of useful classes. So it pays to become familiar with the contents of the library and how to use the most common classes. The power of abstraction is that we don't usually need to know much (if anything, indeed!) about what the class looks like inside to be able to use it effectively.

If we use a library class, it follows that we will be writing code that creates instances of those classes, and then our objects will be interacting with the library objects. Therefore, object interaction will figure highly in this chapter, also.

You will find that the chapters in this book continually revisit and build on themes that have been introduced in previous chapters. We refer to this in the preface as an "iterative approach." One particular advantage of the approach is that it will help you to gradually deepen your understanding of topics as you work your way through the book.

In this chapter, we also extend our understanding of abstraction to see that it does not just mean hiding detail, but also means seeing the common features and patterns that recur again and again in programs. Recognizing these patterns means that we can often reuse part or all of a method or class we have previously written in a new situation. This particularly applies when looking at collections and iteration.

4.2 The collection abstraction

One of the abstractions we will explore in this chapter is the idea of a *collection*—the notion of grouping things so that we can refer to them and manage them all together. A collection might be large (all the students in a university), small (the courses one of the students is taking), or empty even (the paintings by Picasso that I own!).

If we own a collection of stamps, autographs, concert posters, ornaments, music, or whatever, then there are some common actions we will want to perform to the collection from time to time, regardless of what it is we collect. For instance, we will likely want to *add to* the collection, but we also might want to *reduce it*—say if we have duplicates or want to raise money for additional purchases. We also might want to *arrange it* in some way—by date of acquisition or value, perhaps. What we are describing here are typical *operations* on a collection.

Concept

Collection A collection object can store an arbitrary number of other objects.

In a programming context, the collection abstraction becomes a class of some sort, and the operations would be methods of that class. A particular collection (my music collection) would be an instance of the class. Furthermore, the items stored in a collection instance would, themselves, be objects.

Here are some further collection examples that are more obviously related to a programming context:

- Electronic calendars store event notes about appointments, meetings, birthdays, and so on. New notes are added as future events are arranged, and old notes are deleted as details of past events are no longer needed.
- Libraries record details about the books and journals they own. The catalog changes as new books are bought and old ones are put into storage or discarded.