CHAPTER 2

The Elements of a Parser

Parsers are compositions of other parsers, built from instances of a few fundamental parser classes. Learning to create parsers begins with learning to create new compositions of existing parsers.

2.1 What Is a Parser?

A parser is an object that recognizes the elements of a language and translates each element into a meaningful result. A language is a set of strings, and we usually have some way of describing a language as containing all the strings that match a pattern. A fundamental ability of a parser is the ability to declare whether or not a given string belongs to a language. Consider the following language:

```
{"hot coffee",
 "steaming coffee",
 "hot hot hot coffee",
 "hot hot steaming hot coffee",
 ...}
```

This language contains all strings that begin with some sequence of `"hot"` and `"steaming"` and end with `"coffee"`. A parser for this language is an object that can recognize whether or not a given string is an element of this language. For example, this parser will be able to determine that the following is a member of the language:

```
"hot hot hot steaming hot steaming hot coffee"
```

whereas the following is not:

```
"lukewarm coffee"
```

Every parser defines a language—the language that is the set of all strings that the parser recognizes. Notice that such sets are usually infinite. In practice, you never

create a set that contains all the members of a language. Rather, you construct parsers to match patterns of text.

To create a new language you create a new parser object, and this means that you need a `Parser` class. This class does not operate alone; it collaborates with two other classes.

2.2 Parser Collaborations

As a parser recognizes a string, it usually performs a corresponding function, translating the string into a meaningful result. This work forms the *semantics*, or meaning, that the parser brings to a language.

The methods that perform a parser's work must appear as methods of a class. This book places such methods in subclasses of an `Assembler` class. To allow a parser to build a meaningful result, a `Parser` object collaborates with `Assembler` objects that know which work to perform as a parser recognizes parts of a string.

The results of a parser's work must, logically, produce or modify objects. This book associates *work areas* with the text to recognize `Assembly` subclasses. A `Parser` object uses `Assembly` objects to record the progress of `Assembler` objects.

Figure 2.1 shows the three main classes that are used to make practical parsers. These classes complement the `Parser` class hierarchy with `Assembly` and `Assembler` hierarchies. The following summarizes the roles of these three classes.

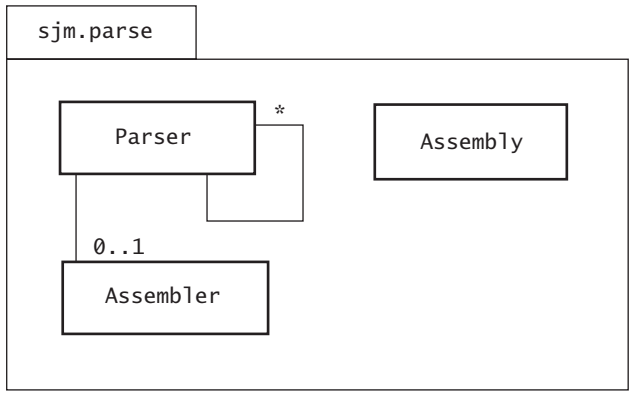

Figure 2.1 The three main classes that collaborate to create a working parser are `Parser`, `Assembly`, and `Assembler`.

- A parser is an object that recognizes a language.
- An assembler helps a parser build a result.
- An assembly provides a parser with a work area.

Objects of these three classes collaborate to recognize an element of a language and to perform work based on the recognition.

This diagram is consistent with the Unified Modeling Language [Booch], as are most of the diagrams in this book. Appendix A explains the features of the Unified Modeling Language that this book applies.

2.3 Assemblies

In practice, we demand more of parsers than simply saying whether a string is a valid member of a language. As a parser recognizes a string, it is useful for the parser to react to the contents of the string and build something. The parser also needs to keep an index of how much of the string it has recognized. An `Assembly` object wraps a string with an index and with work areas for a parser, providing a stack and a target object to work on. For example, a parser that recognizes a data language for coffee might set the target to be a basic `Coffee` object. This parser could work on the target as it recognizes an input string, informing the `Coffee` object of the coffee attributes the string indicates.

If a parser needs to build something based on the content of the string it is recognizing, you can think of the string as a set of instructions. As the parser makes progress recognizing the string, the assembly's index moves forward and the assembly's target begins to take shape. Figure 2.2 shows a partially recognized assembly.

The figure indicates that a parser is working to recognize this string:

```
"place carrier_18 on die_bonder_3"
```

String	"place carrier_18 on die_bonder_3"
index	2
Stack	"carrier_18"
Target	a PlaceCommand object

Figure 2.2 An assembly example. An assembly wraps a string and provides a work area for a parser and its assemblers.

An internal index records the fact that the parser has already recognized two words. On seeing the word "place", the parser set the assembly's target to be a PlaceCommand object. Now the parser has seen "carrier_18" and presumably is about to inform the PlaceCommand object which carrier to place.

2.3.1 The Assembly Class Interfaces

An assembly records the progress of a parser's recognition of an input string. Because the recognition may proceed along different paths, a parser may create multiple copies of an assembly as it tries to determine which is the right path. There are other approaches to modeling the nondeterminism inherent in parsing text, but the parsers in this book consistently use copying. To support this copying and to provide an enumeration interface, the Assembly class in sjm.parse implements the interfaces PubliclyCloneable and Enumeration. Figure 2.3 shows a partial class diagram for Assembly.

PubliclyCloneable is an interface in sjm.utensil. This interface declares that its implementers must implement a public version of the clone() method. The clone() method on java.lang.Object is protected, meaning that unrelated objects cannot use the clone() method. Because Assembly implements PubliclyCloneable, any object can request a clone of an assembly object.

For an assembly to make a clone of itself, it must in turn clone its target, so a target object must itself implement the PubliclyCloneable interface. In fact, the only

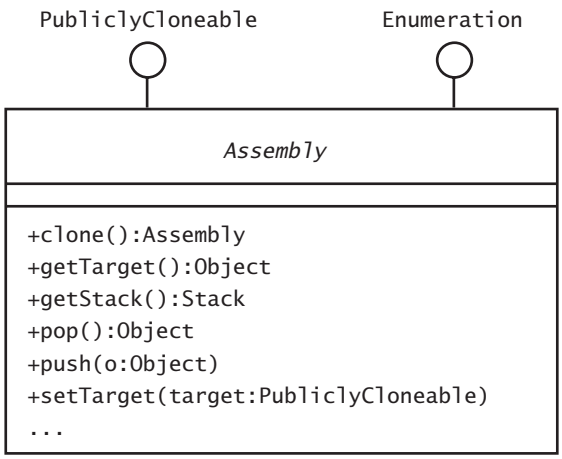

Figure 2.3 The Assembly class. This class implements interfaces that declare that assemblies are cloneable and offer the Enumeration methods hasMoreElements() and nextElement().

requirement for a target object is that it implement this interface, and so `Assembly` declares its target to be of this type. Chapter 3, "Building a Parser," describes how to make a target cloneable (see Section 3.3.8).

Implementing `Enumeration` means that `Assembly` must implement the two methods `hasMoreElements()` and `nextElement()`.

The `Assembly` class itself does not implement these methods, leaving their implementation to subclasses. The two types of assemblies are assemblies of tokens and assemblies of characters.

2.3.2 Token and Character Assemblies

We have said that an assembly is a wrapper around a string. In practice, there are two choices for how to regard the composition of a string. A normal Java `String` object is a string of characters. For example,

```
"hello, world"
```

contains 12 characters, including the blank and the comma. For many parsers, it is far more convenient to treat such a string as a string of *tokens*, where a token can be a word, a number, or a punctuation mark. For example, the string `"hello, world"` can be seen as a string of three tokens:

```
"hello"
','
"world"
```

To allow parsing text as strings of tokens, `Assembly` has two subclasses: `CharacterAssembly` and `TokenAssembly`. Figure 2.4 shows these subclasses and the packages they lie in.

Package `sjm.parse.chars` contains classes that support character-based recognition. Package `sjm.parse.tokens` contains classes that support token-based recognition. Essentially, a `CharacterAssembly` object manipulates an array of characters, and a `TokenAssembly` object manipulates an array of tokens.

The methods `consumed()` and `remainder()` show the amount of input consumed and the amount that remains. The `defaultDelimiter()` method allows the `Assembly` subclasses to decide how to separate their elements. The `TokenAssembly` class places a slash between each token, whereas the `CharacterAssembly` places nothing (an empty string) between characters when showing elements consumed or remaining. The remaining methods of `Assembly` let a calling class request the next element (a character or token) or peek at the next element without removing it.

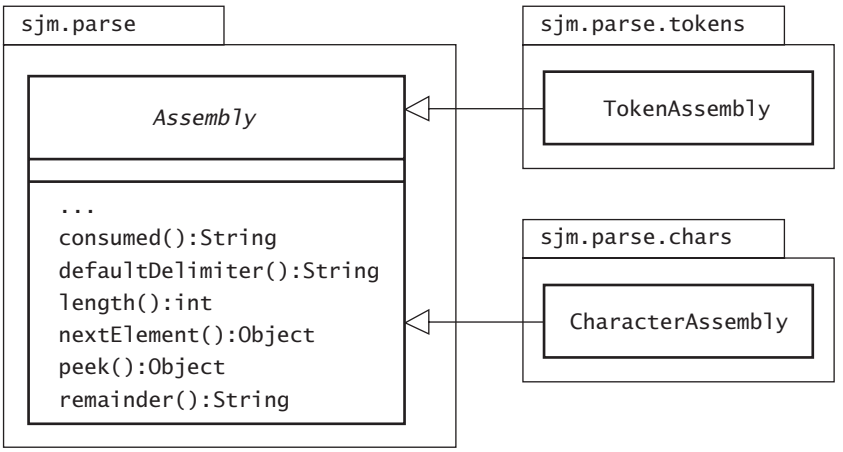

Figure 2.4 The `Assembly` hierarchy. Token and character assemblies implement the `Assembly` methods related to progress in recognizing input.

2.3.3 Tokenizing

Tokenizing a string means breaking the string into logical chunks, primarily words, numbers, and punctuation. This dissection of text is sometimes called *lexical analysis*. Chapter 9, "Advanced Tokenizing," covers tokenization in depth. The point of tokenization is that it can make the task of recognition much easier. It can be much simpler and much more appropriate to describe text as a series of tokens than as a series of characters. For example, consider the string

```
int i = 3;
```

A human reader, especially a Java programmer, will read this as a line of Java. The string contains a data type, a variable name, an equal sign, a number, and a semicolon. It would be accurate, but strange, to describe this string as an `"i"` followed by an `"n"` followed by a `"t"` followed by a blank, another `"i"`, and so on. This string is a pattern of tokens and not a pattern of characters. On the other hand, consider the string

```
"Ja.*"
```

In the right context, this string might describe all words beginning with the letters `"J"` and `"a"`. Here, the string is best understood and most easily recognized as a `"J"` followed by an `"a"` followed by a dot and an asterisk.

The tokenizer used in this book stores tokens in a `Token` class, which is in `sjm.parse.tokens`. Figure 2.5 shows the `Token` class.

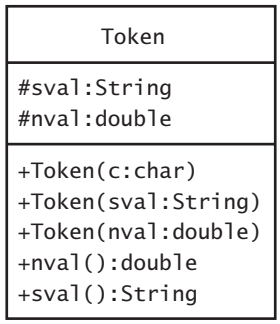

Figure 2.5 The Token class. Typically, a Token is a receptacle for the results of reading a small amount of text, such as a word or a number.

You will most likely encounter tokens in practice when you need to retrieve a token that a terminal has stacked. If a token contains a string, you can retrieve its string value using the `sval()` method. If a token contains a number, you can retrieve the number using the `nval()` method. You can also construct tokens from a string or number, or from a single character that the `Token` constructor converts into a string.

2.3.4 Default and Custom Tokenization

It is imprecise to say that tokenizing breaks a string into "words, numbers, and punctuation." We can find examples that challenge exactly what is and is not a separate token. For example, is an underscore part of a word, or is it a punctuation mark? Does the string `">="` contain one token or two?

To begin writing parsers you need to know what to expect from the class `TokenAssembly`. When you construct a `TokenAssembly` from a string, `TokenAssembly` breaks the string into tokens, relying primarily on the services of another class, `Tokenizer`.

Class `Tokenizer` provides a good set of default rules for how to divide text into tokens. For example, a default `Tokenizer` object properly tokenizes this string:

```
"Let's 'rock and roll'!"
```

The `Tokenizer` object treats the apostrophe in `"Let's"` as part of the word, but it treats the single quotation marks around `"rock and roll"` as single quotes. Here is code that shows this:

```
package sjm.examples.introduction;

import sjm.parse.tokens.*;
import sjm.utensil.*;
```

```
/**
 * Show that apostrophes can be parts of words and can
 * contain quoted strings.
 */
public class ShowApostrophe {

public static void main(String[] args) {
    String s = "Let's 'rock and roll'!";
    TokenAssembly a = new TokenAssembly(s);
    while (a.hasMoreElements()) {
        System.out.println(a.nextElement());
    }
}
}
```

Running this class prints the following:

```
Let's
'rock and roll'
!
```

You may find that the default tokenization does not fit the purposes of your language. For example, you may need to allow blanks to appear inside words. For this and other types of customization, consult Chapter 9, "Advanced Tokenizing."

2.3.5 Assembly Appearance

The preceding example shows the effect of printing one element at a time from an assembly. If you print an entire assembly, it shows its stack, all its elements, and the position of its index. For example:

```
package sjm.examples.introduction;

import sjm.parse.tokens.*;

/**
 * Show how an assembly prints itself.
 */
public class ShowAssemblyAppearance {

public static void main(String[] args) {

    String s1 = "Congress admitted Colorado in 1876.";
    System.out.println(new TokenAssembly(s1));

    String s2 = "admitted(colorado, 1876)";
    System.out.println(new TokenAssembly(s2));
}
}
```

Running this class prints the two `TokenAssembly` objects:

```
[]^Congress/admitted/Colorado/in/1876.0
[]^admitted/(/colorado/,/1876.0/)
```

Both assemblies print their stacks, which are empty and appear as a pair of brackets. These stacks can gain contents only when a parser parses the assembly. Both assemblies show all their elements, separated by slashes. The caret symbolizes the amount of progress a parser has made in recognizing the assembly. Because this example has no parser, both indexes are at the beginning. Note that assemblies include no description of their target when they print. When you want a target to print, you retrieve the target from the assembly and print the target.

2.3.6 Assembly Summary

The `Assembly` classes wrap a string and provide a work area for a parser to record progress in recognizing the string and building a corresponding object. The assembly may tokenize the string, and that simplifies the parser's job of recognition.

2.4 The Parser Hierarchy

A parser is an object that recognizes strings. The examples in this book always wrap a string to be recognized in an `Assembly` object, as an assembly of either characters or tokens. Figure 2.6 shows the public interface of `Parser` objects that the examples use.

The `Parser` class in `sjm.parse` recognizes assemblies and returns assemblies. Often, you want to see whether a parser can recognize an entire string. If the parser can recognize the assembly you provide, its `completeMatch()` method returns a new assembly with its index at the end. If the parser does not recognize an assembly, it returns `null`. If you want a parser only to match as much of a string as it can, `bestMatch()`

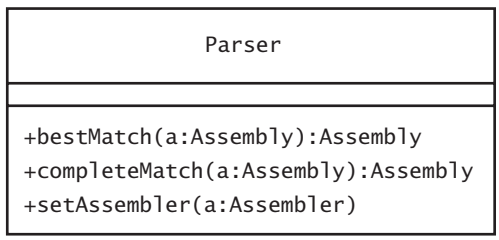

Figure 2.6 `Parser` class. A parser is an object that can recognize a string (wrapped in an `Assembly` object) and assemble a result.

returns an assembly with its index moved as far forward as possible through the supplied assembly. The `Parser` class also has methods for matching collections (`Vectors`) of assemblies and for associating assemblers with the parser.

2.4.1 The Composition of a Parser

The parsers in this book follow the *composite* pattern [Gamma et al.]. This means that some classes in the `Parser` hierarchy define aggregations of other parsers. Other classes in the hierarchy are *terminals*, which are parsers that can match an assembly without the aid of other parsers. They get their name from the fact that they terminate the recursion inherent in the idea that parsers are compositions of parsers. Because there are many types of terminals, there are many ways a parser can look for characters or tokens to appear in an assembly. This also means that there are many terminal classes.

Three types of *nonterminals*—parsers that are compositions of other parsers—are sufficient to build an infinite variety of parsers. Figure 2.7 shows the top of the hierarchy of parser classes. The three types of composition are repetition, sequence, and alternation. Many of the examples that lie ahead show how to use these compositions. In brief,

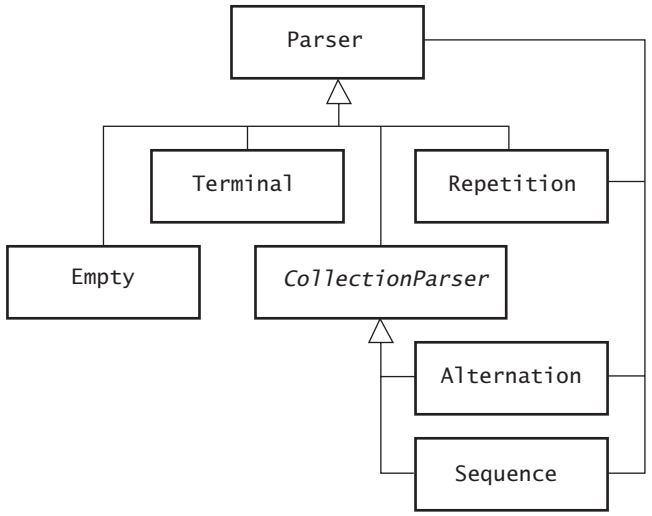

Figure 2.7 Parser hierarchy. The hierarchy of parsers contains three concrete classes that contain other parsers—`Repetition`, `Alternation`, and `Sequence`—and two that do not: `Terminal` and `Empty`.

- A *repetition* matches its underlying parser repeatedly against an assembly.
- A *sequence* is a collection of parsers, all of which must in turn match against an assembly for the sequence parser to successfully match.
- An *alternation* is a collection of parsers, any one of which can successfully match against an assembly.

It is also helpful, if not strictly necessary, to have an empty parser:

- An *empty* parser reports a successful match without consuming any elements from the assembly.

As Figure 2.7 shows, repetitions, alternations, and sequences are compositions of other parsers. The overall composition of a parser can be arbitrarily deep. For example, a parser for a programming language might itself be a sequence of declaration statements followed by executable statements. The parser for executable statements is typically an alternation of different types of statements. Each of these alternatives might be a sequence, and so on.

The composite nature of parsers implies that you must create simple parsers before you make complex ones. The simplest parsers are terminals, which match string assemblies without using other parsers.

2.5 Terminal Parsers

The examples in this book wrap strings either as assemblies of characters or assemblies of tokens. Each terminal must be geared toward one of these two types of assemblies. Each time a terminal asks an assembly for its `nextElement()`, the terminal must anticipate receiving either a character or a token; the terminal decides whether the character or token it receives is a match. Figure 2.8 shows the hierarchy of terminals that work with tokens. The subclasses of `Terminal` in Figure 2.8 are members of the package `sjm.parse.tokens`.

2.5.1 Using Terminals

For an example of how to use a terminal, consider the following program:

```
package sjm.examples.introduction;

import sjm.parse.*;
import sjm.parse.tokens.*;
```

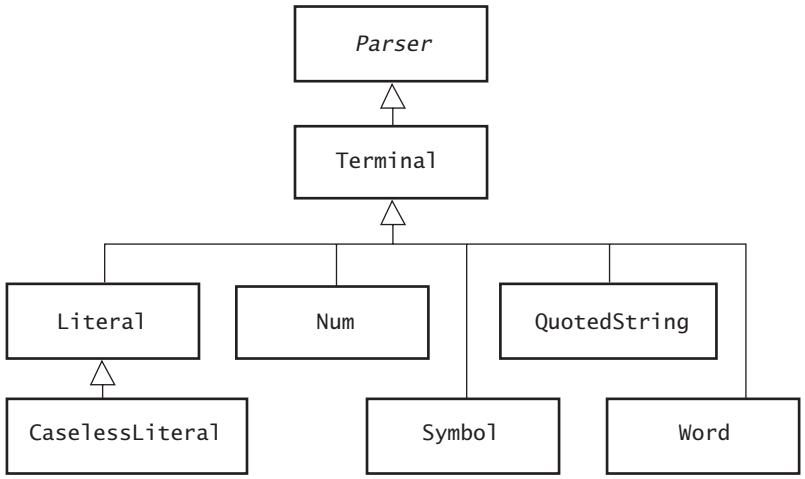

Figure 2.8 Token terminals. The subclasses of `Terminal` shown here expect an assembly's elements to be complete tokens.

```
/**
 * Show how to recognize terminals in a string.
 */
public class ShowTerminal {

public static void main(String[] args) {
    String s = "steaming hot coffee";
    Assembly a = new TokenAssembly(s);
    Parser p = new Word();
    while (true) {
        a = p.bestMatch(a);
        if (a == null) {
            break;
        }
        System.out.println(a);
    }
}
}
```

The parser `p` is a single `Word` object. The assembly `a` is a `TokenAssembly` from the string `"steaming hot coffee"`. The code asks the parser to return the parser's best match against this assembly. Because the parser matches any word, its best match will be a new assembly with the same string and an index moved forward by one word. The first line the program prints is

```
[steaming]steaming^hot/coffee
```

This output demonstrates how an assembly represents itself as a string. The assembly first shows the contents of its stack, `[steaming]`. The default behavior of all `Terminal` objects is to push whatever they recognize onto the assembly's stack. You can prevent this pushing by sending a `Terminal` object a `discard()` message.

After the stack, the assembly shows its tokens, separated by slashes, and shows the location of the index.

The `while` loop of the program asks the parser to return its best match against the revised assembly. In this pass, the program prints

```
[steaming, hot]steaming/hot^coffee
```

Now the stack contains two words, and the index has moved past two words. In the next pass, the program prints

```
[steaming, hot, coffee]steaming/hot/coffee^
```

Now the stack has all three words, and the index is at the end. In the next pass, the `while` loop will again ask the parser for its best match. Because the index of the assembly is at the end, there is no match; the `bestMatch()` method returns `null`, and the logic breaks out of the loop.

2.5.2 Word Terminals

In the preceding example, the parser `p` recognizes the set of all words. To be more specific, the language that `p` recognizes is the set of all strings that class `Tokenizer` in package `sjm.parse.tokens` recognizes as words.

2.5.3 Num Terminals

As a language designer, you can decide what constitutes a number. The default value of `Tokenizer` will find that the string

```
"12 12.34 .1234"
```

contains the numbers 12.0, 12.34, and 0.1234. The default tokenizer will not recognize exponential notation or anything beyond digits, a decimal point, and more digits. Here is a program that shows the default tokenization of numbers:

```
package sjm.examples.introduction;

import sjm.parse.tokens.*;
import sjm.utensil.*;
```

```
/**
 * Show what counts as a number.
 */
public class ShowNums {

public static void main(String[] args) {
    String s = "12 12.34 .1234 1234e-2";
    TokenAssembly a = new TokenAssembly(s);
    while (a.hasMoreElements()) {
        System.out.println(a.nextElement());
    }
}
}
```

Running this class prints the following:

```
12.0
12.34
0.1234
1234.0
e-2
```

Note that by default the tokenizer does not comprehend the exponential notation of the last number. Chapter 9, "Advanced Tokenizing," explains how to change the tokenization of a string to allow for exponential notation. In your own languages, you can use the default number recognition in `Tokenizer` and simply disallow exponential notation. Alternatively, you can customize a `Tokenizer` object to allow exponential, imaginary, and other types of notation for numbers.

2.5.4 Literals

A *literal* is a specific string. Consider the following declaration:

```
"int iq = 177;"
```

In this declaration, the word `"int"` must be a specific, literal value, whereas the variable name that follows it can be any word. To create a `Literal` parser, specify in a constructor the string the parser needs to match, as in

```
Literal intType = new Literal("int");
```

2.5.5 Caseless Literals

Sometimes you want to let the people using your language enter specific values without worrying about capitalization. For example, in a coffee markup language, you might establish a roast parameter that looks for strings such as

```
<roast>French</roast>
```

In building the parser to recognize this parameter, you might include a literal value for the roast parameter using the following object:

```
new Literal("roast")
```

It would be more flexible to allow your language user to type `"Roast"` or `"ROAST"` in addition to the all-lowercase `"roast"`. To achieve this, use a *caseless* literal in place of the normal literal using an object such as:

```
new CaselessLiteral("roast")
```

2.5.6 Symbols

A *symbol* is generally a character that stands alone as its own token. For example, semicolons, equal signs, and parentheses are all characters that a typical tokenizer treats as symbols. In particular, both `StreamTokenizer` in `java.io` and `Tokenizer` in `sjm.parse.tokens` treat such characters as symbol tokens. The default instance of `Tokenizer` treats the following characters as symbols:

```
! # $ % & ( ) * + , : ; < = > ? @ ` [ \ ] ^ _ { | } ~
```

In addition, the default value of `Tokenizer` treats the following multicharacter sequences as symbols:

```
!=    <=    >=    :-
```

These symbols commonly represent "not equal," "less than or equal," "greater than or equal," and "if." The "if" symbol is common in logic languages (see Chapter 13, "Logic Programming"). The `Tokenizer` class gives you complete control over which characters and multicharacter sequences a `Tokenizer` object returns as symbols.

2.5.7 Quoted Strings

You may want to allow users of your language to enter quoted strings in some contexts—for example, when you want to allow a string value to contain a blank. The following program accepts a secret identity as a quoted string:

```
package sjm.examples.introduction;

import sjm.parse.*;
import sjm.parse.tokens.*;
```

```
/**
 * Show how to recognize a quoted string.
 */
public class ShowQuotedString {

public static void main(String[] args) {
    Parser p = new QuotedString();
    String id = "\"Clark Kent\"";
    System.out.println(p.bestMatch(new TokenAssembly(id)));
}
}
```

This program creates and applies a parser that recognizes a quoted string. Running this program prints

```
["Clark Kent"]"Clark Kent"^
```

The output shows that the parser matches the entire string, moving the ^ index past the token and stacking the token. Note that `"Clark Kent"` is a single token even though it contains a blank. Also note that the token contains the quote symbols themselves.

2.6 Composite Parsers

Every parser is either a terminal or a composite. A terminal is a parser that recognizes all or part of a string without using other parsers. For example, a terminal might check that the next word in a string is `"hello"`. A composite parser is a parser that accomplishes its task by using other parsers. The three fundamental ways to compose parsers are repetition, sequence, and alternation.

2.6.1 Repetition

Like a tree that is composed of other trees and leaves, a composite typically contains several subcomponents. A repetition contains only one component: another parser that the repetition applies repeatedly against an assembly to recognize. Figure 2.9 shows this structure.

The repetition can match its underlying parser zero times, one time, two times, or `n` times, where `n` can be arbitrarily large. When a repetition matches an assembly, many versions of the assembly are possible, with the resulting assemblies' indexes at different positions.

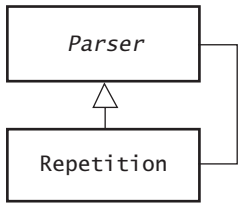

Figure 2.9 The `Repetition` class. Composites usually contain one or more subparsers, but a `Repetition` object contains only one.

For example, consider the phrase `"steaming hot coffee"`. If you match words from the beginning of this phrase, you can match zero, one, two, or three words. You could mark these relative states of progress as follows:

```
"^steaming hot coffee"
"steaming^ hot coffee"
"steaming hot^ coffee"
"steaming hot coffee^"
```

The power of the idea of repetition is that it allows you to describe a variable number of language elements. The challenge this creates for parsers is that a repetition does not know the right number of times to match. It may appear that the right answer is for a repetition to match as many times as possible. However, it turns out that this approach is not always sufficient. For example, you could define a language as a repetition of words, followed by the specific word `"coffee"`. In this case, the "right" number of words for a repetition object to match is two, stopping before the word `"coffee"`.

To avoid the problem of deciding how many times to match, a simple approach is to return a set of all possible matches. This is the approach taken by class `Repetition` in package `sjm.parse`. In fact, each of the composite parsers in the code for this book takes this approach. The subclasses of `Parser` implement `match()`, which accepts a `Vector` of `Assembly` objects and returns a `Vector` of `Assembly` objects. To comply with Java 1.1.6 code, the sample code uses `Vector` instead of `Set`.

To see `match()` and `Repetition` in action, consider the following program:

```
package sjm.examples.introduction;

import java.util.*;
import sjm.parse.*;
import sjm.parse.tokens.*;
```

```
/**
 * Show that a <code>Repetition</code> object creates
 * multiple interpretations.
 */
public class ShowRepetition {

public static void main(String[] args) {
    String s = "steaming hot coffee";
    Assembly a = new TokenAssembly(s);
    Parser p = new Repetition(new Word());

    Vector v = new Vector();
    v.addElement(a);

    System.out.println(p.match(v));
}
}
```

This prints the following (with some added whitespace):

```
[
 []^steaming/hot/coffee,
 [steaming]steaming^hot/coffee,
 [steaming, hot]steaming/hot^coffee,
 [steaming, hot, coffee]steaming/hot/coffee^
]
```

Note that this sample program uses `match()` instead of `bestMatch()`. The `match()` method accepts a `Vector` of assemblies and returns a `Vector` of assemblies. This reflects the underlying mechanics of the parsers in `sjm.parse`. Chapter 10, "Matching Mechanics," describes in detail how matching works.

To show the results of a `Repetition` object, the example prints the results of the underlying `match()` method. When the code constructs a new `Repetition` object, it passes in the parser to repeat. Because a `Repetition` always repeats a single parser, the only constructor for this class requires its caller to supply the parser to repeat. The parser `p` creates a set (or `Vector`) of possible matches against the input. The output shows the outermost pair of brackets that the `Vector` object prints. Inside the vector are four assemblies, with four states of possible matching, including zero, one, two, or three word matches.

2.6.2 Alternation and Sequence

An alternation is a composite of other parsers, any of which may match a provided assembly. A sequence is a composite of other parsers, each of which must match in turn. Figure 2.10 shows the structure of these classes.

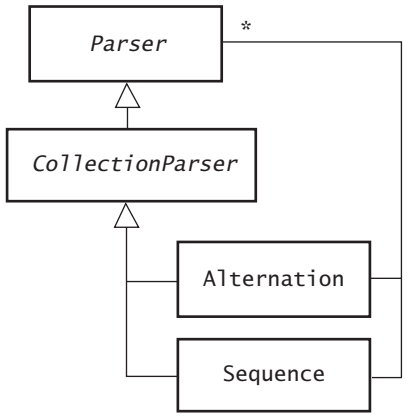

Figure 2.10 The `Alternation` and `Sequence` classes. `Alternation` objects and `Sequence` objects are compositions of other parsers.

2.6.3 Composing a Parser

Creating a new parser is a matter of composing it from an existing selection of terminal and composite parsers. A small example that uses each of the composite parsers is as follows. Consider this string:

```
"hot hot steaming hot coffee"
```

Let us say that this string is a member of a language that, in general, contains strings with any combination of `"hot"` and `"steaming"` adjectives, followed by `"coffee"`. In other words, this language contains all sentences that are

- A sequence of *adjectives* followed by the literal `"coffee"`

where adjectives are

- A repetition of an adjective

and an adjective is:

- An alternation, a choice between `"hot"` and `"steaming"`

Having described this language fairly precisely, we can create a parser object directly from the language description, as the following program shows.

```
package sjm.examples.introduction;

import sjm.parse.*;
import sjm.parse.tokens.*;
```

```
/**
 * Show how to create a composite parser.
 */
public class ShowComposite {

public static void main(String[] args) {

    Alternation adjective = new Alternation();
    adjective.add(new Literal("steaming"));
    adjective.add(new Literal("hot"));

    Sequence good = new Sequence();
    good.add(new Repetition(adjective));
    good.add(new Literal("coffee"));

    String s = "hot hot steaming hot coffee";
    Assembly a = new TokenAssembly(s);
    System.out.println(good.bestMatch(a));

}
}
```

Running this class prints the following:

```
[hot, hot, steaming, hot, coffee]
hot/hot/steaming/hot/coffee^
```

The result of sending `bestMatch()` to `good` is an assembly with all five words placed on its stack and with its index at the end of its tokens.

Figure 2.11 shows an object diagram of `good`.

In Figure 2.11, the box that contains `good:Sequence` means that `good` is an object of type `Sequence`. The underline means that `good` is an object, and the colon separates the object from its type. Some of the objects, such as `:Repetition`, are nameless in the diagram, as they are in the sample program. The literals in the diagram indicate their literal values, although this does not imply that the variable `literal` is publicly accessible.

The leaf nodes of this object diagram are all literals, which are terminals. A parser's composition always terminates with terminals, hence the name "terminal."

2.6.4 The Empty Parser

Sometimes it is convenient to have a parser that returns successfully, having matched nothing at all. For example, you might want to match a list of equipment, recognizing strings such as

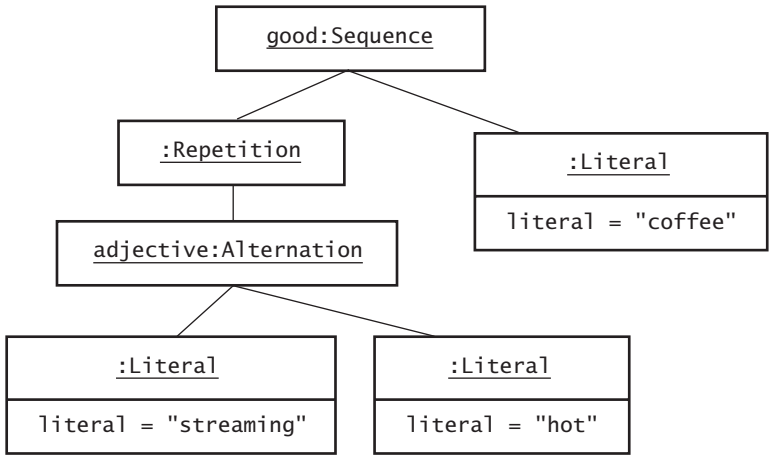

Figure 2.11 Good coffee. This object diagram shows the composition of a good parser, which recognizes a description of a good cup of coffee.

```
"[die_bonder_2, oven_7, wire_bonder_3, mold_1]"
"[]"
"[mold_1]"
```

The contents of the list may be empty, or it may contain a single name or a series of names separated by commas. To match such strings, you could say that the content of a list is either empty or an actual list. An actual list is a name followed by a repetition of (comma, word) sequences. Here is an example that uses `Empty` to help match these lists:

```
package sjm.examples.introduction;

import sjm.parse.*;
import sjm.parse.tokens.*;

/**
 * Show how to put the <code>Empty</code> class to good use.
 */
public class ShowEmpty {

public static void main(String args[]) {

    Parser empty, commaTerm, actualList, contents, list;
    empty = new Empty();
    commaTerm = new Sequence()
        .add(new Symbol(',').discard())
        .add(new Word());
```

```
        actualList = new Sequence()
            .add(new Word())
            .add(new Repetition(commaTerm));

        contents = new Alternation()
            .add(empty)
            .add(actualList);

        list = new Sequence()
            .add(new Symbol('[').discard())
            .add(contents)
            .add(new Symbol(']').discard());

        String test[] = new String[]{
            "[die_bonder_2, oven_7, wire_bonder_3, mold_1]",
            "[]",
            "[mold_1]"};

        for (int i = 0; i < test.length; i++) {
            TokenAssembly a = new TokenAssembly(test[i]);
            System.out.println(
                list.completeMatch(a).getStack());
        }
    }
    }
```

This prints the following:

```
[die_bonder_2, oven_7, wire_bonder_3, mold_1]
[]
[mold_1]
```

The brackets in the output are part of the way a `Stack` object represents itself; they are not the brackets from the input. The output shows that the `list` parser recognizes lists with many elements, zero elements, or one element.

2.6.5 Parser Summary

In general, a parser is an object that recognizes a string. The code in this book wraps the string to be recognized in either a `CharacterAssembly` object or a `TokenAssembly` object. This approach simplifies the parser's job by giving the parser a place to work as it begins to recognize the contents of the string. In the next section we cover assemblers, which are objects that work on an assembly as the parser recognizes it.

You can create simple parsers by creating instances of subclasses of `Terminal`. You can create more-complex parsers by composing new parsers as sequences, alternations, or repetitions of other parsers. This method scales up, so you can create advanced parsers, building them from other terminal and composite parsers.

2.7 Assemblers

To this point, this book has emphasized *recognizing* the strings that make up a language. In practice, you usually want to react to this recognition by doing something. What you do gives meaning, or semantics, to your parser. What you might want to do is unbounded. A common task is the creation of an object that a string describes. For example, consider a file that contains the following textual description of types of coffees:

```
Brimful, Regular, Kenya, 6.95
Caress (Smackin), French, Sumatra, 7.95
Fragrant Delicto, Regular/French, Peru, 9.95
Havalavajava, Regular, Hawaii, 11.95
Launch Mi, French, Kenya, 6.95
Roman Spur (Revit), Italian, Guatemala, 7.95
Simplicity House, Regular/French, Columbia, 5.95
```

You can use a parser to verify that each line of this file is an element of a coffee description language. In addition to this verification, in practice you often want to produce objects from such textual data. In this case, you would want your coffee parser to create a collection of coffee objects so that the meaning of the input is a corresponding set of objects. Assemblers make this meaning possible.

2.7.1 Parsers Use Assemblers

Any `Parser` object can have an associated assembler. When a parser is a composite, it can have its own assembler, and any of its subparsers can have their own assemblers, all the way down to the terminals. Figure 2.12 shows the relationship of parsers and assemblers.

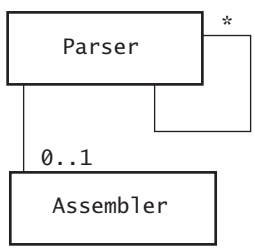

Figure 2.12 The `Parser`/`Assembler` relation. A parser can have an assembler, which it uses to work on an assembly after the parser matches against the assembly.

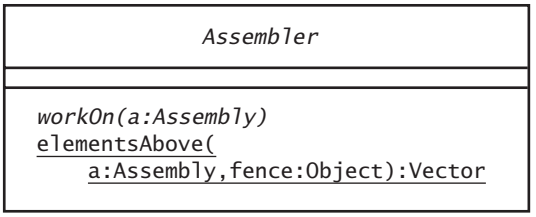

Figure 2.13 The `Assembler` class. The `Assembler` class is abstract, requiring subclasses to implement the `workOn()` method.

Figure 2.13 shows the `Assembler` class. After a parser matches successfully against an assembly, it calls its assembler's `workOn()` method.

2.7.2 Assemblers Work On Assemblies

When an assembler's `workOn()` method is called, the assembler knows that its parser has just completed a match. For example, a parser that recognizes the preceding coffee text will include a parser to match only the country portion of a coffee description as part of building a target `Coffee` object. The terminal that matches `"Sumatra"` in the middle of

```
Caress (Smackin), French, Sumatra, 7.95
```

will place `"Sumatra"` on the assembly's stack, something that terminals do by default. After this terminal matches, it will ask its assembler to go to work. Its assembler might look like this:

```
package sjm.examples.coffee;

import sjm.parse.*;
import sjm.parse.tokens.*;

/**
 * This assembler pops a string and sets the target
 * coffee's country to this string.
 */
public class CountryAssembler extends Assembler {

public void workOn(Assembly a) {
    Token t = (Token) a.pop();
    Coffee c = (Coffee) a.getTarget();
    c.setCountry(t.sval().trim());
}
}
```

This class assumes that a parser will call this class's `workOn()` method in the appropriate context. This includes the assumption that the top of the assembly's stack is, in fact, a token containing the name of a country. The assembler also assumes that the assembly's target object is a `Coffee` object. In practice, these assumptions are safe, because the parser that calls this assembler knows with certainty that it has just matched a country name as part of an overall task of recognizing a type of coffee.

2.7.3 Elements Above

The `Assembler` class includes the static method `elementsAbove()`, which lets you design a parser that stacks a *fence* and later retrieves all the elements above the fence. A fence is any kind of marker object. For example, suppose a list parser is matching this list:

```
"{Washington Adams Jefferson}"
```

The list parser might allow the opening curly brace to go on an assembly's stack while discarding the closing curly brace. This lets the opening brace token act as a fence. After matching the elements in the list, the parser can retrieve them with an assembler that removes all the elements on the stack above the `'{'` token. The parser's assembler can use the `elementsAbove()` method, passing it the assembly and the opening brace as the fence to look for. For example:

```
package sjm.examples.introduction;

import sjm.parse.*;
import sjm.parse.tokens.*;

/**
 * Show how to use <code>Assembler.elementsAbove()</code>.
 */
public class ShowElementsAbove {

public static void main(String args[]) {

    Parser list = new Sequence()
        .add(new Symbol('{'))
        .add(new Repetition(new Word()))
        .add(new Symbol('}').discard());

    list.setAssembler(new Assembler() {
        public void workOn(Assembly a) {
            Token fence = new Token('{');
            System.out.println(elementsAbove(a, fence));
        }
    });
```

```
        list.bestMatch(
            new TokenAssembly("{ Washington Adams Jefferson }"));
    }
    }
```

This class's `main()` method creates a list parser that implements this pattern:

```
list = '{' Word* '}';
```

In the code, the `list` parser allows the opening brace to go onto an assembly's stack, and discards the closing brace. The `main()` method gives `list` an assembler by using an anonymous subclass of `Assembler` that retrieves the elements that match after the opening brace.

Note that the opening brace terminal stacks a *token* and not a brace character or a string. Thus, the fence the assembler looks for is a `'{'` token. In this example the assembler retrieves the elements above the opening brace and prints them. Running this class prints the vector

```
[Jefferson, Adams, Washington]
```

The Logikus parser in Chapter 14, "Parsing a Logic Language," uses the `elementsAbove()` method to collect the elements of a Logikus list. The Sling parser in Chapter 16, "Parsing an Imperative Language," uses `elementsAbove()` to collect the statements in the body of a `for` loop.

2.8 Summary

A parser is an object that recognizes the elements of a language and builds meaningful target objects based on the text it recognizes. To achieve this combination of recognition and building, this book uses a collaboration of three hierarchies of classes: `Parser`, `Assembler`, and `Assembly`. An assembly contains both the text to be recognized and work areas that house the results of work completed in building a result. Assemblers plug in to parsers and work on assemblies as the parsers recognize text.

The `Assembly` hierarchy has two subclasses: one that provides individual characters as its elements and a second one that provides tokens.

The `Parser` hierarchy has four main subclasses: `Terminal`, `Sequence`, `Alternation`, and `Repetition`. The `Terminal` class itself is at the top of a wide variety of subclasses. You need a `Terminal` subclass for each group of characters or tokens you want to recognize.

You can compose new parsers as sequences, alternations, and repetitions of other parsers, creating an infinite variety of parsers from these few building blocks.