if the referenced object is *not* of the appropriate type. In addition, the generic collections are *backward compatible* with Java code that was written before generics were introduced.

Good Programming Practice 16.1

Avoid reinventing the wheel—rather than building your own data structures, use the interfaces and collections from the Java collections framework, which have been carefully tested and tuned to meet most application requirements.

Choosing a Collection

The documentation for each collection discusses its memory requirements and its methods' performance characteristics for operations such as adding and removing elements, searching for elements, sorting elements and more. Before choosing a collection, review the online documentation for the collection category you're considering (`Set`, `List`, `Map`, `Queue`, etc.), then choose the implementation that best meets your application's needs. Chapter 19, Searching, Sorting and Big O, discusses a means for describing how hard an algorithm works to perform its task—based on the number of data items to be processed. After reading Chapter 19, you'll better understand each collection's performance characteristics as described in the online documentation.