METHOD NAMES

Names of methods in Ruby follow the same rules and conventions as local variables (except that they can end with `?`, `!`, or `=`, with significance you'll see later). This is by design: methods don't call attention to themselves as methods but rather blend into the texture of a program as, simply, expressions that provide a value. In some contexts you can't tell, just by looking at an expression, whether you're seeing a local variable or a method name—and that's intentional.

Speaking of methods: now that you've got a roadmap to Ruby identifiers, let's get back to some language semantics—in particular, the all-important role of the object and its methods.

1.1.4 Method calls, messages, and Ruby objects

Ruby sees all data structures and values (including scalar (atomic) values like integers and strings, but also including complex data structures like arrays) as *objects*. Every object is capable of understanding a certain set of *messages*. Each message that an object understands corresponds directly to a *method*: a named, executable routine whose execution the object has the ability to trigger.

*Object*s are represented either by literal constructors—like quotation marks for strings—or by variables to which they have been bound. Message-sending is achieved via the special dot operator: the message to the right of the dot is sent to the object on the left of the dot. (There are other, more specialized ways to send messages to objects, but the dot is the most common, most fundamental way.) Consider this example from table 1.1:

```
x = "100".to_i
```

The dot means that the message "to_i" is being sent to the string "100". The string "100" is called the *receiver* of the message. We can also say that the method `to_i` is being *called* on the string "100". The result of the method call—the integer 100—serves as the right-hand side of the assignment to the variable `x`.

Why the double terminology?

Why bother saying both "sending the message 'to_i'" and "calling the method `to_i`"? Why have two ways of describing the same operation? Because they aren't quite the same. Most of the time, you send a message to a receiving object, and the object executes the corresponding method. But sometimes, there is no corresponding method. You can put anything to the right of the dot, and there's no guarantee that the receiver will have a method that matches the message you send.

If that sounds like chaos, it isn't, because objects can intercept unknown messages and try to make sense of them. The Ruby on Rails web development framework, for example, makes heavy use of the technique of sending unknown messages to objects, intercepting those messages, and making sense of them on the fly based on dynamic conditions like the names of the columns in the tables of the current database.