3.3 Designing Data Types

The ability to create data types turns every programmer into a language designer. You do not have to settle for the types of data and associated operations that are built into the language, because you can create your own data types and write client programs that use them. For example, Java does not have a predefined data type for complex numbers, but you can define `Complex` and write client programs such as `Mandelbrot`. Similarly, Java does not have a built-in facility for turtle graphics, but you can define `Turtle` and write client programs that take immediate advantage of this abstraction. Even when Java does include a particular facility, you might prefer to create separate data types tailored to your specific needs, as we do with `Picture`, `In`, `Out`, and `Draw`.

3.3.1 Complex number (alternate) 434
3.3.2 Counter. 437
3.3.3 Spatial vector. 444
3.3.4 Document sketch 461
3.3.5 Similarity detection 463

Programs in this section

The first thing that we strive for when creating a program is an understanding of the types of data that we will need. Developing this understanding is a *design* activity. In this section, we focus on developing APIs as a critical step in the development of any program. We need to consider various alternatives, understand their impact on both client programs and implementations, and refine the design to strike an appropriate balance between the needs of clients and the possible implementation strategies.

If you take a course in systems programming, you will learn that this design activity is critical when building large systems, and that Java and similar languages have powerful high-level mechanisms that support code reuse when writing large programs. Many of these mechanisms are intended for use by experts building large systems, but the general approach is worthwhile for every programmer, and some of these mechanisms are useful when writing small programs.

In this section we discuss *encapsulation*, *immutability*, and *inheritance*, with particular attention to the use of these mechanisms in data-type design to enable modular programming, facilitate debugging, and write clear and correct code.

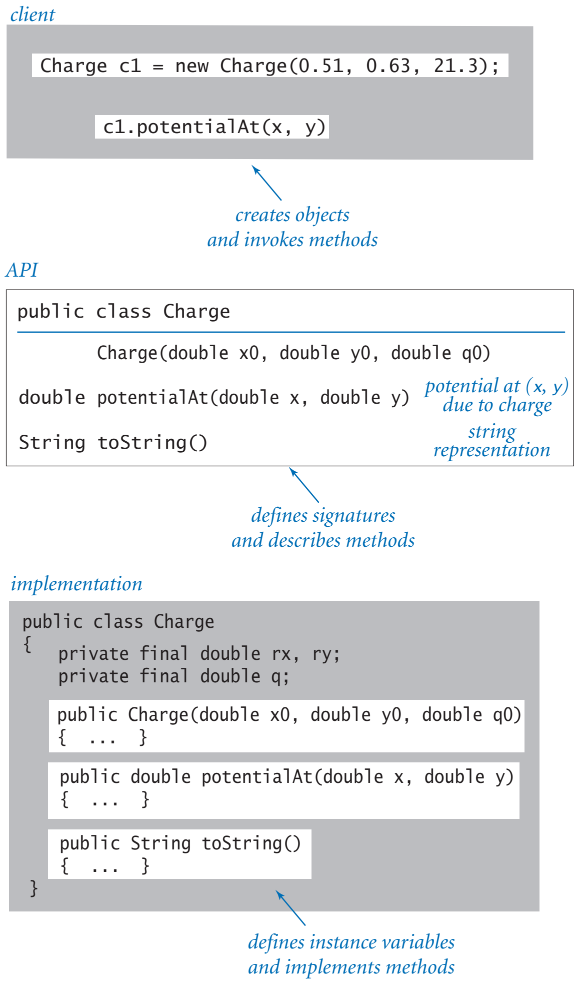
At the end of the section, we discuss Java's mechanisms for use in checking design assumptions against actual conditions at run time. Such tools are invaluable aids in developing reliable software.

Designing APIs In Section 3.1, we wrote client programs that *use* APIs; in Section 3.2, we *implemented* APIs. Now we consider the challenge of *designing* APIs. Treating these topics in this order and with this focus is appropriate because most of the time that you spend programming will be writing client programs.

Often the most important and most challenging step in building software is designing the APIs. This task takes practice, careful deliberation, and many iterations. However, any time spent designing a good API is certain to be repaid in time saved during debugging or with code reuse.

Articulating an API might seem to be overkill when writing a small program, but you should consider writing every program as though you will need to reuse the code someday—not because you know that you will reuse that code, but because you are quite likely to want to reuse *some* of your code and you cannot know *which* code you will need.

Standards. It is easy to understand why writing to an API is so important by considering other domains. From railroad tracks, to threaded nuts and bolts, to MP3s, to radio frequencies, to Internet standards, we know that using a common standard interface enables the broadest usage of a technology. Java itself is another example: your Java programs are clients of the *Java virtual machine*, which is a standard interface that is implemented on a wide variety of hardware and software platforms. By using APIs to separate clients from implementations, we reap the benefits of standard interfaces for every program that we write.


client
creates objects and invokes methods
API
potential at (x, y) due to charge
string representation
defines signatures and describes methods
implementation
defines instance variables and implements methods

```
Charge c1 = new Charge(0.51, 0.63, 21.3);

      c1.potentialAt(x, y)
```

```
public class Charge
-----------------------------------------------------
        Charge(double x0, double y0, double q0)
double  potentialAt(double x, double y)
String  toString()
```

```
public class Charge
{  private final double rx, ry;
   private final double q;

   public Charge(double x0, double y0, double q0)
   {  ...  }

   public double potentialAt(double x, double y)
   {  ...  }

   public String toString()
   {  ...  }
}
```

Object-oriented library abstraction

Specification problem. Our APIs are lists of methods, along with brief English-language descriptions of what the methods are supposed to do. Ideally, an API would clearly articulate behavior for all possible inputs, including side effects, and then we would have software to check that implementations meet the specification. Unfortunately, a fundamental result from theoretical computer science, known as the *specification problem*, says that this goal is actually *impossible* to achieve. Briefly, such a specification would have to be written in a formal language like a programming language, and the problem of determining whether two programs perform the same computation is known, mathematically, to be *unsolvable*. (If you are interested in this idea, you can learn much more about the nature of unsolvable problems and their role in our understanding of the nature of computation in a course in theoretical computer science.) Therefore, we resort to informal descriptions with examples, such as those in the text surrounding our APIs.

Wide interfaces. A *wide interface* is one that has an excessive number of methods. An important principle to follow in designing an API is to *avoid wide interfaces*. The size of an API naturally tends to grow over time because it is easy to add methods to an existing API, whereas it is difficult to remove methods without breaking existing clients. In certain situations, wide interfaces are justified—for example, in widely used systems libraries such as `String`. Various techniques are helpful in reducing the effective width of an interface. One approach is to include methods that are orthogonal in functionality. For example, Java's `Math` library includes trigonometric functions for sine, cosine, and tangent but not secant and cosecant.

Start with client code. One of the primary purposes of developing a data type is to simplify client code. Therefore, it makes sense to pay attention to client code from the start. Often, it is wise to write the client code *before* working on an implementation. When you find yourself with some client code that is becoming cumbersome, one way to proceed is to write a fanciful simplified version of the code that expresses the computation the way you are thinking about it. Or, if you have done a good job of writing succinct comments to describe your computation, one possible starting point is to think about opportunities to convert the comments into code.

Avoid dependence on representation. Usually when developing an API, we have a representation in mind. After all, a data type is a set of values and a set of operations on those values, and it does not make much sense to talk about the operations without knowing the values. But that is different from knowing the *representation* of the values. One purpose of the data type is to simplify client code by allowing it to avoid details of and dependence on a particular representation. For example, our client programs for `Picture` and `StdAudio` work with simple abstract representations of pictures and sound, respectively. The primary value of the APIs for these abstractions is that they allow client code to ignore a substantial amount of detail that is found in the standard representations of those abstractions.

Pitfalls in API design. An API may be *too hard to implement*, implying implementations that are difficult or impossible to develop, or *too hard to use*, creating client code that is more complicated than without the API. An API might be t*oo narrow*, omitting methods that clients need, or *too wide*, including a large number of methods not needed by any client. An API may be *too general*, providing no useful abstractions, or *too specific*, providing abstractions so detailed or so diffuse as to be useless. These considerations are sometimes summarized in yet another motto: *provide to clients the methods they need and no others.*

When you first started programming, you typed in `HelloWorld.java` without understanding much about it except the effect that it produced. From that starting point, you learned to program by mimicking the code in the book and eventually developing your own code to solve various problems. You are at a similar point with API design. There are many APIs available in the book, on the booksite, and in online Java documentation that you can study and use, to gain confidence in designing and developing APIs of your own.

Encapsulation The process of separating clients from implementations by hiding information is known as *encapsulation*. Details of the implementation are kept hidden from clients, and implementations have no way of knowing details of client code, which may even be created in the future.

As you may have surmised, we have been practicing encapsulation in our data-type implementations. In Section 3.1, we started with the mantra *you do not need to know how a data type is implemented to use it*. This statement describes one of the prime benefits of encapsulation. We consider it to be so important that we have not described to you any other way of designing a data type. Now, we describe our three primary reasons for doing so in more detail. We use encapsulation for the following purposes:

- To enable modular programming
- To facilitate debugging
- To clarify program code

These reasons are tied together (well-designed modular code is easier to debug and understand than code based entirely on primitive types in long programs).

Modular programming. The programming style that we have been developing since Chapter 2 has been predicated on the idea of breaking large programs into small modules that can be developed and debugged independently. This approach improves the resiliency of our software by limiting and localizing the effects of making changes, and it promotes code reuse by making it possible to substitute new implementations of a data type to improve performance, accuracy, or memory footprint. The same idea works in many settings. We often reap the benefits of encapsulation when we use system libraries. New versions of the Java system often include new implementations of various data types, but *the APIs do not change*. There is strong and constant motivation to improve data-type implementations because *all* clients can potentially benefit from an improved implementation. The key to success in modular programming is to maintain *independence* among modules. We do so by insisting on the API being the *only* point of dependence between client and implementation. *You do not need to know how a data type is implemented to use it*. The flip side of this mantra is that a data-type implementation can assume that the client knows nothing about the data type except the API.