Chapter 10

Recurring Themes

In Chapter 9, we learned how `while` loops can be used to command a computer to execute millions of instructions without having to actually type in millions of instructions. Now that we know how to make a computer perform millions of operations, we would like to be able to write programs that process millions of pieces of information. We have seen how to manipulate small quantities of information in a program. Names have been essential to this process. In order to work with information, we must first associate a name with the value or object to be manipulated so that we can write instructions telling the computer to apply methods or operators to the data. It is, however, hard to imagine writing a program containing millions of instance variable declarations! There must be another way.

We encounter programs that process large collections of information every day. The book you are reading contains roughly a million words. The program used to format this book has the ability to manage all of these words, decide how many can fit on each page, etc. When you go to Google and search for web pages about a particular topic, the software at Google somehow has to examine data describing millions of web pages to find the ones of interest to you. When you load a picture from your new 8 megapixel digital camera into your computer, the computer has to store the 8 million data values that describe the colors of the 8 million dots of color that make up the image and arrange to display the right colors on your screen.

In this and the following chapter, we will explore two very different ways of manipulating large collections of information in a program. The technique presented in the following chapter involves a new feature of the Java language called arrays. The technique presented in this chapter, on the other hand, is interesting because it does not involve any new language features. It merely involves using features of Java you already know about in new ways.

The technique we discuss in this chapter is called *recursion*. Recursion is a technique for defining new classes and methods. When we define new classes and methods, we usually use the names of other classes and methods to describe the behavior of the class being defined. For example, the definition of the very first class we presented, `TouchyButton` depended on the `JButton` class and the definition of the `buttonClicked` method in that class depended on the `add` method of the `contentPane`. Recursive definitions differ from other examples of method and class definitions we have seen in one interesting way. In a recursive definition, the name of the method or class being defined is used as part of the definition.

At first, the idea of defining something in terms of itself may seem silly. It certainly would not be helpful to look up some word in the dictionary and find a definition that assumed you already

knew what the word meant:[1]

recursion (noun)

- A formula that generates the successive terms of a recursion.

In programming, surprisingly, recursive definitions provide a way to describe complex structures simply and effectively.

10.1 Long Day's Journey

An ancient proverb explains that

> "A journey of a thousand miles begins with a single step."
> Lao-Tzu, Chinese Philosopher (604 BC - 531 BC)

In today's world, however, a long journey often begins with getting driving instructions from maps.google.com or some similar site. Figure 10.1 shows an example of the type of information one can obtain at such sites.

The web page in Figure 10.1 shows the directions requested in several forms. On the right side of the page, the route is traced on a map showing the starting point and destination. To the left, the directions are expressed step-by-step in textual form.

1.	Head northwest on E 69th St toward 2nd Ave	0.5 mi
2.	Head southwest on 5th Ave toward E 68th St	0.5 mi
3.	Turn right at Central Park S	0.4 mi
4.	Turn left at 7th Ave	0.2 mi

The data required to generate these instructions is an example of a collection. It is a collection of steps. We will explore how to define a class that can represent such a collection of driving instructions as our first example of a recursive definition in Java.

In our introduction to loops, we stressed that it is important to know how to perform an operation once before attempting to write a loop to perform the operation repeatedly. Similarly, if we want to define a collection of similar objects, we better make sure we know how to represent a single member of the collection first. Therefore, we will begin by defining a very simple class to represent a single step from a set of driving directions.

The code for such a `Step` class is shown in Figure 10.2. It is a very simple class. We use three pieces of information to describe a step in a set of driving directions. The instance variable

[1]The definition used as an example here was actually found in the online version of the American Heritage Dictionary. I must, however, admit to a bit of cheating. The American Heritage Dictionary provides two definitions for recursion. The entry listed in the text is the second. The first definition provided does not depend on the word "recursion", but provides little insight that will be helpful here:

recursion (noun)

- An expression, such as a polynomial, each term of which is determined by application of a formula to preceding terms.

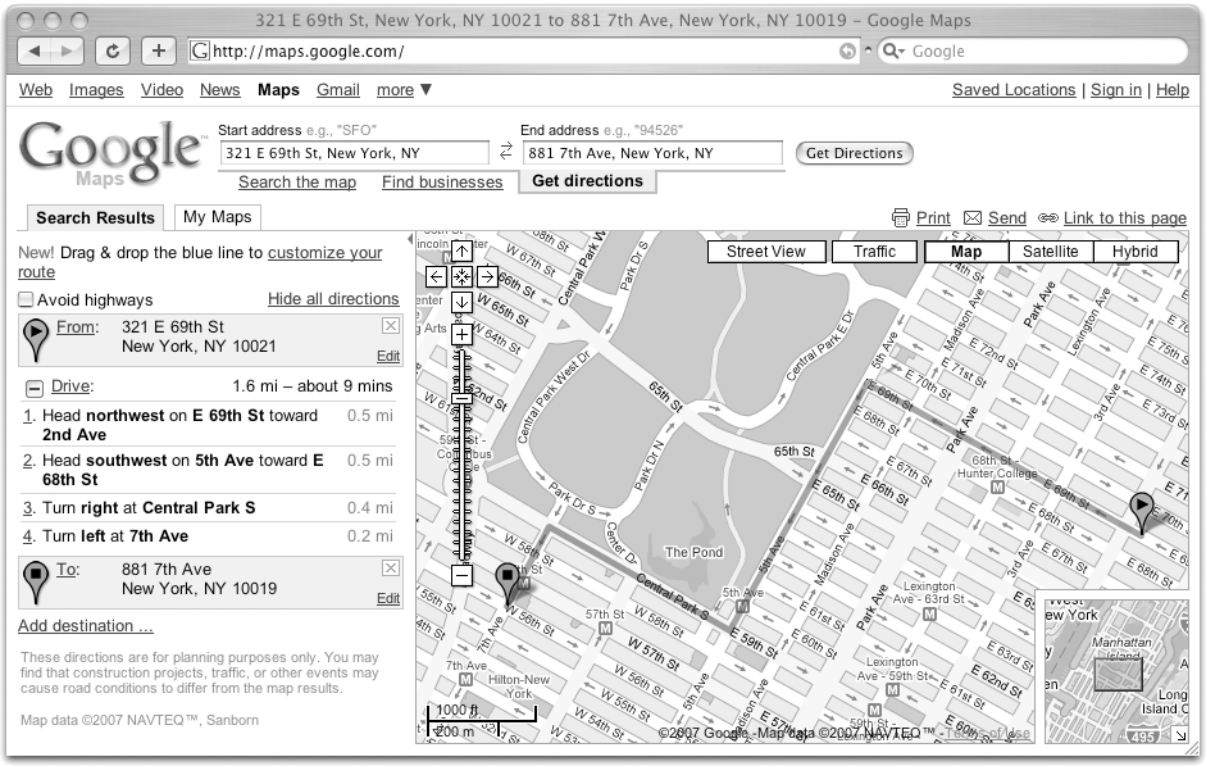

Figure 10.1: Driving directions provided by maps.google.com

```
// Describe a single step in directions to drive from one location to another
public class Step {
    // The distance to drive during this step
    private double length;

    // A brief description of the road used during this step
    private String roadDescription;

    // The angle of the turn made at the start of the step.
    // Right turns use positive angles, left turns use negative angles
    private int turn;

    // Create a description of a new step
    public Step( int direction, double distance, String road ) {
        length = distance;
        roadDescription = road;
        turn = direction;
    }

    // Return text that summarize this step
    public String toString() {
        return sayDirection() + roadDescription + " for " + length + " miles";
    }

    // Return the length of the step
    public double length() {    return length;    }

    // Return the angle of the turn at the beginning of the step
    public int direction() {     return turn;     }

    // Return the name of the road used
    public String routeName() {    return roadDescription;     }

    // Convert turn angle into short textual description
    private String sayDirection() {
        if ( turn == 0 ) {
            return "continue straight on ";
        } else if ( turn < 0 ) {
            return "turn left onto ";
        } else {
            return "turn right onto ";
        }
    }
}
```

Figure 10.2: A class to represent a single step in a journey

`length` will be associated with the total distance traveled. The instance variable `turn` holds the angle of the turn the driver should make at the beginning of the step. If we were only interested in displaying the instructions as text, we might simply save a `String` describing the turn, but the angle provides more information. In particular, it could be used to draw the path to be followed on a map. Finally, `roadDescription` will be associated with a `String` describing the road traveled during a step like "Main St." or "5th Avenue".

The constructor defined with the `Step` class simply takes the three values that describe a step and associates them with the appropriate instance variables. For example, the construction

```
new Step( -90, 0.5, "5th Ave toward E 68th St" )
```

could be used to create a `Step` corresponding to the second instruction in the Google Maps results shown in Figure 10.1.

The `Step` class definition also includes several methods. The `length`, `direction`, and `routeName` methods provide access to the three values used to describe the `Step`. The `toString` method is designed to convert a `Step` into a string that could be displayed as part of a set of driving directions. For example, if invoked on the `Step` created by the construction shown above, this method would return the text

```
turn left onto 5th Ave toward E 68th St for 0.5 miles
```

The definition of `toString` depends heavily on a private method named `sayDirection` which converts the turning angle associated with the instance variable `turn` into an appropriate `String`. This method could easily be refined to say things like "head southwest on" or "make a sharp right onto," but the simple version shown is sufficient for our purposes.

Given the definition of the `Step` class, we could represent the four steps from the instructions shown in Figure 10.1 by declaring the four local variables

```
Step stepOne =   new Step(   0, 0.5, "E 69th St toward 2nd Ave" );
Step stepTwo =   new Step( -90, 0.5, "5th Ave toward E 68th St" );
Step stepThree = new Step(  90, 0.4, "Central Park S" );
Step stepFour =  new Step( -90, 0.2, "7th Ave" );
```

This approach, however, is not very flexible. What if we need to manipulate a different set of instructions that required ten steps? We would need to modify our program by adding six additional variables. Worse yet, this approach does not scale well to handle really large sets of instructions. Would if seem reasonable to define 1000 variables to handle the thousand-step journey described in Lao-Tzu's proverb? Probably not.

Let's think a little bit harder about the proverb

> "A journey of a thousand miles begins with a single step."

By telling us how a journey begins, this proverb also suggests something important about how a journey ends. It might be tempting to parrot Lao-Tzu's famous words by saying

> "A journey of a thousand miles ends with a single step."

but doing so would fail to capture the full nature of a journey. "Begin" and "end" are opposites. What is not a beginning is an ending. So a "deeper" way to rephrase Lao-Tzu's words would be

```
class Journey {

    // The first step
    private Step beginning;

    // The rest of the journey
    private Journey end;

    . . .
```

Figure 10.3: The first step in defining a `Journey` class

> "A journey of a thousand miles ends with a journey of a thousand miles minus a single step."

or more succinctly

> "A long journey ends with a long journey."

The beauty of twisting Lao-Tzu's words in this way is that it leads to a recursive definition of a journey:

> **journey** (noun)
>
> - A single step followed by a journey.

In fact, we can construct a recursive class in Java based our somewhat creative interpretation of Lao-Tzu's saying about journeys.

In other class definitions we have considered, instance variables have been used to keep track of the pieces of information that describe the object the new class is designed to represent. The instance variables in the `Step` class are nice examples of using instance variables in this way. In our `Journey` class, we will similarly define two instance variables to represent the two key parts of the journey, the beginning and the end. The declarations that will be used for these instance variables are shown in Figure 10.3. The first of the instance variables refers to a `Step`, and the other refers to another `Journey`. This is how the `Journey` class becomes recursive. One of its instance variables is of the same type that the class defines.

We will add other instance variables and methods to complete this class definition shortly, but to give some sense of how a recursive class actually encodes a description of a collection, we will first describe an incomplete constructor for our incomplete class and show how it could be used.

The constructor for our `Step` class simply associated values provided as parameters with the instance variables in the class. For each instance variable in the `Step` class, there was a corresponding parameter to the constructor. The constructor for the `Journey` class will work similarly. Most of the code for the constructor is shown in Figure 10.4.

Looking at this code, you should quickly recognize an interesting problem. Since the `Journey` constructor requires a `Journey` as a parameter, you cannot construct a `Journey` unless you already have constructed a `Journey`. Which comes first, the chicken or the egg? Isn't recursion fun?

```
public Journey( Step first, Journey rest ) {
    beginning = first;
    end = rest;
     . . .
}
```

Figure 10.4: A (slightly incomplete) `Journey` constructor

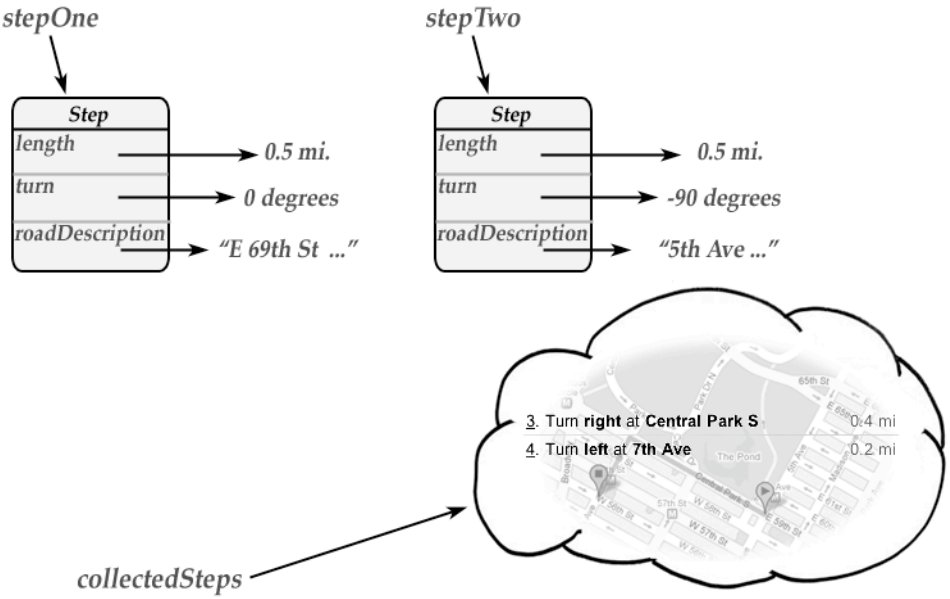

Figure 10.5: Objects used to build a bigger `Journey`

We will explain how to resolve this issue in the next section. For now, let us just assume that somehow we have built a `Journey` composed of the two last steps in the Google Maps directions shown in Figure 10.1 and associated it with a local variable named `collectedSteps` declared as

```
Journey collectedSteps;
```

In addition, assume that we have declared two `Step` variables as

```
Step stepOne;
Step stepTwo;
```

and associated them with `Steps` constructed by executing the assignments

```
stepOne = new Step(   0, 0.5, "E 69th St toward 2nd Ave" );
stepTwo = new Step( -90, 0.5, "5th Ave toward E 68th St" );
```

Figure 10.5 represents the assumptions we are making about the variables `collectedSteps`, `stepOne`, and `stepTwo`. In the figure, each of these variable names is connected by an arrow to a diagram of the object with which it has been associated. The arrow leading to the value associated

with `collectedSteps` points to a cloud-like blob containing an image of the route described by the last two steps of the instructions shown in Figure 10.1. This amorphous shape is used because we cannot yet accurately explain how this object would be constructed. The two `Step` variables, on the other hand, point to tabular diagrams meant to represent the internal structure of the `Step` objects to which they refer. The name of the class `Step` appears at the top of the diagrams representing these objects. Each `Step` has three instance variables: `length`, `turn`, and `roadDescription`. There is a slot for each of these three variables in each of the tabular diagrams representing a `Step`. Just as arrows are used to show the values associated with the local variables, arrows lead from each entry in the `Step` objects to the values associated with the corresponding instance variables. For clarity, we have annotated these values with units like "mi." and "degrees" even though only the actual numeric values would be associated with the variables by the computer.

While we cannot yet explain how to construct a `Journey` from scratch at this point, we can explain how to construct a `Journey` given the objects and variables shown in Figure 10.5. In particular, we could use the (still incomplete) constructor definition shown in Figure 10.4 to create a three step `Journey` by evaluating the construction

```
new Journey( stepTwo, collectedSteps )
```

This construction forms a new `Journey` whose "beginning" is step 2 from our Google Map directions and whose "end" is the third and fourth steps from those instructions. The resulting `Journey` would be a bigger collection of steps, but it would still be a collection of steps. Therefore, we could include this construction in an assignment of the form

```
collectedSteps = new Journey( stepTwo, collectedSteps );
```

Just as we represented `Step`s using tabular diagrams in Figure 10.5, we can use similar diagrams to represent this new `Journey` and its relationship to the objects involved. Such a diagram is shown in Figure 10.6. The arrow showing the object associated with the name `collectedSteps` no longer points to the amorphous blob representing the last two steps. Instead, it shows that this name is associated with a newly constructed `Journey` object. Since we know the structure of this object, it is represented using a tabular diagram similar to the `Step`s. The name `Journey` appears at the top of the table representing the new object. It has one entry for each of the instance variables, `beginning` and `end`. The arrows showing the values associated with these instance variables, however, don't point to simple numbers or `String`s. Instead they point to one of the existing `Step` objects and the blob representing the existing `Journey`.

We can repeat this process to create a longer, four-step `Journey` representing the entire route described in our Google Maps example. To do this, we would execute the assignment

```
collectedSteps = new Journey( stepOne, collectedSteps );
```

The state of the objects and variables in the computer after this assignment is executed is shown in Figure 10.7. The name `collectedSteps` now refers to the most recently constructed `Journey`. This `Journey`'s `end` instance variable refers to the three-step `Journey` created earlier, and the three-step `Journey`'s `end` variable in turn refers to the mysteriously created two-step `Journey`.

Obviously, if we created more `Step` objects, we could also create additional `Journey` objects to represent even longer collections of instructions.

In some sense, none of the `Journey` objects created in this process contain any information. They merely refer to information held in the `Step` objects. We will see, however, that they fill the key role of providing the links needed to access this information. As a result, structures of this form are called *linked lists*.

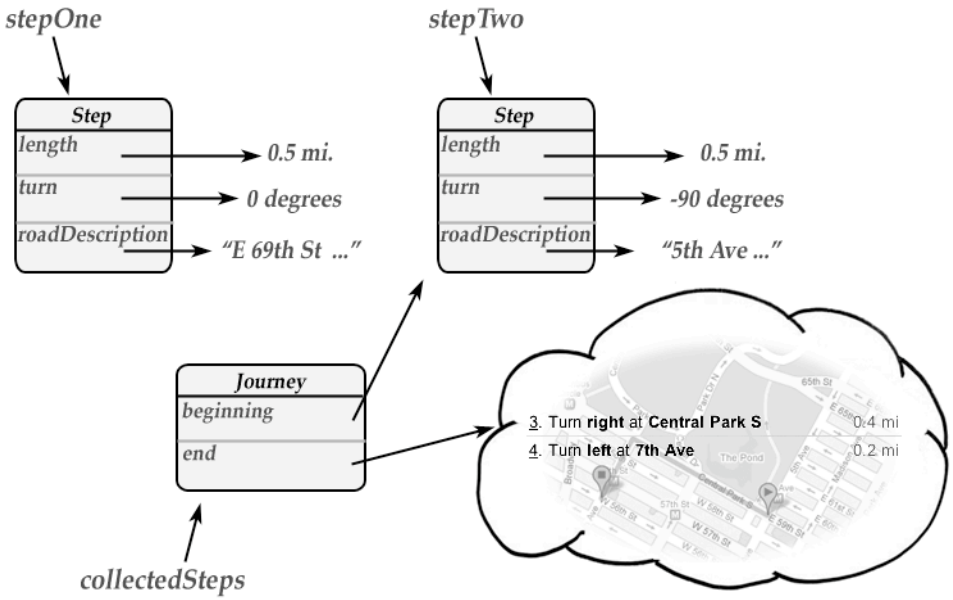

Figure 10.6: Relationships between a `Journey` and its parts

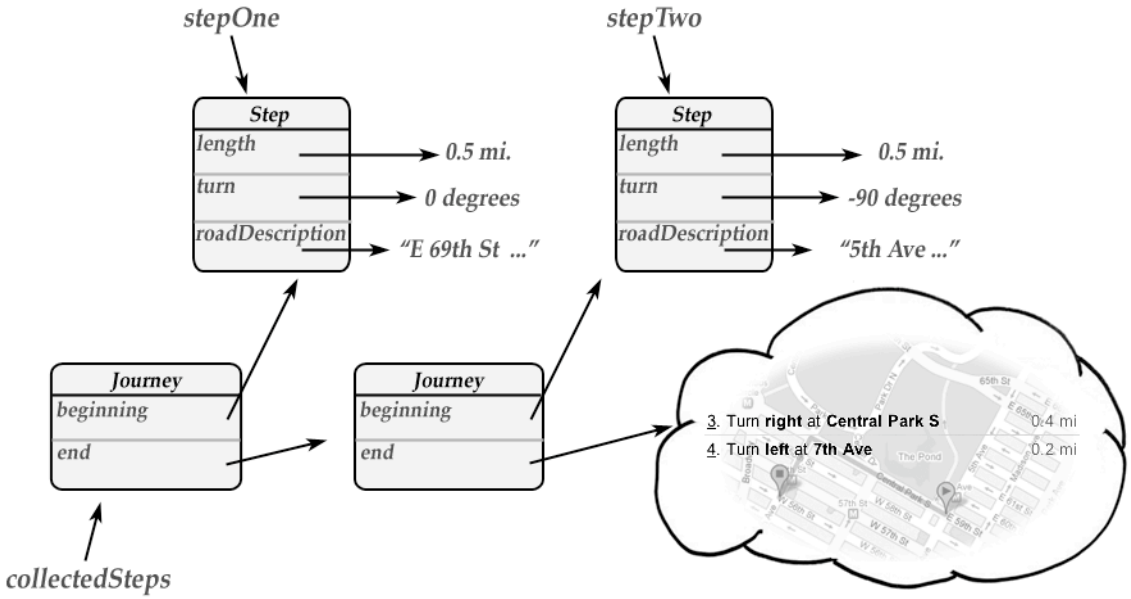

Figure 10.7: Relationships between multiple `Journey`s and their parts

10.2 Journeys End

As presented so far, our recursive definition of a journey is not really workable. If every journey is composed of a first step and another journey, then no journey can ever end! After you take any step, there must be another journey to complete and it must start with a first step followed by another journey, and so on. This flaw in our abstract definition of a journey is also reflected in the fragments of the definition of the `Journey` class we have presented. It is the fundamental reason we cannot yet explain how to create a `Journey` from scratch. Before we can understand how to start the process of creating a `Journey`, we have to refine our recursive definition of a journey so that a journey can end.

In a certain sense, it is obvious when a journey ends. It ends with the last step. That, however, is not the sense we have in mind. Recall that our goal is to learn how to use recursion to manipulate large collections of information. In this context, a journey is a collection of steps. A collection of a million steps is definitely a journey, but is a collection of 10 steps a journey? To know when a journey ends, we need to know when a collection of steps no longer qualifies as a journey.

If by steps we really mean putting one foot in front of another, a series of 500 steps might be considered "going for a walk", but most people would not consider 500 steps a journey. Somewhere around 5000 steps, most of us might be willing to talk about taking a "hike" rather than a walk, but even 5000 steps (roughly 3 miles), isn't what we think of when we talk about a "journey". On the other hand, 100,000 steps is enough of a hike, we might be willing to call traveling that far a journey.

If we can agree on a number like 100,000 as the minimum number of steps that qualify as a journey, there is a fairly simple way to fix our recursive definition. In English, words often have two meanings. If you look up such a word in the dictionary, the definition provided will be broken down into several entries or cases. For example, the definition of the word "case" will include at least five cases:

> **case** (noun)
>
> 1. a container designed to hold or protect something
> 2. a set of circumstances or conditions, i.e., "is the statement true in all three cases"
> 3. an instance of a disease, or problem
> 4. a legal action, esp. one to be decided in a court of law
> 5. any of the inflected forms of a noun, adjective, or pronoun that express the semantic relation of the word to other words in the sentence

If a particular case in a word's definition refers to the word being defined, we say it is a *recursive case*. If all of the cases in a word's definition are recursive, the definition will indeed be circular (and useless). If at least one of the entries is not recursive, however, the circularity can be broken. Such a non-recursive components of a recursive definition is called a *base case*.

To illustrate this, let us give a give a refined, recursive definition of a journey:

> **journey** (noun)
>
> 1. A single step followed by a journey.
> 2. Any collection of 100,000 steps.

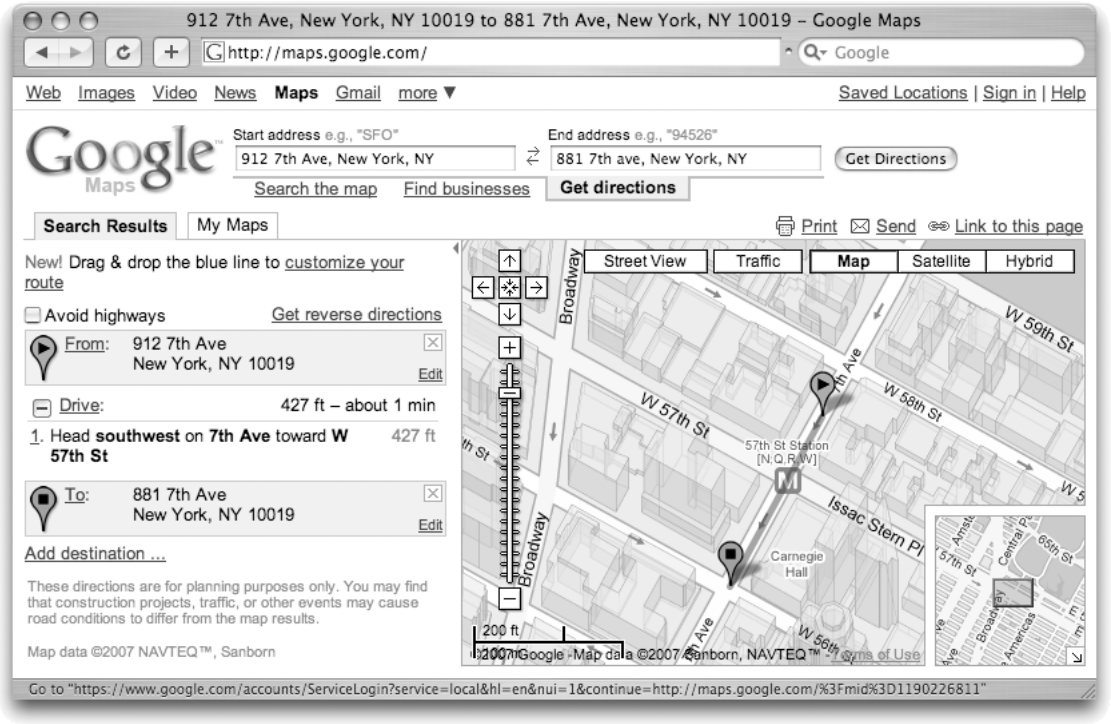

Figure 10.8: Google's answer to a request for simple driving instructions

Given this definition, it is clear that a trip that involves 100,000 steps is a journey because the second case in the definition states so explicitly. It is also true that a trip that involves 100,001 steps is a journey since it is composed of a single step followed by 100,000 steps which we already know is a journey. That is, it fits the first case in the definition. Similarly, now that we know that a trip involving 100,001 steps is a journey, it is clear that a trip involving 100,002 steps is a journey. With similar reasoning, we can see that any trip involving more than 99,999 step fits this definition. On the other hand, a trip that only takes 10 steps is not a journey according to this definition.

While this definition works, it is not clear everyone would agree with the choice of 100,000 as the boundary between journeys and hikes. Similarly, it might not be clear how many steps qualify as a "journey" in the context for which we are designing the `Journey` class, representing computer-generated driving directions. Requiring 100,000 steps is clearly too much. The example "journey" shown in the Google Maps response only consisted of four steps.

We can resolve the question of how few steps Google considers a journey by experimenting with the site. As shown in Figure 10.8, Google definitely recognizes cases where just a single step qualifies as a journey. If you get even sillier, however, and ask Google for directions from an address to the same address, it refuses to accept the idea that this particular journey involves no steps at all. Instead, as shown in Figure 10.9, it insist that you take one step of distance 0.

Who are we to argue with Google?! We will complete our `Journey` class on the assumption that the shortest journey we need to be able to represent is a journey of one step. That is, for our purposes, the abstract definition of a journey will be

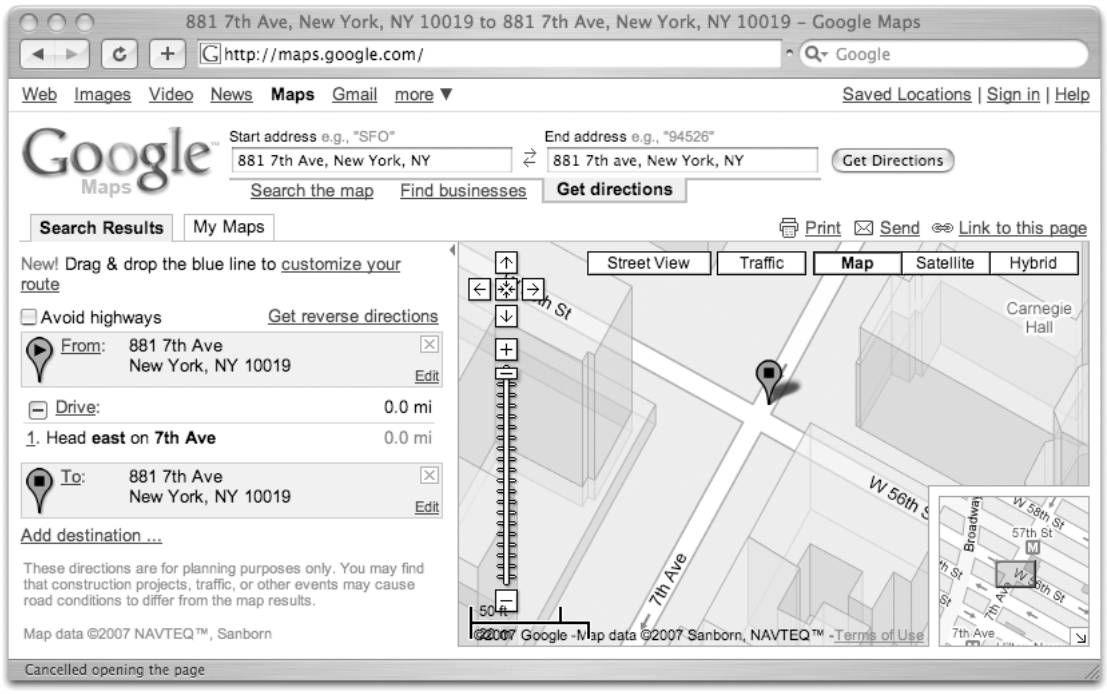

Figure 10.9: How to get to where you already are

journey (noun)

1. A single step followed by a journey.
2. A collection containing just 1 step.

To realize this abstract definition in concrete Java code, we need to define a class with instance variables corresponding to the parts of each of the alternatives included in the definition. The incomplete code in Figure 10.3 already contains the variables `beginning` and `end` needed to describe the step and journey mentioned in alternative 1:

> A single step followed by a journey.

We could add an additional variable to keep track of the single step mentioned in alternative 2:

> A collection containing just 1 step.

An even simpler approach, however, is to use the `beginning` variable to describe this step. After all, if a journey consists of just one step, that step is both its beginning and its end.

We still need to add one new instance variable to enable our `Journey` class to reflect the two part definition of a journey. We need a way to determine which of the alternative definitions describes each `Journey` we create. Since there are only two choices, we can do this using a `boolean` variable. We will add the instance variable declaration

```
// Does this journey contain exactly one step?
private boolean singleStep;
```

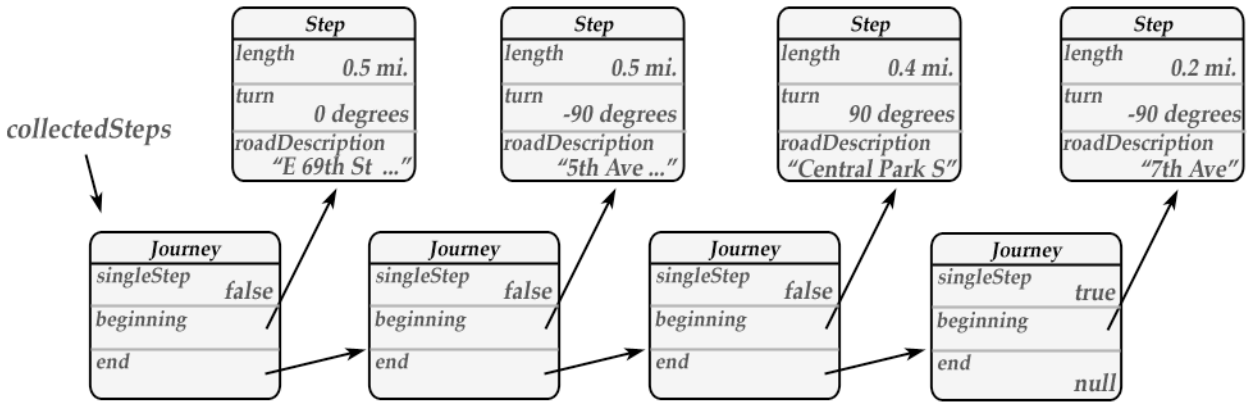

Figure 10.10: A complete `Journey` and its parts

to our class. If the value of this variable is `false`, then we will assume that `Journey` contains more than one `Step`. In such a `Journey` both of the other instance variables must be associated with appropriate objects. If the value of `singleStep` is `true`, then we will assume that `Journey` contains only one `Step`. In that case, the variable `end` will have no useful information associated with it. It will be `null`.

With this addition, we can now do away with the cloud used to represent the last two steps of the `Journey` we described in the preceding section. Figure 10.10 shows a representation of a complete collection of `Journey`s representing the set of driving instructions from Google we first showed in Figure 10.1.

Given the number of objects represented in this figure, we have used a slightly different notation to show the values of an object's instance variables. For instance variable's of types `int`, `double`, `boolean`, and `String`, we have simply written the value of each variable in the same box as its name rather than connecting the names to the values using arrows. In particular, note that the value of the new instance variable `singleStep` is shown in each of the tables representing a `Journey` object. As one would expect, as one follows the chain of `end` arrows through the steps of a journey, the values associated with `singleStep` are all `false` until we reach the final step of the `Journey`. Then, in the last `Journey`, `singleStep` is `true`. This is how `Journey`s end.

10.3 Overload

Now that we have added the `singleStep` instance variable to the `Journey` class, we can complete the process of defining the constructor for the class. If we followed the pattern we used when defining the constructor for the `Step` class, the addition of `singleStep` would lead to the definition of a constructor that expected three parameter values, one for each instance variable. This constructor would simply associate the instance variables with the parameter values provided. We could proceed in this way, but Java supports a better approach.

When defining a class, we can include several constructor definitions as long as each constructor we define expects a different number of parameters or parameters of different types than any of the other constructors. In this case, the constructor is said to be *overloaded*. When asked to evaluate a construction for an object of a type containing multiple constructor definitions, Java uses the types

of the actual parameter values provided to determine which of the constructor definitions should be used.

We will use this feature to provide two constructors for the `Journey` class, one will be for `Journey`s with just one step, the base case, and the other constructor will be for multi-step `Journey`s, the recursive case.

The beginning of the text of the `Journey` class including all of the instance variables and the definitions of these two constructors is shown in Figure 10.11. The first constructor definition is very similar to the incomplete definition shown in Figure 10.4. The only change we have made is to add the assignment

```
singleStep = false;
```

This constructor will be used to construct `Journey`s that represent driving directions that contain multiple steps. We set `singleStep` equal to `false` to reflect this.

The second constructor will be used only if we evaluate a construction in which we provide only a single `Step` parameter like

```
new Journey( aStep )
```

This version of the constructor associates the instance variable `beginning` with its parameter value and sets `singleStep` equal to `true` to reflect the fact that there are no other steps in this journey. It has no value to associate with `end`. Just to be safe, it explicitly associates `end` with `null`, the name that represents "no value" in Java.

To better illustrate the roles of these constructors, consider how we could construct the complete collection of objects shown in Figure 10.10. First, we could create the four `Step`s involved and associate them with local variable using the initialized declarations

```
Step stepOne =   new Step(   0, 0.5, "E 69th St toward 2nd Ave" );
Step stepTwo =   new Step( -90, 0.5, "5th Ave toward E 68th St" );
Step stepThree = new Step(  90, 0.4, "Central Park S" );
Step stepFour =  new Step( -90, 0.2, "7th Ave" );
```

Then, we would create a `Journey` representing just the last step using the initialized declaration

```
Journey collectedSteps = new Journey( stepFour );
```

Because only one parameter is included in the construction used in this statement, Java will use the second constructor definition shown in Figure 10.11 to process this construction.

Finally, we could execute the sequence of assignments

```
collectedSteps = new Journey( stepThree, collectedSteps );
collectedSteps = new Journey( stepTwo, collectedSteps );
collectedSteps = new Journey( stepOne, collectedSteps );
```

to add each of the other steps to the structure associated with the variable `collectedSteps`. Each of these assignments would be processed using the first constructor definition.

While these instructions will create the structure shown in Figure 10.11, they do not really show how recursion makes it easier to manipulate large collections of information. We still need one variable for each step in the instructions for the `Journey` we want to represent. Therefore,

```
class Journey {

    // The first step
    private Step beginning;

    // The rest of the journey
    private Journey end;

    // Does this journey contain exactly one step?
    private boolean singleStep;

    // Construct a multi-step journey given a step to add to an existing journey
    public Journey( Step firstStep, Journey remainder ) {
        beginning = firstStep;
        end = remainder;
        singleStep = false;
    }

    // Construct a single step journey
    public Journey( Step onlyStep ) {
        beginning = onlyStep;
        end = null;
        singleStep = true;
    }


      . . .  more to come  . . .


}
```

Figure 10.11: Overloaded constructors for the `Journey` class

you may have already guessed that this code is not typical of the instructions used to construct a collection of linked `Journey` objects in a real program.

If our `Journey` class was part of a program that generated driving directions for a site like Google Maps, it would be used in conjunction with an algorithm that could derive a good route to travel from one location to another. Such algorithms are called *routing algorithms*. The presentation of a routing algorithm is far beyond the scope of this chapter. We will not attempt to explain such algorithms here. However, to give you a realistic sense of how a `Journey` would be constructed in a real program, we will describe a possible interface to a routing algorithm and then show code to construct a `Journey` using this interface.

We will assume the existence of two additional classes:

`RoadMap` — This class will represent the information in a map.

`Location` — This class will represent a location within the area described by the map.

We will not discuss the details of the implementation of these classes, but we will assume that the code for the `RoadMap` class includes the implementation of a routing algorithm. We will also assume that the class provides one method through which we can request information from the routing algorithm.

Given two `Location`s and a `RoadMap`, we need a method that will use the routing algorithm to determine the path from one location to the other. We do not want a method that will return the entire path as a `Journey`. We want a method that will return one `Step` of the path at a time. With this in mind, we will assume that if `aMap` is a `Map` object, and `startingLoc` and `endingLoc` are `Location`s, then an invocation of the form

```
aMap.getLastStepOfRoute( startingLoc, endingLoc )
```

will return a `Step` object describing the last step someone should follow to drive from `startingLoc` to `endingLoc`. This may seem a bit counterintuitive. You might have expected a method that would return the first step or the Nth step to a destination. The `getLastStepOfRoute` method, however, provides just what we will need.

We will also assume the existence of two methods that connect the `Location` class with our `Step` class. Our `Step` class provides directions that would take a person from one location on the map to another. Therefore, we will assume that if `aStep` is an object of the `Step` class, then the method invocations

```
aStep.getStart()
aStep.getEnd()
```

will return the `Location` objects describing the locations at either end of a `Step`.

Given such class and method definitions, the code we might use to construct a `Journey` representing the entire route to follow from a given `startingLoc` and `endingLoc` is shown in Figure 10.12. It first creates a single step `Journey` containing just the last step of the route. Then it executes a loop that repeatedly creates longer `Journey`s by adding earlier steps. The most recently added step is always associated with the name `currentStep`. Therefore, the loop terminates when the starting position of `currentStep` equals the starting location for the complete route.

Note that this code can handle a journey of as few or as many steps as necessary. Regardless of the number of `Step`s included in the `Journey` created by the loop, only the single `Step` variable `currentStep` is required within the code for the loop. When the loop is complete, each `Step` returned by the routing algorithm is associated with the name `beginning` in one of the list of `Journey` objects created.

```
//  Create a Journey containing just the last step
Step currentStep = aMap.getLastStepOfRoute( startingLoc, endingLoc );
Journey completeRoute = new Journey( currentStep );

// The variable intermediateLoc will always refer to the
// current starting point of completeRoute
Location intermediateLoc = currentStep.getStart();

//  Repeatedly add earlier steps until reaching the starting position
while ( ! startingLoc.equals( intermediateLoc ) ) {

    currentStep = aMap.getLastStepOfRoute( startingLoc, intermediateLoc );
    completeRoute = new Journey( currentStep, completeRoute );
    intermediateLoc = currentStep.getStart();
}
```

Figure 10.12: Using the `Journey` class in a realistic algorithm

10.4 Recurring Methodically

Our `Journey` class now has all the instance variables and constructors it needs, but it does not have any methods. Without methods, all we can do with `Journey`s is construct them and draw lovely diagrams to depict them like Figure 10.10. To make the class more useful, we need to add a few method definitions.

Our `Step` class included a `toString` method. It would be helpful to have a similar method for the `Journey` class. The `toString` method of the `Journey` class would create a multi-line `String` by concatenating together the `String`s produced by applying `toString` to each of the `Step`s in the `Journey`, placing a newline after the text that describes each step. Such a method would make it easy to display the instructions represented by a `Journey` in a `JTextArea` or in some other human-readable form.

Since the value returned by applying the `toString` method to a `Journey` will usually include multiple lines, you might expect the body of the method to contain a loop. In fact, no loop will be required. Instead, the repetitive behavior the method exhibits will result from the fact that the method, like the class in which it is defined, will be recursive. Within the body of the `toString` method, we will invoke `toString` on another `Journey` object.

10.4.1 Case by Case

The defintion of a recursive method frequently includes an `if` statement that reflects the different cases used in the definition of the recursive class in which the method is defined. For each case in the class definition, there will be a branch in this `if` statement. The branches corresponding to recursive cases will invoke the method recursively. The branches corresponding to non-recursive cases will return without making any recursive invocations. The definition of our `Journey` class had two cases: the recursive case for journeys containing several steps and the base case for journeys

```
public String toString() {
    if ( singleStep ) {
         ... Statements to handle single-Step Journeys (the base case) ...
    } else {
         ... Statements to handle multi-Step Journeys ...
    }
}
```

Figure 10.13: Basic structure for a `Journey` `toString` method

containing just one step. As a result, the body of our `toString` method will have the structure shown in Figure 10.13

The code to handle a single step `Journey` is quite simple. The method should just return the text produced by applying `toString` to that single `Step` and appending a newline to the result. That is, the code in the first branch of the `if` statement will be

```
return beginning.toString() + "\n";
```

The code to handle multiple step `Journey`s is also quite concise and quite simple (once you get used to how recursion works). The `String` returned to describe an entire `Journey` must start with a line describing its first step. This line is produced by the same expression used in the base case:

```
beginning.toString() + "\n"
```

The line describing the first step should be followed by a sequence of lines describing all of the other steps. All of these other steps are represented by the `Journey` associated with the instance variable `end`. We can therefore obtain the rest of the `String` we need by invoking `toString` recursively on `end`.

One of the tricky things about describing a recursive method is making it very clear exactly which object of the recursive type is being used at each step. We are writing a method that will be applied to a `Journey` object. Within that method, we will work with another `Journey` object. This second `Journey` has a name, `end`. The original object really does not have a name. We will need a way to talk about it in the following paragraphs. We will do this by referring to it as the "original `Journey`" or the "original object."

When the `toString` method is applied to `end`, it should return a sequence of lines describing all of the steps in the `Journey` named `end`. That is, it should return a `String` describing all but the first step of the original `Journey`. Therefore, the expression

```
end.toString()
```

describes the `String` that should follow the line describing the first step of the original `Journey`. Putting this together with the line describing the first step will give us a complete description of the original `Journey`. As a result, we can complete the code in Figure 10.13 by placing the instruction

```
return beginning.toString() + "\n" + end.toString();
```

in the second branch of the `if` statement.[2] The complete code for `toString` as it would appear in the context of the definition of the `Journey` class is shown in Figure 10.14.

[2]In fact, if we want to be even more concise, we can use the statement

```
class Journey {

    // The first step
    private Step beginning;

    // The rest of the journey
    private Journey end;

    // Does this journey contain exactly one step?
    private boolean singleStep;

      . . .

    // Constructor code has been omitted here to save space.
    // The missing code can be found in Figure 10.11

      . . .

    // Return a string describing the journey
    public String toString() {
        if ( singleStep ) {
            return beginning.toString() + "\n";
        } else {
            return beginning.toString() + "\n" + end.toString();
        }
    }

          . . .  more to come  . . .

}
```

Figure 10.14: The recursive definition of the `Journey toString` method

Note that using the expression `beginning.toString()` in our `toString` method does not make the method recursive. At first, it might seem like it does. We are using a method named `toString` within the definition of `toString`. When we use a method name, however, Java determines how to interpret the method name by look at the type of the object to which the method is being applied. In this case, it looks at the type of the name `beginning` that appears before the method name. Since `beginning` is a `Step`, Java realizes that the `toString` method we are using is the `toString` method of the `Step` class. The method we are defining is the `toString` method of the `Journey` class. These are two different methods. Therefore, this invocation alone does not make the method recursive. It is the invocation of `end.toString()` that makes the definition recursive. Since `end` refers to another `Journey`, Java interprets this use of the name `toString` as a reference to the method being defined.

10.4.2 Understanding Recursive Methods

When trying to understand a recursive method, whether you are writing it yourself or trying to figure out how someone else's method works, there are several key steps you should take:

1. identify the cases involved, distinguishing base cases from recursive cases,
2. ensure that the definition is recursive but not circular by verifying that all recursive invocations involve "simpler cases", and
3. verify the correctness of the code for each case while assuming that all recursive invocations will work correctly.

As we have explained in the description of the `toString` method, the cases that will be included in a recursive method often parallel the cases included in the recursive class with which the method is associated. We will, however, see that additional cases are sometimes necessary.

There is a danger when we write a recursive method that one recursive invocation will lead to another in a cycle that will never terminate. The result would be similar to writing a loop that never stopped executing.

To ensure that a recursive method eventually stops, the programmer should make sure that the objects involved in all recursive invocations are somehow simpler than the original object. In the case of our recursive `toString` method, the `Journey` associated with the name `end` is simpler than the original `Journey` in that it is shorter. It contains one less step. In general, if we invoke a method recursively, the object used in the recursive invocation must be "simpler" by somehow being closer to one of the base cases of the recursive method. This is how we ensure the method will eventually stop. Every recursive invocation gets closer to a base case. Therefore, we know that our repeated recursive invocations will eventually lead to base cases. The base cases will stop because they do not make any recursive invocations.

Even if one believes a recursive method will stop, it may still not be obvious that it will work as desired. The correct way to write a recursive method is to assume the method will work on any object that is "simpler" than the original object. Then, for each case in the definition of the

---

```
return beginning + "\n" + end;
```

because Java automatically applies the `toString` method to any object that is not a `String` when that object is used as an argument to the concatenation operator ("`+`"). For now, however, we will leave the `toString`s in our statement to make the recursion in the definition explicit.

recursive class, figure out how to calculate the correct result to return using the results of recursive invocations on simpler objects as needed. As a result, the correct way to convince yourself that a recursive method is correct if by checking the code written to handle each of the cases under the assumption that all recursive invocations will work correctly.

How can we assume our method definition will work if we have not even finished writing it? To many people, an argument that a method will work that is based on the assumption that it will work (on simpler objects) seems vacuous. Surprisingly, even if we make this strong assumption, it will still be not be possible to conclude that an incorrect method is incorrect.

As a simple example of this, suppose we replaced the instruction

```
return beginning.toString() + "\n" + end.toString();
```

in our recursive `toString` method with

```
return beginning.toString() + end.toString();
```

While similar to the original definition, this method would no longer work as expected. It would concatenate together all the lines of instructions as desired, but it would not place any new line characters between the steps so that they would all appear together as one long line of text.

Suppose, however, that we did not notice this mistake and tried to verify the correctness of the alternate version of the code by assuming that the invocation `end.toString()` would work correctly. That is, suppose that we assumed that the recursive invocation would return a sequence of separate lines describing the steps in a `Journey`. Even is we make this incorrect assumption about the recursive invocations we will still realize that the new method will not work correctly if we examine its code carefully. Looking at the code in the recursive branch of the `if` statement, it is clear the first newline will be missing. The assumption that the recursive calls will work is not sufficient to hide the flaw in the method. This will always be the case. If you can correctly argue that a recursive works by assuming that all the recursive calls it makes work correctly, then the method must indeed work as expected.

10.4.3 Blow by Blow

At first, most programmers find it necessary to work through the sequence of steps involved in the complete processing of a recursive invocation before they can really grasp how such methods work. With this in mind, we will carefully step through the process that would occur while a computer was evaluating the invocation `collectedSteps.toString()` which applies our `toString` method to the structure discussed as an example in Section 10.3.

Warning! We do not recommend doing this every time you write or need to understand a recursive method. Tracing through the steps of the execution of a recursive method can be quite tedious. Once you get comfortable with how recursion works, it is best to understand a recursive method by thinking about its base cases and recursive cases as explained in the previous section. In particular, if you become confident you understand how the recursive `toString` method works before completing this section, feel free to skip ahead to the next section.

As we examine the process of applying `toString` recursively, it will be important to have a way to clearly identify each of the objects involved. With this in mind, we will assume that the `Journey` to which `toString` is applied is created slightly differently than we did in Section 10.3. We will still assume that the process begins with the creation of the four `Step` objects

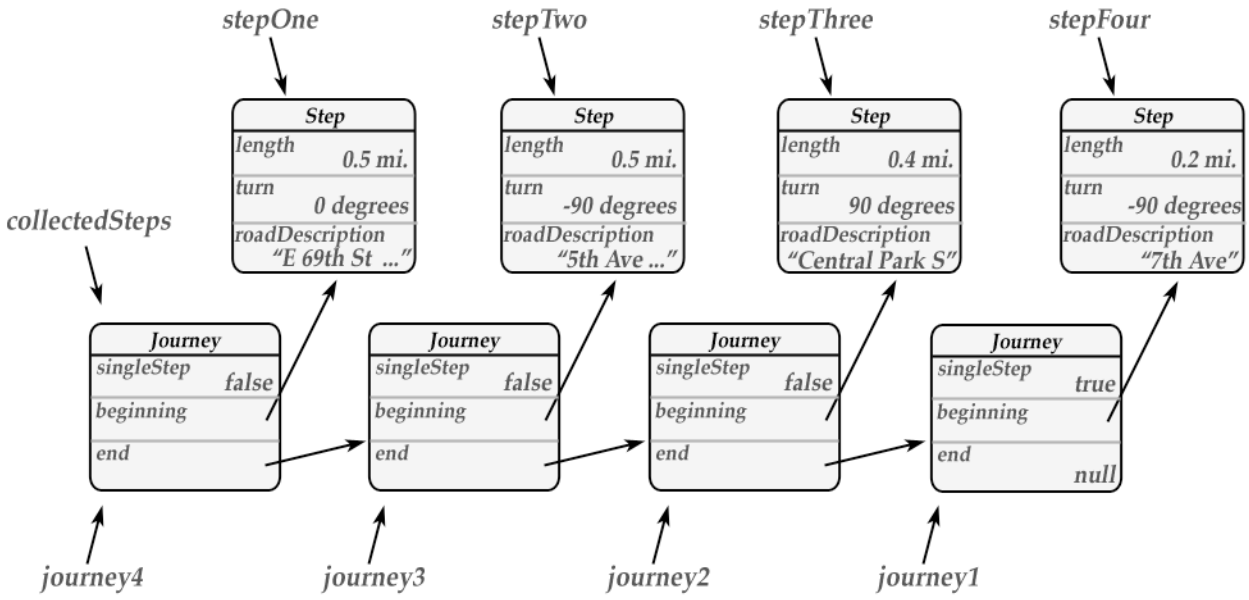

Figure 10.15: A `Journey` with names for all of its parts

```
Step stepOne =   new Step(   0, 0.5, "E 69th St toward 2nd Ave" );
Step stepTwo =   new Step( -90, 0.5, "5th Ave toward E 68th St" );
Step stepThree = new Step(  90, 0.4, "Central Park S" );
Step stepFour =  new Step( -90, 0.2, "7th Ave" );
```

Now, however, we will assume that the `Journey` objects are created using the code

```
Journey journey1 = new Journey( stepFour );
Journey journey2 = new Journey( stepThree, journey1 );
Journey journey3 = new Journey( stepTwo, journey2 );
Journey journey4 = new Journey( stepOne, journey3 );
Journey collectedSteps = journey4;
```

This code creates exactly the same structure as the code in Section 10.3, but it associates a distinct name with each part of the structure. The structure and the names associated with each component are shown in Figure 10.15. We will use the names `journey1`, `journey2`, `journey3`, and `journey4` to unambiguously identify the objects being manipulated at each step in the execution of the recursive method.

Having distinct names for each of the objects involved will be helpful because as we trace through the execution of this recursive method invocation we will see that certain names refer to different objects in different contexts. For example, the condition in the `if` statement that forms the body of the `toString` method of the `Journey` class checks whether the value associated with the name `singleStep` is `true` or `false`. Looking at Figure 10.15 we can see that `singleStep` is associated with values in all four of the `Journey` objects that will be involved in our example. In three of the objects, it is associated with `false` and in one it is associated with `true`. In order to know which branch of this `if` statement will be executed, we have to know which of the four values associated with `singleStep` should be used.

When `singleStep` or any other instance variable is referenced within a method, the computer uses the value associated with the variable within the object identified in the invocation of the method. In the invocation

```
collectedSteps.toString()
```

`toString` is being applied to the object associated with the names `collectedSteps` and `journey4`. Therefore, while executing the steps of the method, the values associated with instance variable are determined by the values in `journey4`. Within this object, `singleStep` is `false`. On the other hand, if we were considering the invocation `journey1.toString()`, then the values associated with instance variables would be determined by the values in the object named `journey1`. In this situation, the value associated with `singleStep` is `true`.

One final issue that complicates the description of the execution of a recursive method is that fact that when a recursive invocation is encountered, the computer begins to execute the statements in the method again, even though it hasn't finished its first (or nth) attempt to execute the statements in the method. When a recursive invocation is encountered, the ongoing execution of the recursive method is suspended. It cannot complete until all the steps of the recursive invocation are finished and the result of the recursive invocation are available. As we step through the complete execution process, it is important to remember which executions of the method are suspended pending the results of recursive invocations. We will use a simple formatting trick to help your memory. The entire description of any recursive invocation will be indented relative to the text describing the execution that is awaiting its completion and result. To make this use of indentation as clear as possible, we will start the description of the execution process on a fresh page.

The first thing a computer must do to evaluate

```
collectedSteps.toString()
```

is determine which branch of the `if` statement in the `toString` method to execute. It does this by determining the value of `singleStep` within the object named `collectedSteps`. Since `singleStep` is `false` in this object, the computer will execute the second branch of the `if` statement:

```
return beginning.toString() + "\n" + end.toString();
```

To do this, the computer must first evaluate the expression

```
beginning.toString() + "\n"
```

by appending a newline to whatever is produced by applying `toString` to `beginning`. Looking at Figure 10.15, we can see that within `collectedSteps`, `beginning` is associated with `stepOne`. Applying `toString` to `beginning` will therefore produce

```
continue straight on E 69th St toward 2nd Ave for 0.5 miles
```

Next the computer must evaluate `end.toString()`. Within `collectedSteps`, the name `end` is associated with `journey3`. Therefore, this invocation is equivalent to `journey3.toString()`. This is a recursive invocation, so we will indent the description of its execution.

> The computer begins executing `journey3.toString()` by examining the value of `singleStep` within `journey3`. In this context, `singleStep` is `false`, so the computer will again execute the second, recursive branch of the `if` statement. Within `journey3`, the name `beginning` refers to the object `stepTwo`, so the application of `toString` to `beginning` will return
>
> ```
> turn left onto 5th Ave toward E 68th St for 0.5 miles
> ```
>
> The computer will next evaluate the invocation `end.toString()`. Within `journey3`, the name `end` refers to `journey2`. This invocation is therefore equivalent to `journey2.toString()`. It is recursive, so its description deserves more indentation.
>
> > Within `journey2`, `singleStep` has the value `false`. Therefore, the computer will again choose to execute the recursive branch of the `if` statement. Within `journey2`, `beginning` is associated with `stepThree` and therefore applying `toString` will produce the text
> >
> > ```
> > turn right onto Central Park S for 0.4 miles
> > ```
> >
> > Next, the computer applies `toString` to `end` (which refers to `journey1` in this context). This is a recursive invocation requiring even more indentation.
> >
> > > Within `journey1`, `singleStep` is `true`. Instead of executing the second branch of the `if` statement again, the computer finally get to execute the first branch
> > >
> > > ```
> > > return beginning.toString() + "\n";
> > > ```
> > >
> > > This does not require any recursive calls. The computer simply applies `toString` to `stepFour`, the object associated with the name `beginning` within `journey1`. This returns

```
turn left onto 7th Ave for 0.2 miles
```

The computer sticks a newline on the end of this text and returns it as the result of the recursive invocation of `toString`. This brings the computer back to...

... its third attempt to execute the recursive branch of the `if` statement:

```
return beginning.toString() + "\n" + end.toString();
```

This instruction was being executed to determine the value that should be produce when `toString` was applied to `journey2`. It had already determined the value produced by `beginning.toString()`. Now that the value of the recursive invocation is available it can concatenate the two `String`s together and return

```
turn right onto Central Park S for 0.4 miles
turn left onto 7th Ave for 0.2 miles
```

as its result. This result gets returned to the point where ...

... the computer was making its second attempt to execute recursive branch of the `if` statement. This was within the invocation of `toString` on the object `journey3`. The invocation of `toString` to `beginning` in this context had returned

```
turn left onto 5th Ave toward E 68th St for 0.5 miles
```

By concatenating together this line, a newline, and the two lines returned by the recursive invocation of `toString`, the computer realizes that this invocation of `toString` should return

```
turn left onto 5th Ave toward E 68th St for 0.5 miles
turn right onto Central Park S for 0.4 miles
turn left onto 7th Ave for 0.2 miles
```

to the point where ...

... the computer was making its first attempt to execute the recursive branch of the `if` statement. This was within the original application of `toString` to `journey4` through the name `collectedSteps`. Here, the application of `toString` to `beginning` had produced

```
continue straight on E 69th St toward 2nd Ave for 0.5 miles
```

Therefore, the computer will put this line together with the three lines produced by the recursive call and produce

```
continue straight on E 69th St toward 2nd Ave for 0.5 miles
turn left onto 5th Ave toward E 68th St for 0.5 miles
turn right onto Central Park S for 0.4 miles
turn left onto 7th Ave for 0.2 miles
```

as the final result.

10.4.4 Summing Up

One must see several example of a new programming technique in order to recognize important patterns. Therefore, before moving onto another topic, we would like to present the definition of another recursive method similar to the `toString` method.

Given a `Journey`, one piece of information that can be important is the total length of the trip. This information is certainly displayed by sites that provide driving directions like `maps.google.com` and `www.mapquest.com`. We would like to add the definition of a `length` method for our `Journey` class that returns the total length of a journey in miles.

Again, the structure of the method will reflect the two categories of `Journey`s we construct — multi-step `Journey`s and single-step `Journey`s. We will write an `if` statement with one branch for each of these cases.

The `Step` class defined in Figure 10.2 includes a `length` method that returns the length of a single `Step`. This will make the code for the base case in the `length` method for the `Journey` class very simple. It will just return the length of the `Journey`'s single step.

The code for the recursive case in the `length` method will be based on the fact that the total length of a journey is the length of the first step plus the length of all of the other steps. We will use a recursive invocation to determine the length of the `end` of a `Journey` and then just add this to the length of the first step.

Code for a `length` method based on these observations is shown in Figure 10.16.

```
public double length() {
    if ( singleStep ) {
        return beginning.length();
    } else {
        return beginning.length() + end.length();
    }
}
```

Figure 10.16: Definition of a `length` method for the `Journey` class

10.5 Lovely spam! Wonderful spam!

We are now ready to move on to a new example that will allow us to explore additional aspects of recursive definitions. In this example, we will again define a class to manage a list, but instead of being a list of `Step`s, it will be a list of `String`s. The features of this class will be motivated by an annoyance we can all relate to, the proliferation of unwanted email messages known as "spam".

Much to the annoyance of the Hormel Foods Corporation[3], the term spam is now used to describe unwanted emails offering things like stock tips you cannot trust, herbal remedies guaranteed to enlarge body parts you may or may not have, prescription drugs you don't have a prescription for, and approvals for loan applications you never submitted. Many email programs now contain

[3]Before people starting calling unwanted email spam, the name was (and actually still is) associated with a canned meat product produced by Hormel. If you have never had the pleasure of eating Spam, you should visit http://www.spam.com or at least read through the text of Monty Python's skit about the joys of eating Spam (http://www.detritus.org/spam/skit.html).

Figure 10.17: Interface for a simple mail client

features to identify messages that are spam. These features make it possible to either automatically delete spam messages or to at least hide them from the user temporarily. If your email client is doing a very good job of recognizing spam, you might not even be aware of the vast amount of electronic junk mail that is sent your way every day. If so, look for a way to display the "unwanted mail" or "trash" folder on your email client. You might be surprised. We will consider aspects of how such a spam control mechanism could be incorporated in an email client.

Commercial email clients depend on sophisticated algorithms to automatically identify messages as spam. We will take a much simpler approach. Our program will allow its user to enter a list of words or phrases like "enlargement", "online pharmacy", and "loan application" that are likely to appear in spam messages. The program will then hide all messages containing phrases in this list from the user.

Samples of the interface we have in mind are shown in Figures 10.17 through 10.20. The program provides fields where the user can enter account information and buttons that can be used to log in or out of the email server. Once the user logs into an account, the program will display summaries of the available messages in a pop-up menu as shown in Figure 10.18. Initially, this menu will display all messages available, probably including lots of unwanted messages as shown in the figure.[4]

Below the area in which messages are displayed, there are components that allow the user to control the program's ability to filter spam. The user can enter a phrase that should be used to

[4]Alas, I did not have to "fake" the menu shown to make the spam look worse than it really is. The messages shown are the messages I actually found on my account the morning I created these figures. In fact, the only "faking" that occurred was to delete a few of the more objectionable messages before capturing the window snapshots.

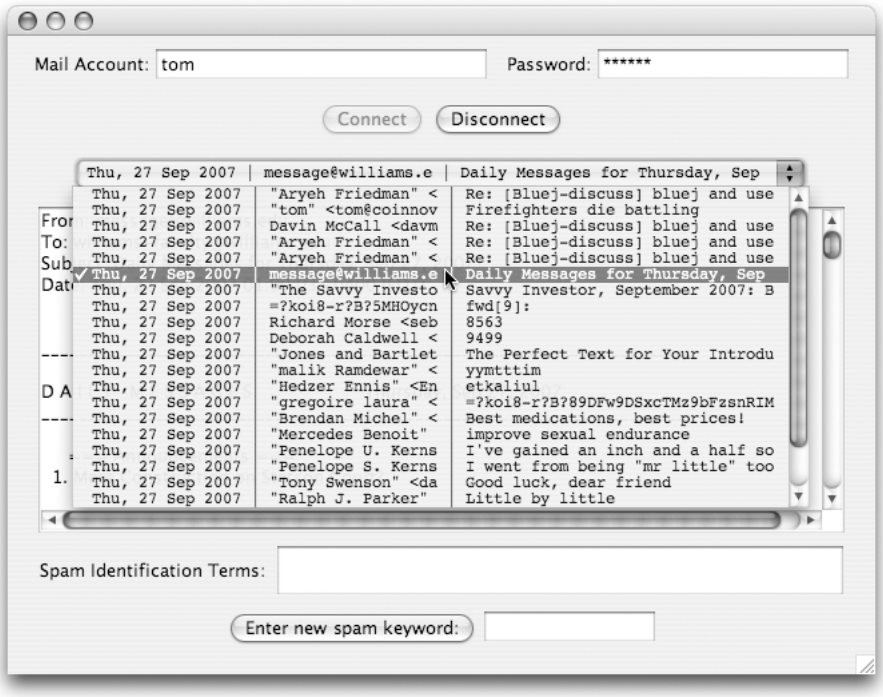

Figure 10.18: Looking for a message amidst the spam

identify spam messages and then press the "Enter new spam keyword" button. The program will then remove all messages containing the phrase from the menu of messages it displays as shown in Figure 10.19.

The user must enter spam identification terms one at a time, but the user can enter as many terms as desired by simply repeating this process. The program will display a list of all of the terms that have been entered and remove messages containing any of these terms from its menu as shown in Figure 10.20

We will not attempt to present code for this entire program here. Our goal will be to explore the design of one class that could be used in such a program, a recursive class named `BlackList` that could hold the collection of spam identification terms entered by the user. This class should provide a method named `looksLikeSpam`. The `looksLikeSpam` method will take the text of an email message as a parameter and return `true` if that text contains any of the phrases in the `BlackList`. The program will use this method to decide which messages to include in the menu used to select a message to display.

10.6 Nothing Really Matters

The first interesting aspect of the definition of the `BlackList` class is its base case. For the `Journey` class, the base case was a list containing just a single step. If we took a similar approach here, the base case for the `BlackList` class would be a list containing just a single `String`. Such a list could be used to represent the list of spam identification terms shown in Figure 10.19.

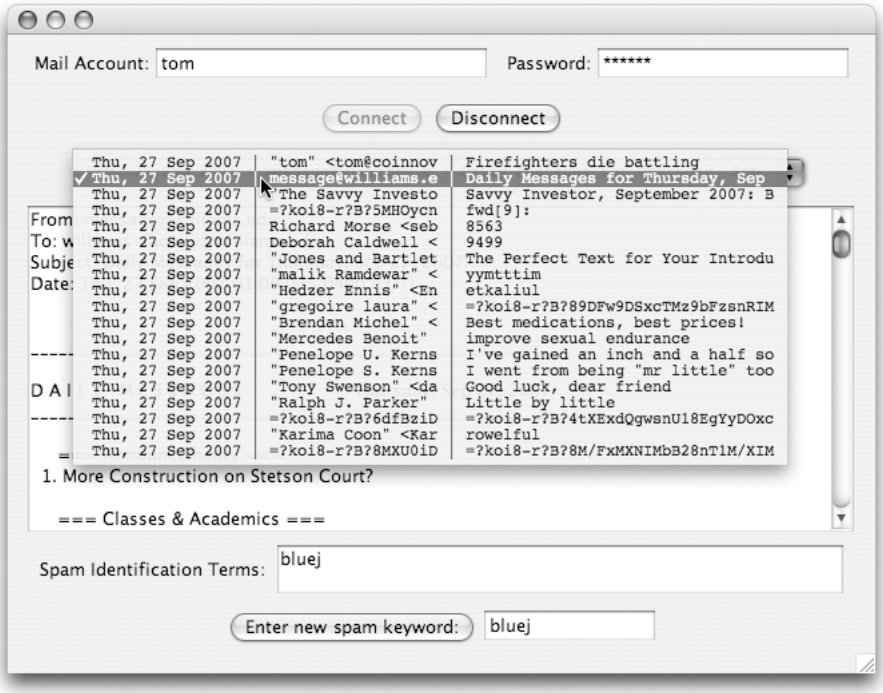

Figure 10.19: Message list filtered using a single spam identification term

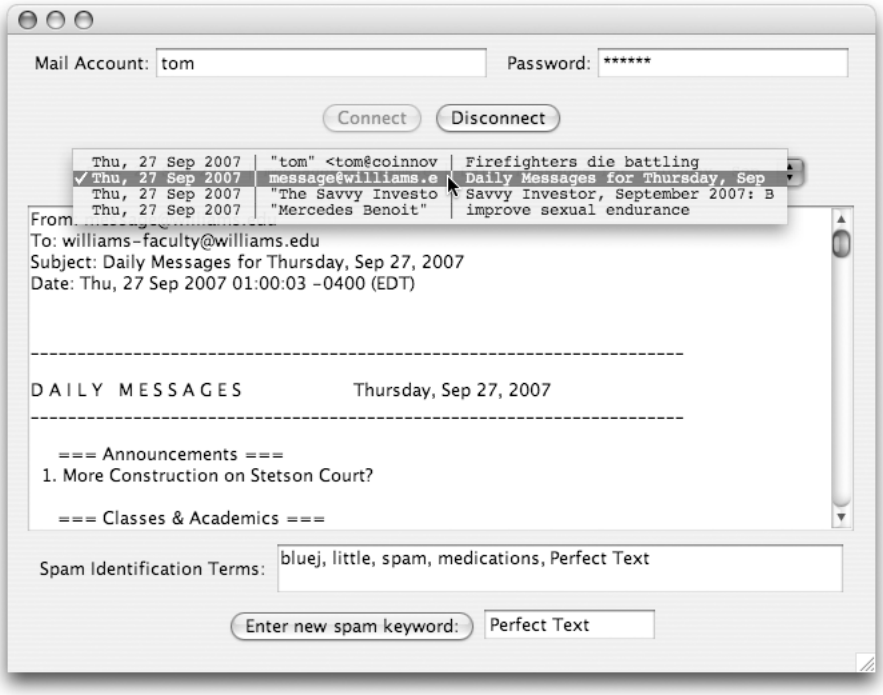

Figure 10.20: Message list filtered using a list of spam identification terms

For this program, however, we also need to be able to represent the list of spam identification terms shown in Figure 10.18. If you have not found the list of terms shown in Figure 10.18, look harder. There it is! Right under the email message displayed and to the right of the words "Spam Identification Terms" you will find a list of 0 spam identification terms.

At first, this may seem like an odd idea. It seems perfectly reasonable to think of 10 phrases as a "list" of phrases. On the other hand, getting to the point where we think of a single item as a list or collection, as we did with the `Journey` class, is a bit of a stretch. Now, we are asking you to think of a *no phrases at all* as a collection!

On the other hand, your experience with programming and mathematics should prepare you for the fact that sometimes it is very important to be able to explicitly talk about nothing. The number 0 is very important in mathematics. While 0 in some sense means nothing, the 0's in the number 100,001 convey some very important information. Getting $100,001 is very different from getting $11. In Java, the empty `String` (`""`) is very important. The empty `String` enables us to distinguish a `String` that contains nothing from no `String` at all. The latter is represented by the value `null`. If the `String` variable `s` is associated with the empty `String`, then `s.length()` produces 0. On the other hand, if no value has been associated with `s` or it has been explicitly set to `null`, then evaluating `s.length()` will lead to an error, producing a `NullPointerException` message.

Similarly, in many programs it is very helpful to have a class that can explicitly represent a collection that contains nothing. In particular, in our email program, such a collection will be the initial value associated with the variable used to keep track of our spam identification list. Accordingly, our Java definition for the `BlackList` class is based on the abstract definition:

BlackList (noun)

1. A single spam identification phrase followed by a BlackList.
2. Nothing at all.

This change in our base case requires only slight changes in the basic structure of the `BlackList` class compared to the `Journey` class. For the recursive case in the definition, we will still need two instance variables, one to refer to a single member of the collection and the other to refer recursively to the rest of the collection. We can also still use a `boolean` to distinguish the base case from the recursive case. Naming this `boolean singleStep`, however, would clearly be inappropriate. We will name it `empty` instead. The only other major difference is that instead of a constructor that takes one item and constructs a collection of size one, we need a constructor that takes no parameters and creates an empty collection or empty list. Based on these observations, the code for the instance variable and constructor definitions for the `BlackList` class are shown in Figure 10.21

Now, let us consider how to write the `looksLikeSpam` method that will enable a program to use a `BlackList` to filter spam. Recursive methods to process a collection in which the base case is empty resemble the methods we wrote for our `Journey` class in many ways. The body of such a method will typically have an `if` statement that distinguishes the base case from the recursive case. For our `BlackList` class we will do this by checking to see if `empty` is `true`. The code for the base case in such a method is typically very simple since there is no "first element" involved. For example, if there are no words in the `BlackList`, then `looksLikeSpam` should return `false` without even looking at the contents of the message.

The recursive case in the `looksLikeSpam` method will be more complex because the result produced by `looksLikeSpam` depends on the contents of the collection in an interesting way. The

```
public class BlackList {

    // Is this a black list with no terms in it?
    private boolean empty;

    // The last phrase added to the list
    private String badWord;

    // The rest of the phrases that belong to the list
    private BlackList otherWords;

    // Construct an empty black list
    public BlackList() {
        empty = true;
    }

    // Construct a black list by adding a new phrase to an existing list
    public BlackList( String newWord, BlackList existingList ) {
        empty = false;
        badWord = newWord;
        otherWords = existingList;
    }

    . . .
}
```

Figure 10.21: Instance variables and constructors for the `BlackList` class

```
// Check whether a message contains any of the phrases in this black list
public boolean looksLikeSpam( String message ) {
    if ( empty ) {
        return false;
    } else {
        if ( message.contains( badWord ) ) {
            return true;
        } else {
            return otherWords.looksLikeSpam( message );
        }
    }
}
```

Figure 10.22: A definition of `looksLikeSpam` emphasizing the cases and sub-cases

`toString` and `length` methods we defined for the `Journey` class always looked at every `Step` in a `Journey` before producing a result. The `looksLikeSpam` method will not need to do this. If the very first phrase in a `BlackList` appears in a message processed by `looksLikeSpam`, then the method can (and should) return `true` without looking at any of the other phrases in the `BlackList`. This means there are two sub-cases within the "recursive" case of the method. One case will deal with situations where the first phrase appears in the message. We will want to return `true` immediately in this case. Therefore, the code for this case will not actually be recursive. The other case handles situations where the first phrase does not appear in the message. In this case, we will use a recursive call to see if any of the other phrases in the `BlackList` occur in the message.

We will show two, equivalent versions of the Java code for `looksLikeSpam`. The first, shown in Figure 10.22, most closely reflects the approach to the method suggested above. The body of this method is an `if` statement that distinguishes between the base case and recursive case of the `BlackList` class definition. Within the second branch of the `if` statement, a nested `if` is used to determine whether or not the first phrase in the `BlackList` appears in the message and return the appropriate value.

A better approach, however was suggested in Section 5.3.1. There, we explained that in many cases, the nesting of `if` statements is really just a way to encode multi-way choices as a collection of two-way choices. We suggested that in such cases, extraneous curly braces might be deleted and indentation adjusted to more clearly suggest that a multi-way decision was being made. Applying that advice to the code in Figure 10.22 yields the code shown in Figure 10.23. This code more accurately reflects the structure of this method. Although the definition of the class involves only two cases, one base case and one recursive case, the definition of this method requires three cases, two of which are base cases and only one of which is recursive.

10.7 Recursive Removal

One useful feature we might want to add to our email client and to the `BlackList` class is the ability to remove terms from the spam list. We might discover that after entering some word like

```
// Check whether a message contains any of the phrases in this black list
public boolean looksLikeSpam( String message ) {
    if ( empty ) {
        return false;
    } else if ( message.contains( badWord ) ) {
        return true;
    } else {
        return otherWords.looksLikeSpam( message );
    }
}
```

Figure 10.23: A definition of `looksLikeSpam` making a 3-way choice

"little" that we had seen in many spam messages the program hid not only the spam messages but also many real messages. In such cases, it would be nice to be able to change your mind and tell the program to remove "little" or any other word from the list.

Figure 10.24 suggests a way the email client's interface might be modified to provide this functionality. Instead of displaying the spam terms the user has entered in a `JTextArea`, this version of the program displays them in a menu. The user can remove a term from the list by first selecting that term in the menu and then pressing the "Remove selected keyword" button.

Of more interest to us is how we would modify the interface of the `BlackList` class to provide the ability to remove terms. We will accomplish this by adding a method named `remove` to the class definition. The term to be removed will be passed to the method as a parameter. In defining this method, we will assume that only one copy of any term will appear in a `BlackList`.

Naming this method `remove` is a little misleading. It will not actually remove terms from an existing list. Instead, it will return a different list that is identical to the original except that the requested term will not appear in the new list.

The structure of the `remove` method will be similar to the `looksLikeSpam` method. It will have two base cases and one recursive case. The first base case handles empty `BlackList`s. If `remove` is invoked on an empty list, there is no work to do. The correct value to return is just an empty `BlackList`. We can do this by either creating a new, empty list or returning the original list. The second base case occurs when the term to be removed is the first term in the original `BlackList`. In this case, the method should simply return the rest of the `BlackList`. Finally, if the list is not empty but the term to be removed is not the first term in the list, the method must explicitly create and return a new `BlackList` composed of the original list's first element and the result of recursively removing the desired term from the rest of the original list. The code shown in Figure 10.25 reflects this structure.

To understand how this method works, consider the diagrams shown in Figures 10.26 and 10.27. These diagrams assume that the `BlackList` used to represent the items in the menu shown in Figure 10.24 have been associated with a variable declared as

```
private BlackList spamPhrases;
```

Figure 10.26 shows the collection of `BlackList` objects used to represent the collection of spam phrases before the remove operation is performed. Note that the order of the items in the linked

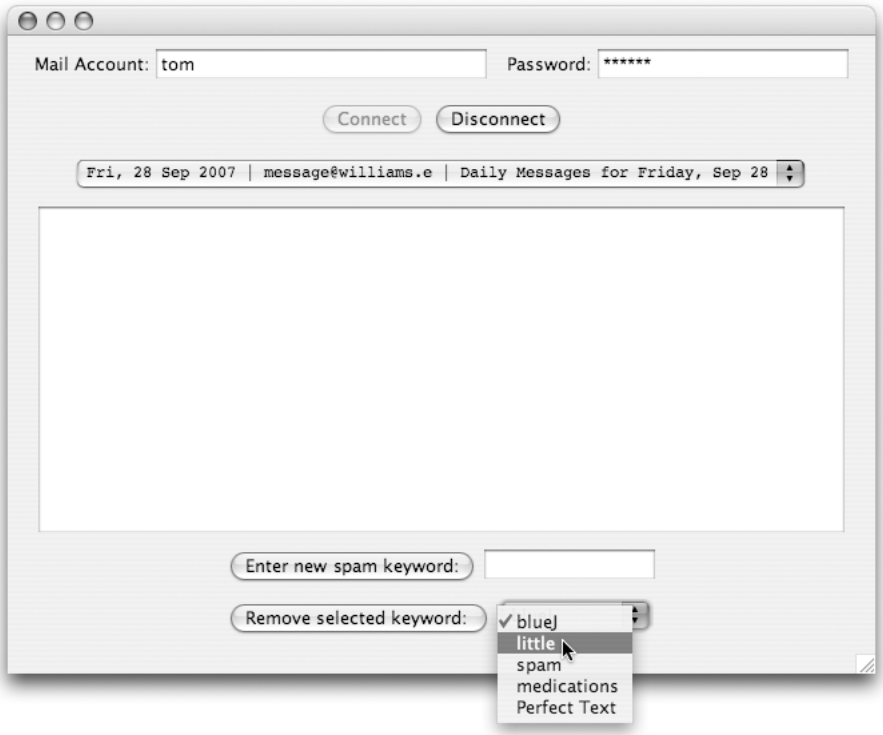

Figure 10.24: Email client providing the ability to add or remove spam terms

```
// Return a list obtained by removing the requested term from this list
public BlackList remove( String word ) {
    if ( empty ) {
        return this;
    } else if ( word.equals( badWord ) ) {
        return otherWords;
    } else {
        return new BlackList( badWord, otherWords.remove( word ) );
    }
}
```

Figure 10.25: A recursive `remove` method for the `BlackList` class

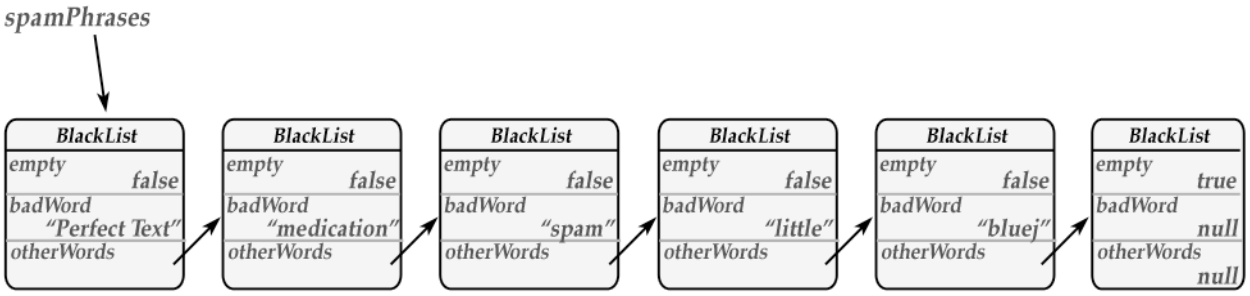

Figure 10.26: `BlackList` objects representing list shown in Figure 10.24

list is the opposite of the order in which we assumed the phrases were added to the list (and the opposite of the order in which they are displayed in the menu). This is because when we "add" an element by constructing a new `BlackList`, the new item appears first rather than last in the structure created.

Figure 10.27 shows how the structure would be changed if the statement

```
spamPhrases = spamPhrases.remove( "little" );
```

was used to remove the phrase "little" from the list in response to a user request. In this case, the computer would execute the final, recursive case in the definition of the `remove` method three times to process the entries for "Perfect text", "medication", and "spam". Each time this case in the method is executed, it creates a new `BlackList` object that is a copy of the object processed. Therefore, in Figure 10.27, we show three new objects that are copies of the first three objects in the original list. The objects that were copies are shown in light gray in the figure below the new copies.

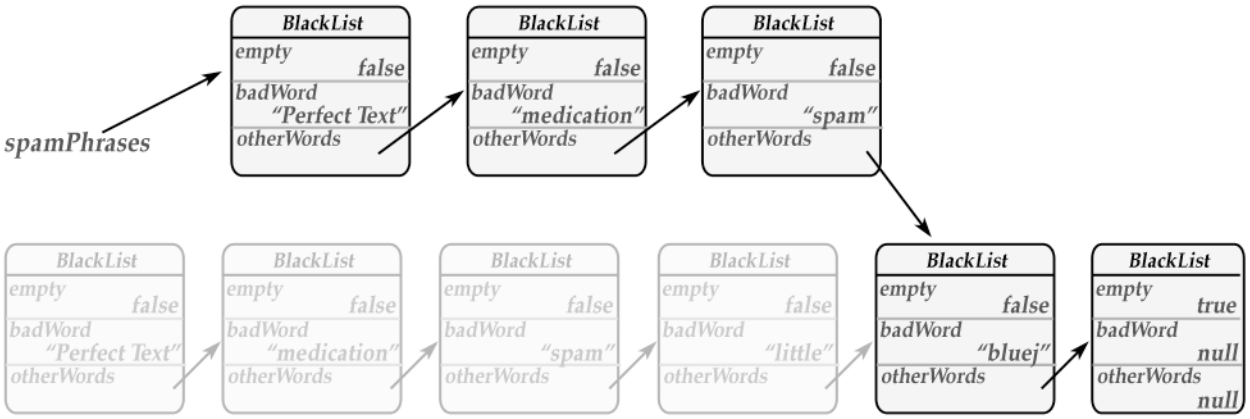

Figure 10.27: Objects representing spam list after removing "little"

The next recursive invocation would execute the second branch of the if statement. This is the branch that is executed once the item to be deleted is found. It does not make a copy of the

object to be deleted or of any other object. Instead, it returns the collection of objects referred to by `otherWords` within the object to be deleted. Because the method returns this value to the preceding recursive invocation, it becomes the value of `otherWords` in the last object that was copied. Therefore, in the figure, the `otherWords` variable within the new object for "spam" refers to the original object for "bluej".

Once the invocation of `remove` is complete, the name `spamPhrases` will be associated with the object it returns. Therefore, as shown in Figure 10.27, `spamPhrases` will now refer to the copy of the object for "Perfect Text" rather than to the original. Following the arrows representing the values of `otherWords` leads us through a list that correctly represents the reduced list of four spam phrases. Part of this list consists of objects that were part of the original list and part of it consists of new copies of objects from the original list.

If `spamPhrases` was the only variable name associated with the original `BlackList`, then there will be no name associated with the original object for "Perfect Text" after the `remove` is complete. This means that there will no longer be any way for the program to refer to this object or any of the other three original objects that are not part of the new list. That is why we have shown these objects in gray in the figure. The Java system will eventually recognize that these `BlackList` objects are no longer usable and remove them from the computer's memory.

10.8 Wrapping Up

In the preceding section, we noted that the processes of removing elements from a `BlackList` often involves creating new `BlackList` objects rather than simply modifying existing objects. Adding an element to a `BlackList` is similar. When add an element by using the `BlackList` constructor to make a new, bigger `BlackList` rather than by modifying an existing `BlackList`. Adding or removing an item from a `BlackList` always involves assigning a new value to some `BlackList` variable. That is, if we declare

```
String someWord;
BlackList spamTerms;
```

then we can add a word to `spamTerms` by executing the assignment

```
spamTerms = new BlackList( someWord, spamTerms );
```

and we can remove the word by executing

```
spamTerms = spamTerms.remove( someWord );
```

Recursive structures are one of many ways to define a collection of objects. The `JComboBox` class, one of the library classes we have used extensively, also provides the ability to manipulate collections. The `JComboBox` handles the addition and removal of entries very differently from our `BlackList` class. If we declare

```
JComboBox menu = new JComboBox();
```

then we can add an item by executing the invocation

```
menu.addItem( someWord );
```

```
public void addItem( String word ) {
    if ( empty ) {
        empty = false;
        badWord = word;
        otherWords = new BlackList();
    } else {
        otherWords.addItem( word );
    }
}
```

Figure 10.28: An `addItem` method for `BlackList`s

and remove an item using the invocation

```
menu.removeItem( someWord );
```

Neither of these are assignment statements. When we add or remove items from `JComboBox`es, we don't create a new `JComboBox` or associate a new object with a variable. We simply invoke a mutator method that changes an existing `JComboBox`.

The `addItem` and `removeItem` methods of the `JComboBox` have several advantages over the interface our `BlackList` class provides for adding and removing elements. It is not uncommon to write a program in which a single object is shared between two different classes. Suppose we want to share a `BlackList` between two classes named A and B. Class A might pass a `BlackList` associated with instance variable "x" as a parameter to the constructor of class B. Within its constructor, B might associate this `BlackList` with the instance variable "y". Unfortunately, if B tries to add an item by executing the statement

```
y = new BlackList( ..., y );
```

this addition will not be shared with A. On the other hand, if B could say

```
y.addItem( ... );
```

as it might with a `JComboBox`, the change would be shared with A.

It is possible to define methods like `addItem` and `removeItem` as part of a recursively defined linked list like the `BlackList`. A possible definition of `addItem` for the `BlackList` class is shown in Figure 10.28. Such methods are more complicated than the simple approaches we used to add and remove items earlier in this chapter, and, in the case of `addItem`, less efficient. The `addItem` method shown adds new items at the end of a list and looks at every entry in the existing list in the process. By contrast, the technique of creating a new list with the new item at the start only takes a single step.

As a result, a common alternate technique used to provide functionality similar to the `addItem` and `removeItem` methods is to "wrap" a recursive collection class definition within a simple non-recursive class that implements methods like `addItem` and `removeItem` using the constructor and `remove` method of the underlying recursive class.

The class `SpamFilter` shown in Figure 10.29 is such a wrapper for the `BlackList` class. The `SpamFilter` class contains only one instance variable that is used to refer to the `BlackList` it

```
// A wrapper for the recursive BlackList class that provides addItem
// and removeItem methods in addition to the essential looksLikeSpam method
public class SpamFilter {

    // The underlying recursive collection
    private BlackList wordList;

    // Create a new filter
    public SpamFilter() {
        wordList = new BlackList();
    }

    // Add a phrase to the list of terms used to filter spam
    public void addItem( String newWord ) {
        wordList = new BlackList( newWord, wordList );
    }

    // Remove a phrase from the list of terms used to filter spam
    public void removeItem( String word ) {
        wordList = wordList.remove( word );
    }

    // Check whether the text of a message contains any of the phrases
    // in this black list
    public boolean looksLikeSpam( String message ) {
        return wordList.looksLikeSpam( message );
    }

}
```

Figure 10.29: `SpamFilter`: a non-recursive wrapper for the `BlackList` class

manages. The `addItem` and `removeItem` methods of the class provide the ability to add and remove items from a collection using an interface similar to that provided by the `JComboBox` class. Internally, however, these methods are implemented using the constructor and `remove` method of the `BlackList` class. The goal is simply to hide the constructor and `remove` method from the rest of the program. Wherever one might have declared a variable such as

```
BlackList badWords;
```

elsewhere in a program, one would now instead say

```
SpamFilter badWords;
```

More importantly, wherever one said

```
badWords = new BlackList( ..., badWords );
```

one would now instead say

```
badWords.addItem( ... );
```

Similarly, all statements of the form

```
badWords = badWords.remove( ... );
```

could be replaced by

```
badWords.removeItem( ... );
```

Finally, when defining such a wrapper class it is typically desirable to make other aspects of the interface to the wrapper class identical or at least similar to the underlying recursive class. For example, the `SpamFilter` class defines a method named `looksLikeSpam` that provides the same interface as the similarly named method of the `BlackList` class . This method is implemented by simply invoking the corresponding method of the underlying class. As a result, any statement containing a condition of the form

```
badWords.looksLikeSpam( ... )
```

can remain unchanged if we switch to use a `SpamFilter` in place of a `BlackList`.

10.9 Summary

In this chapter, we have explored the application of recursive definitions to manipulate collections of objects. A definition is said to be recursive if it depends on itself. The most direct way this can happen is if the name being defined is used within its own definition. This is the form of recursion we have discussed in this chapter.

We have seen examples of classes which are recursive because they have instance variables whose types are the same as the type of the class in which they are defined. We have also seen examples of method definitions which are recursive because the include invocations that apply the method begin defined. In this chapter, all of the recursive methods have appeared within recursive classes, although this is not necessary in general.

A recursive definition must be divided into several cases. If there were only one case in the definition and it referred to the name being defined then the definition would truly be circular and therefore unusable. By having several cases, the definition can include some cases called *base cases* that do not involve the name being defined and other cases that do refer to the name being defined.

We focused our attention on the use of recursive definitions to represent collections of objects. In this application, the base case of a class definition is typically either the empty collection or a collection of some small fixed size. The recursive case is then based on the fact that any collection can be seen as one item plus another collection that is one smaller than the original.

The structure of recursive methods that manipulate collections typically reflects the base case/recursive case used in the definition of the class to which the method belongs. Other cases in such definitions involve the single distinguished item that is set apart from the rest of the collection.

While we have introduced many of the important principles one must understand to use recursion, we have kept our exploration of recursion in this chapter narrowly focused on the manipulation of collections viewed as lists. As you learn more about programming, you will learn that recursion can be used in many more ways. For example, instead of using the name of a class directly in its own definition, we can define one class in terms of a second class that is in turn defined using the first class. Such collections of classes are said to be mutually recursive. Only when you learn to use recursion in these more general ways will you fully appreciate the power of this technique.