FOREWORD

Learning to program is more than learning the syntactic and semantic rules of a programming language. It also requires learning how to design programs. Any good book on programming must therefore teach program design.

Like any other form of design, program design has competing schools. These schools are often associated with a particular set of languages. Since Java is an object-oriented programming language, people teaching Java should emphasize object-oriented design.

Felleisen and Friedman show that the functional (input-output driven) method of program design naturally leads to the use of well-known object-oriented design patterns. In fact, they integrate the two styles seamlessly and show how well they work together. Their book proves that the functional design method does not clash with, but supports object-oriented programming.

Their success doesn’t surprise me, because I’ve seen it in Smalltalk for many years, though unfortunately, it seems to have remained one of the secrets of object-oriented design. I am happy to see that Felleisen and Friedman have finally exposed it. This book will be especially useful if you are a C++ programmer learning Java, since you probably haven’t seen functional program design before. If you know functional design, the book will gently introduce you to pattern-based programming in Java. If you don’t know it, Felleisen and Friedman will teach you a powerful new way of thinking that you should add to your design toolbox.

Enjoy the pizzas!

Ralph E. Johnson
Champaign, Illinois
Preface

An object-oriented programming language enables a programmer to construct reusable program components. With such components, other programmers can quickly build large new programs and program fragments. In the ideal case, the programmers do not modify any existing code but simply glue together components and add a few new ones. This reusability of components, however, does not come for free. It requires a well-designed object-oriented language and a strict discipline of programming.

Java is such a language, and this book introduces its object-oriented elements: (abstract) classes, fields, methods, inheritance, and interfaces. This small core language has a simple semantic model, which greatly helps programmers to express themselves. In addition, Java implementations automatically manage the memory a program uses, which frees programmers from thinking about machine details and encourages them to focus on design.

The book's second goal is to introduce the reader to design patterns, the key elements of a programming discipline that enhances code reuse. Design patterns help programmers organize their object-oriented components so that they properly implement the desired computational process. More importantly still, design patterns help communicate important properties about a program component. If a component is an instance of an explicitly formulated pattern and documented as such, other programmers can easily understand its structure and reuse it in their own programs, even without access to the component's source.

The Intended Audience

The book is primarily intended for people—practicing programmers, instructors and students alike—who wish to study the essential elements of object-oriented programming and the idea of design patterns. Readers must have some basic programming experience. They will benefit most from the book if they understand the principles of functional design, that is, the design of program fragments based on their input-output behavior. An introductory computer science course that uses Scheme (or ML) is the best way to get familiar with this style of design, but it is not required.

What this Book is Not About

Java provides many useful features and libraries beyond its object-oriented core. While these additional Java elements are important for professional programming, their coverage would distract from the book's important goals: object-oriented programming and the use of design patterns. For that reason, this book is not a complete introduction to Java. Still, readers who master its contents can quickly become skilled Java programmers with the supplementary sources listed in the Commencement.

The literature on design patterns evolves quickly. Thus, there is quite a bit more to patterns than an introductory book could intelligibly cover. Yet, the simplicity of the patterns we use and the power that they provide should encourage readers to study the additional references about patterns mentioned at the end of the book.

Acknowledgments

We are indebted to many people for their contributions and assistance throughout the development of this book. Several extensive discussions with Shriram Krishnamurthi, Jon Rossie,
and Mitch Wand kept us on track; their detailed comments deeply influenced our thinking at critical junctures. Michael Ashley, Sundar Balasubramaniam, Cynthia Brown, Peter Drake, Bob Fink, Robby Findler, Steve Ganz, Paul Graumke, John Greiner, Erik Hilsdale, Matthew Kudzin, Julia Lawall, Shinn-Der Lee, Michael Levin, Gary McGraw, Benjamin Pierce, Amr Sabry, Jonathan Sobel, and George Springer read the book at various stages of development and their comments helped produce the final result. We also wish to thank Robert Prior at MIT Press who loyally supported us for many years and fostered the idea of a “Little Java.” The book greatly benefited from Dorai Sitaram’s incredibly clever Scheme typesetting program \textsc{Sl\textsc{i}TeX}. Finally, we would like to thank the National Science Foundation for its continued support and especially for the Educational Innovation Grant that provided us with the opportunity to collaborate for the past year.

\textbf{Reading Guidelines}

Do not rush through this book. Allow seven sittings, at least. Read carefully. Mark up the book or take notes; valuable hints are scattered throughout the text. Work through the examples, don’t scan them. Keep in mind the motto “Think first, experiment later.”

The book is a dialogue about interesting Java programs. After you have understood the examples, experiment with them, that is, modify the programs and examples and see how they behave. Since most Java implementations are unfortunately batch interpreters or compilers, this requires work of a repetitive nature on your side. Some hints on how to experiment with Java are provided on the following pages.

We do not give any formal definitions in this book. We believe that you can form your own definitions and thus remember and understand them better than if we had written them out for you. But be sure you know and understand the bits of advice that appear in most chapters.

We use a few notational conventions throughout the text to help you understand the programs on several levels. The primary conventions concern typeface for different kinds of words. Field and method names are in \textit{italic}. Basic data, including numbers, booleans, and constructors introduced via datatypes are set in \textsc{sans serif}. Keywords, e.g., \texttt{class}, \texttt{abstract}, \texttt{return} and \texttt{interface} are in \textbf{boldface}. When you experiment, you may ignore the typefaces but not the related framemotes. To highlight this role of typefaces, the programs in framemotes are set in a \texttt{typewriter} face.

Food appears in many of our examples for two reasons. First, food is easier to visualize than abstract ideas. (This is not a good book to read while dieting.) We hope the choice of food will help you understand the examples and concepts we use. Second, we want to provide you with a little distraction. We know how frustrating the subject matter can be, and a little distraction will help you keep your sanity.

You are now ready to start. Good luck! We hope you will enjoy the experiences waiting for you on the following pages.

\begin{flushright}
Bon appétit!
\end{flushright}

Matthias Felleisen
Daniel P. Friedman
EXPERIMENTING WITH JAVA

Here are some hints on how to experiment with Java.\footnote{See Arnold and Gosling \cite{arnold1997} for details on how they work. These hints make little sense out of context, so for now, just follow them as you read this book.}

1. Create a file that contains a complete hierarchy of classes.

2. To each class whose name does not end with a superscript $D$, $V$, $I$, or $M$, add a `toString` method according to these rules:
   a) if the class does not contain any fields, use

   ```java
   public String toString() {
       return "new " + getClass().getName() + ";";
   }
   ```

   b) if the class has one field, say $x$, use

   ```java
   public String toString() {
       return "new " + getClass().getName() + "(" + x + ")";
   }
   ```

   c) if the class has two fields, say $x$ and $y$, use

   ```java
   public String toString() {
       return "new " + getClass().getName() + "(" + x + ", " + y + ")";
   }
   ```

3. Add the following class at the bottom of the file:

   ```java
   class Main {
       public static void main(String args[]) {
           DataType_or_Interface y = new _ _ _ _ _;
           System.out.println( ....... );
       }
   }
   ```

   With `DataType_or_Interface y = new _ _ _ _ _`, create the object $y$ with which you wish to experiment. Then replace `.......` with the example expression that you would like to experiment with. For example, if you wish to experiment with the `distanceToO` method of `ManhattanPt` as defined in chapter 2, add the following definition to the end of your file:

   ```java
   class Main {
       public static void main(String args[]) {
           PointD y = new ManhattanPt(2,8);
           System.out.println( y.distanceToO() );
       }
   }
   ```
If you wish to experiment with a sequence of expressions that modify y, as in chapter 10, e.g.,

```
  y. = = = =
  y. = = = =
  y. = = = =
```

replace . . . . with

```
  y. = = = = + "\n " +
  y. = = = = + "\n " +
  y. = = = =
```

For example, if you wish to experiment with the methods of PiemanM as defined in chapter 10, add the following definition to the end of your file:

```java
public class Main {
    public static void main(String args[ ]) {
        PiemanM y = new PiemanM();
        System.out.println(
            y.addTop(new Anchovy()) + "\n " +
            y.addTop(new Anchovy()) + "\n " +
            y.substTop(new Tuna(), new Anchovy()) );
    }
}
```

4. Finally, compile the file and interpret the class Main.
1. Modern Toys
Is 5 an integer?  

Yes, it is.

Is this a number: -23?  

Yes, but we don’t use negative integers.

Is this an integer: 5.32?  

No, and we don’t use this type of number.

What type of number is 5?  

\textit{int}.\textsuperscript{1}  

\textsuperscript{1} In Java, \textit{int} stands for “integer.”

Quick, think of another integer!  

How about 19?

What type of value is true?  

\textit{boolean}.

What type of value is false?  

\textit{boolean}.

Can you think of another \textit{boolean}?  

No, that’s all there is to \textit{boolean}.

What is \textit{int}?  

A type.

What is \textit{boolean}?  

Another type.

What is a type?  

A type is a name for a collection of values.

What is a type?  

Sometimes we use it as if it were the collection.

Can we make new types?  

We don’t know how yet.
Draw the picture that characterizes the essential relationships among the following classes.

abstract class Seasoning\textsuperscript{D} {}

class Salt extends Seasoning\textsuperscript{D} {}

class Pepper extends Seasoning\textsuperscript{D} {}

\textsuperscript{D} This superscript is a reminder that the class is a datatype. Lower superscripts when you enter this kind of definition in a file: Seasoning\textsuperscript{D}.

14 Is this it?

Yes. We say Seasoning\textsuperscript{D} is a datatype, and Salt and Pepper are its variants.

15 Okay. But aren’t all three classes introducing new types?

Yes, in a way. Now, is new Salt() a Seasoning\textsuperscript{D}?

16 Yes, it is, because new Salt() creates an instance of Salt, and every instance of Salt is also a Seasoning\textsuperscript{D}.

And new Pepper()?

17 It’s also a Seasoning\textsuperscript{D}, because new Pepper() creates an instance of Pepper, and every instance of Pepper is also a Seasoning\textsuperscript{D}.

What are abstract, class, and extends?

18 Easy:

abstract class introduces a datatype,
class introduces a variant, and
extends connects a variant to a datatype.

Is there any other Seasoning\textsuperscript{D}?  

19 No, because only Salt and Pepper extend Seasoning\textsuperscript{D}. \textsuperscript{1}

\textsuperscript{1} Evaluating new Salt() twice does not produce the same value, but we ignore the distinction for now.
Correct, Salt and Pepper are the only variants of the datatype Seasoning. Have we seen a datatype like Seasoning before?

No, but boolean is a type that also has just two values.

Let’s define more Seasoning’s.

```java
class Thyme extends Seasoning {}
class Sage extends Seasoning {}
```

We can have lots of Seasoning’s.

And then there were four.

Yes.

What is a Cartesian point?

It is basically a pair of numbers.

What is a point in Manhattan?

An intersection where two city streets meet.

How do CartesianPt and ManhattanPt differ from Salt and Pepper?

```java
abstract class Point {}

class CartesianPt extends Point {
    int x;
    int y;
    CartesianPt(int x, int y) {
        x = x;
        y = y;
    }
}

class ManhattanPt extends Point {
    int x;
    int y;
    ManhattanPt(int x, int y) {
        x = x;
        y = y;
    }
}
```

Each of them contains three things between { and }. The x and the y are obviously the coordinates of the points. But what is the remaining thing above the bold bar?¹

¹ This bar indicates the end of the constructor definition. It is used as an eye-catching separator. We recommend that you use "-----------------------------" when you enter it in a file.
The underlined occurrences of \texttt{CartesianPt} and \texttt{ManhattanPt} introduce the constructors of the respective variants.

A constructor is used with \texttt{new} to create new instances of a \texttt{class}.

When we create a \texttt{CartesianPt} like this:
\begin{verbatim}
new CartesianPt(2,3),
\end{verbatim}
we use the constructor in the definition of \texttt{CartesianPt}.

So now we have created a \texttt{CartesianPt} whose \texttt{x} field is 2 and whose \texttt{y} field is 3. And because \texttt{CartesianPt} \texttt{extends} \texttt{Point\textsuperscript{P}}, it is also a \texttt{Point\textsuperscript{P}}.

Correct. Is this a \texttt{ManhattanPt}:
\begin{verbatim}
new ManhattanPt(2,3)?
\end{verbatim}

Yes, and its \texttt{x} field is 2 and its \texttt{y} field is 3.

Isn’t all this obvious?

Mostly, but that means we have used constructors before without defining them. How does that work?

When a \texttt{class} does not contain any fields, as in \texttt{Salt} and \texttt{Pepper}, a constructor is included by default.

And that’s the constructor we used before, right?

Yes, that’s correct. Default constructors never consume values, and, when used with \texttt{new}, always create objects without fields.

Good. But what is \texttt{new Point\textsuperscript{P}()}?

\texttt{An abstract} class is by definition incomplete, so \texttt{new} cannot create an instance from it.

That makes sense. Let’s move on.
Do the following classes define another datatype with variants?

```java
abstract class NumD {}

class Zero extends NumD {}

class OneMoreThan extends NumD {
    NumD predecessor;
    OneMoreThan(NumD .p) {
        predecessor = .p;
    }
}
```

Yes, they define a datatype and two variants.

Draw the picture, too.

Is this a NumD:

```java
new Zero();
```

Obviously, just like new Salt() is a SeasoningD.

Is this a NumD:

```java
new OneMoreThan(
    new Zero());
```

Yes, because OneMoreThan constructs a NumD from a NumD, and every instance of OneMoreThan is also a NumD.

How does OneMoreThan do that?

We give it new Zero(), which is a NumD, and it constructs a new NumD.

And what does it mean to construct this new instance?

This new instance of OneMoreThan is a value with a single field, which is called predecessor. In our example, the field is new Zero().

Does predecessor always stand for an instance of Zero?

No, its type says that it stands for a NumD, which, at the moment, may be either a Zero or a OneMoreThan.

Modern Toys
What is
new One More Than(
new One More Than(
new Zero()))?

40 A Num^P, because One More Than constructs an instance from a Num^P and we agreed that
new One More Than(
new Zero())
is a Num^P.

What is
new One More Than(
0)?

41 That is nonsense,^1 because 0 is not a Num^P.

^1 We use the word “nonsense” to refer to expressions for which Java cannot determine a type.

Is new Zero() the same as 0?

42 No. 0 is similar to, but not the same as.
new Zero().

Is
new One More Than(
new Zero())
like
1?

43 1 is similar to, but not the same as.
new One More Than(
new Zero()).

And what is
new One More Than(
new One More Than(
new One More Than(
new One More Than(
new Zero()))))
similar to?

44 4.

Are there more Num^P’s than bools?

45 Lots.

Are there more Num^P’s than ints?

46 No.^1

^1 This answer is only conceptually correct. Java limits the number of ints to approximately 2^32.
What is the difference between new Zero() and 0?

Correct. In general, if two things are instances of two different basic types, they cannot be the same.

Easy: new Zero() is an instance of Zero and, by implication, is a Num^P, whereas 0 is an int. This makes it difficult to compare them, but we can compare them in our minds.

The primitive types (int and boolean) are distinct; others may overlap.

So are types just names for different collections with no common instances?

What are non-basic types?

Class definitions do not introduce primitive types. For example, a value like new Zero() is not only an instance of Zero, but is also a Num^P, which is extended by Zero. Indeed, it is of any type that Num^P extends, too.

And what is that?

Every class that does not explicitly extend another class implicitly extends the class Object.

This must mean that everything is an Object.

Almost. We will soon see what that means.

Okay.

The First Bit of Advice

When specifying a collection of data, use abstract classes for datatypes and extended classes for variants.

Modern Toys
What do the following define?

```java
abstract class LayerD {}
```

53 They define a new datatype and its two variants. The first variant contains a field of type `Object`.

```java
class Base extends LayerD {
    Object o;
    Base(Object _o) {
        o = _o;
    }
}
```

54 It looks like an instance of `Base`, which means it is also a `LayerD` and an `Object`.

```java
class Slice extends LayerD {
    LayerD l;
    Slice(LayerD _l) {
        l = _l;
    }
}
```

55 It also looks like an instance of `Base`. But how come both
```
new Base{
    new Zero()
}
```
and
```
new Base{
    new Salt()
}
```
are instances of the same variant?

They are, because everything created with `new` is an `Object`, the class of all objects.

56 Hence, we can use both
```
new Zero()
```
and
```
new Salt()
```
for the construction of a `Base`, which requires an `Object`. 

Chapter 1
Is anything else an Object?  

Correct. Is this a Layer\textsuperscript{D}:
\begin{verbatim}
new Base(5)
\end{verbatim}  

Is this a Layer\textsuperscript{D}:
\begin{verbatim}
new Base(false)
\end{verbatim}  

Correct again! How about this Layer\textsuperscript{D}:
\begin{verbatim}
new Base(new Integer(5))
\end{verbatim}  

Guess how we create a Layer\textsuperscript{D} from false?

Is it confusing that we need to connect \texttt{int} with \texttt{Integer} and \texttt{boolean} with \texttt{Boolean}?

Ready for more?

---

57 We said that only things created with \texttt{new} are Objects.\textsuperscript{1}

1 Arrays and strings are objects, too. We don’t discuss them.

58 5 is not created with \texttt{new}, so this must be nonsense.

59 \texttt{false} is not created with \texttt{new}, so this must be nonsense, too.

60 This must mean that \texttt{Integer} creates an object from an \texttt{int}.

61 Easy now:
\begin{verbatim}
new Base(new Boolean(false)).
\end{verbatim}

62 Too much coffee does that.

63 Can’t wait.
2. Methods to Our Madness
Remember points?

```java
abstract class PointD {
    Point
}
```

```java
class CartesianPt extends PointD {
    int x;
    int y;
    CartesianPt(int x, int y) {
        x = x;
        y = y;
    }
}
```

```java
class ManhattanPt extends PointD {
    int x;
    int y;
    ManhattanPt(int x, int y) {
        x = x;
        y = y;
    }
}
```

Sure, we just talked about them. But what are these labeled ovals about?

We will find out soon. Did you notice the big white space on the right?

It must be for drawing the picture of the classes.

How far is

```java
new ManhattanPt(3,4)
```

from the Empire State Building?

If the Empire State Building is the origin, we have to walk seven blocks: 3 over, 4 up.

And how far is

```java
new CartesianPt(3,4)
```

from the origin?

5, which is $\sqrt{3^2 + 4^2}$.  

Methods to Our Madness 13
Write the methods `distanceToO` using `.`, `+`, `Math.sqrt`, and `-`, which determine how far a point is from the origin. Of course, you can’t write these methods yet. Okay, you deserve something sweet for enduring this last question.

What do the methods produce? *ints*, which represent the distances to the origin.

Here they are.

```
abstract int distanceToO();
```

* They correspond to the unexplained labels in the definition of the datatype and its variants.

```
int distanceToO() {
    return Math.sqrt(x*x + y*y);
}
```

```
int distanceToO() {
    return x + y;
}
```

To what do `Point`, `CartesianPt`, and `ManhattanPt` in the boxes refer?

1 When you enter this in a file, use `Math.sqrt(x*x + y*y)`. `Math` is a class that contains `sqrt` as a static method. Later we will see what `{int}` means.

The labels remind us that we need to insert these methods into `Point`, `CartesianPt`, and `ManhattanPt`.

That’s simple enough.

How many times have we defined the method `distanceToO`? Three times, but the first one differs from the other two. It is labeled `abstract`, while the others are not preceded by a special word.
Do **abstract** methods belong to the **abstract** class?  
Yes, they always do.

**An abstract** method in an **abstract** class introduces an obligation, which says that all concrete classes that extend this abstract class\(^1\) must contain a matching method definition.

\(^1\) Directly or indirectly. That is, the concrete class may extend an abstract class that extends the abstract class with the obligation and so on.

What is the value of

```java
new ManhattanPt(3,4)
.distanceToO();
```

7.

How do we arrive at that value?  
We determine the value of 

\[ x + y, \]

with \( x \) replaced by 3 and \( y \) replaced by 4.

What is the value of

```java
new CartesianPt(3,4)
.distanceToO();
```

5, because that is the value of 

\[ \sqrt{x^2 + y^2} \]

with \( x \) replaced by 3 and \( y \) replaced by 4.

What does \( \lfloor \sqrt{x} \rfloor \) compute?  
The largest **int** that does not exceed the square root of \( x \).

Time for a short break?  
An apple a day keeps the dentist away. A cup of coffee does not.
Here is another datatype with its variants. What is different about them?

```java
abstract class ShishD {
    Shish
}

class Skewer extends ShishD {
    Skewer
}

class Onion extends ShishD {
    ShishD s;
    Onion(ShishD s) {
        s = s;
    }
    Onion
}

class Lamb extends ShishD {
    ShishD s;
    Lamb(ShishD s) {
        s = s;
    }
    Lamb
}

class Tomato extends ShishD {
    ShishD s;
    Tomato(ShishD s) {
        s = s;
    }
    Tomato
}
```

Did you notice the big space on the right? Yes, isn't it for drawing the picture of the classes?

17 It is like NumD but has more variants.

18 Yes, isn't it for drawing the picture of the classes?
Construct a Shish$^D$. 

Yes, every Skewer is also a Shish$^D$. How about another one?

And a third?

Are there only Onions on this Shish$^D$: 
new Skewer()?

Are there only Onions on this Shish$^D$: 
new Onion( 
new Skewer())?

And how about: 
new Lamb( 
new Skewer())?

Is it true that 
new Onion( 
new Onion( 
new Onion( 
new Skewer())))
contains only Onions?

And finally: 
new Onion( 
new Lamb( 
new Onion( 
new Skewer())))?

How about 
new Skewer()?

Here's one: 
new Onion( 
new Skewer()).

Here's one more: 
new Onion( 
new Lamb( 
new Onion( 
new Skewer()))).

true, because there is neither Lamb nor Tomato on new Skewer().

true.

false.

true.

false.
Write the methods onlyOnions using { }, { }, . . . ; true, false, return, and boolean.

1 A better name for these methods would be nothingButOnions.

Of course, you can’t write these methods, yet. Okay, you deserve a lollipop for enduring this kind of question again.

And what do they produce? booleans.

Here are the methods.

abstract boolean onlyOnions();

boolean onlyOnions() {
    return true;
}

Shish

boolean onlyOnions() {
    return s.onlyOnions();
}

Skewer

boolean onlyOnions() {
    return false;
}

Onion

Lamb

boolean onlyOnions() {
    return false;
}

Tomato

Did you notice the labels in the boxes?

Good. How many methods have we defined? Five, but the first one is abstract; the others are concrete.
Do **abstract** methods belong to the **abstract** class?

Yes, we said so.

Does each variant of Shish\textsuperscript{D} contain a method called `onlyOnions`?

Yes, because Shish\textsuperscript{D} contains an **abstract** method called `onlyOnions` that obligates each variant to define a matching, **concrete** method.

Is this always the case?

Always.

What do these concrete methods consume?

Nothing, just as the **abstract** method says.

What do these concrete methods produce?

**booleans**, just as the **abstract** method says.

What is the value of

```java
new Onion(
    new Onion(  
        new Skewer())
    .onlyOnions());
```

**true**.

And how do we determine the value of

```java
new Onion(  
    new Onion(  
        new Skewer())
    .onlyOnions());
```

We will need to pay attention to the method definitions.

Which definition of `onlyOnions` must we use to determine the value of

```java
new Onion(  
    new Onion(  
        new Skewer())
    .onlyOnions());
```

This object is an instance of `Onion`, so we need to use the definition of `onlyOnions` that belongs to the `Onion` variant.
What follows the word `return` in the `onlyOnions` method in `Onion`?

Who is the field `s` of the object `new Onion
  new Onion(new Skewer())`?

Does `s` always stand for an `Onion`?

Then what is `s.onlyOnions()`?

Why do we need to know the meaning of `new Onion
  new Skewer()
  .onlyOnions()`?

How do we determine the answer for `new Onion
  new Skewer()
  .onlyOnions()`?

Because the answer for `new Onion
  new Skewer()
  .onlyOnions()` is also the answer for `new Onion
  new Onion(new Skewer())
  .onlyOnions()`.

Let's see.
Which definition of `onlyOnions` must we use to determine the value of `new Onion(
    new Skewer())
  .onlyOnions()? 

This object is an instance of Onion, so we need to use the definition of `onlyOnions` that belongs to the Onion variant.

What follows the word `return` in the `onlyOnions` method in Onion? 

`s.onlyOnions()`.

What is the field `s` of the object `new Onion(
    new Skewer())`? 

`new Skewer()`.

Then what is `s.onlyOnions()`? 

It is

```
new Skewer()
  .onlyOnions(),
```

just as we would have expected.

Why do we need to know the meaning of `new Skewer()
  .onlyOnions()`? 

Because the answer for `new Skewer()
  .onlyOnions()` is also the answer for `new Onion(
    new Skewer())
  .onlyOnions()`, which in turn is the answer for `new Onion(
    new Onion(
      new Skewer()))
  .onlyOnions()`.

How do we determine the answer for `new Skewer()
  .onlyOnions()`? 

We need to determine one more time which version of `onlyOnions` we must use.
Is `new Skewer()` a `Skewer`?

Obviously.

Then what is the answer?

true.

Why?

Because true is what the `onlyOnions` method in `Skewer` always returns.

Are we done?

Yes! The answer for `new Onion(  
    new Onion(  
        new Skewer()))  
    .onlyOnions()`

is the same as the answer for `new Onion(  
    new Skewer())  
    .onlyOnions()`,

which is the same as the answer for `new Skewer()  
    .onlyOnions()`,

which is

true.

What is the value of `new Onion(  
    new Lamb(  
        new Skewer()))  
    .onlyOnions()`?

false, isn’t it?

Which definition of `onlyOnions` must we use to determine the value of `new Onion(  
    new Lamb(  
        new Skewer()))  
    .onlyOnions()`?

This object is an instance of `Onion`, so we need to use the definition of `onlyOnions` that belongs to the `Onion` variant.
What follows the word return in the onlyOnions method in Onion?

What is the field s of the object new Onion(new Lamb(new Skewer()))?

Then what is s.onlyOnions()?

Why do we need to know the meaning of new Lamb(new Skewer()).onlyOnions()?

How do we determine the answer for new Lamb(new Skewer()).onlyOnions()?

And?

And now what is the answer?

s.onlyOnions().

It is the object built from new Lamb(new Skewer()).

It is new Lamb(new Skewer()).onlyOnions(), of course.

Because the answer for new Lamb(new Skewer()).onlyOnions() is also the answer for new Onion(new Lamb(new Skewer())).onlyOnions().

We determine which version of onlyOnions to use.

We use the one that belongs to Lamb.

false, because false follows the word return in the corresponding method definition in Lamb.
Are we done?  

64 Yes! The answer for
new Onion(
    new Lamb(
        new Skewer(()))
) .onlyOnions()
is the same as the answer for
new Lamb(
    new Skewer(()))
) .onlyOnions(),
which is
false.

Describe the methods (i.e., the function) *onlyOnions* in your own words.  

65 Here are our words:
"The methods determine for a Shish$^P$ whether its contents are edible by an onion lover."

Describe how the methods (i.e., the function) *onlyOnions* accomplish this.  

66 Here are our words again:
"For each layer of the Shish$^P$, except for Onion, the corresponding method knows whether it is good or bad. The method for Onion needs to determine whether the remaining layers are only Onions sitting on a Skewer."

Is
new Tomato(
    new Skewer(()))a Shish$^P$?

67 Yes.

Is
new Onion(
    new Tomato(
        new Skewer(())) )a Shish$^P$?

68 The object
new Tomato(
    new Skewer(()))
is an instance of Shish$^P$, so we can also wrap an Onion around it.

And how about another Tomato?  

69 Sure.
Is
new Tomato(
new Onion(
new Tomato(
new Skewer()))))
a vegetarian shish kebab?

70 Of course, there is no Lamb on it.

And
new Onion(
new Onion(
new Onion(
new Skewer())))?

71 Yes, it is a vegetarian shish kebab, because it only contains Onions.

Define the methods (i.e., the function)
is Vegetarian,
which return true if the given object does not contain Lamb.
Hint: The method for tomatoes is the same as the one for onions.

72 That's no big deal now.

<table>
<thead>
<tr>
<th>abstract boolean isVegetarian():</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shish</td>
</tr>
</tbody>
</table>

| boolean isVegetarian() {         |
| return true; }                  |
| Skewer                           |

| boolean isVegetarian() {         |
| return s.isVegetarian(); }       |
| Onion                            |

| boolean isVegetarian() {         |
| return false; }                  |
| Lamb                             |

| boolean isVegetarian() {         |
| return s.isVegetarian(); }       |
| Tomato                           |
How many methods have we defined?  

71 Five: one \texttt{abstract}, the others concrete.

Do \texttt{abstract} methods belong to the \texttt{abstract class}?  

74 Yes, they always do.

Does each variant of \texttt{Shish}\textsuperscript{P} contain a method called \texttt{isVegetarian}?  

75 Yes, because \texttt{Shish}\textsuperscript{P} contains an \texttt{abstract} method called \texttt{isVegetarian}.

Is this always the case?  

76 Always.

What do these concrete methods consume?  

77 Nothing, just as the \texttt{abstract} method says.

What do these concrete methods produce?  

78 \texttt{booleans}, just as the \texttt{abstract} method says.

\textbf{The Second Bit of Advice}

\begin{quote}
When writing a function over a datatype, place a method in each of the variants that make up the datatype. If a field of a variant belongs to the same datatype, the method may call the corresponding method of the field in computing the function.
\end{quote}
Collect all the pieces of Shish\(^D\). Here is the datatype.

```java
abstract class Shish\(^D\) {
    abstract boolean onlyOnions();
    abstract boolean isVegetarian();
}
```

There are two methods per variant.

```java
class Skewer extends Shish\(^D\) {
    boolean onlyOnions() {
        return true;
    }
    boolean isVegetarian() {
        return true;
    }
}
```

```java
class Onion extends Shish\(^D\) {
    Shish\(^D\) s;
    Onion(Shish\(^D\) s) {
        s = s;
    }
    boolean onlyOnions() {
        return s.onlyOnions();
    }
    boolean isVegetarian() {
        return s.isVegetarian();
    }
}
```

```java
class Lamb extends Shish\(^D\) {
    Shish\(^D\) s;
    Lamb(Shish\(^D\) s) {
        s = s;
    }
    boolean onlyOnions() {
        return false;
    }
    boolean isVegetarian() {
        return false;
    }
}
```

```java
class Tomato extends Shish\(^D\) {
    Shish\(^D\) s;
    Tomato(Shish\(^D\) s) {
        s = s;
    }
    boolean onlyOnions() {
        return false;
    }
    boolean isVegetarian() {
        return s.isVegetarian();
    }
}
```
What do the following define?

```
abstract class Kebab^D {
  Kebab
}
```

class Holder extends Kebab^D {
  Object o;
  Holder(Object o) {
    o = o;
  }
}

class Shallot extends Kebab^D {
  Kebab^D k;
  Shallot(Kebab^D k) {
    k = k;
  }
}

class Shrimp extends Kebab^D {
  Kebab^D k;
  Shrimp(Kebab^D k) {
    k = k;
  }
}

class Radish extends Kebab^D {
  Kebab^D k;
  Radish(Kebab^D k) {
    k = k;
  }
}

They define a datatype and four variants that are similar in shape to Shish^D.

Don’t forget the picture.

Chapter 2
What is different about them? They are placed onto different Holders.

Here are some holders.

abstract class Rod\textsuperscript{D} {}  

\texttt{class} Dagger\texttt{extends} Rod\textsuperscript{D} {}  

\texttt{class} Sabre\texttt{extends} Rod\textsuperscript{D} {}  

\texttt{class} Sword\texttt{extends} Rod\textsuperscript{D} {}  

Are they good ones?

Sure, a rod is a kind of holder, and every rod is an Object, so \texttt{o} in Holder can stand for any rod. Is it necessary to draw another picture?

Think of another kind of holder. Are you tired of drawing pictures, yet?

We could move all of the food to various forms of plates.

abstract class Plate\textsuperscript{D} {}  

\texttt{class} Gold\texttt{extends} Plate\textsuperscript{D} {}  

\texttt{class} Silver\texttt{extends} Plate\textsuperscript{D} {}  

\texttt{class} Brass\texttt{extends} Plate\textsuperscript{D} {}  

\texttt{class} Copper\texttt{extends} Plate\textsuperscript{D} {}  

\texttt{class} Wood\texttt{extends} Plate\textsuperscript{D} {}  

What is  
\begin{verbatim}
new Shallot(
    new Radish(
        new Holder(
            new Dagger()))))?
\end{verbatim}

It's a Kebab\textsuperscript{D}.  

Methods to Our Madness  

29
Is
  new Shallot(
    new Radish(
      new Holder(
        new dagger())))
a vegetarian Kebab^D?  

85 Sure it is. It only contains radishes and shallots.

Is
  new Shallot(
    new Radish(
      new Holder(
        new Gold())))
a Kebab^D?  

86 Sure, because Gold is a Plate^D. Plate^D is used as a Holder, and radishes and shallots can be put on any Holder.

87 Sure it is. It is basically like
    new Shallot(
      new Radish(
        new Holder(
          new Dagger()))), except that we have moved all the food from a Dagger to a Gold plate.

Let’s define the methods (i.e., the function) 88 If you can, you may rest now.
isVeggie.

which check whether a kebab contains only vegetarian foods, regardless of what Holder it is on.

Write the abstract method isVeggie. 89 That’s possible now.

abstract boolean isVeggie()  
Kebab

Of course, isVeggie belongs to Kebab^D and isVegetarian to Shish^D.
The concrete methods are similar to those called *Vegetarian*. Here are two more; define the remaining two.

```java
boolean isVeggie() {
    return true;
}
```

**Holder**

```java
boolean isVeggie() {
    return false;
}
```

**Shrimp**

```java
boolean isVeggie() {
    return k.isVeggie();
}
```

**Shallot**

```java
boolean isVeggie() {
    return k.isVeggie();
}
```

**Radish**

What is the value of

```java
new Shallot(
    new Radish(
        new Holder(
            new Dagger())))
  .isVeggie();
```

91 It is true.

What is

```java
new Shallot(
    new Radish(
        new Holder(
            new Dagger())));
```

92 It is a Kebab^D^, but we also know that it is an instance of the Shallot variant.

What is the value of

```java
new Shallot(
    new Radish(
        new Holder(
            new Gold())))
  .isVeggie();
```

93 It is true, too.

And what is

```java
new Shallot(
    new Radish(
        new Holder(
            new Gold())));
```

94 It is also a Kebab^D^, because any kind of Holder will do.
What type of value is
new Shallot(
    new Radish(
        new Holder(
            new Integer(52))))
.isVeggie()?

What type of value is
new Shallot(
    new Radish(
        new Holder(
            new OneMoreThan(
                new Zero()))))
.isVeggie()?

What type of value is
new Shallot(
    new Radish(
        new Holder(
            new Boolean(false))))
.isVeggie()?

Does that mean isVeggie works for all five kinds of Holders?

Yes, and all other kinds of Objects that we could possibly think of.

What is the holder of
new Shallot(
    new Radish(
        new Holder(
            new Dagger())))?

All the food is on a Dagger.

What is the holder of
new Shallot(
    new Radish(
        new Holder(
            new Gold())))?

All the food is now on a Gold plate.
What is the holder of new Shallot(
    new Radish(
        new Holder(
            new Integer(52))))?

What is the value of new Shallot(
    new Radish(
        new Holder(
            new Dagger())))
    .whatHolder()?  

102 The dagger.

What is the value of new Shallot(
    new Radish(
        new Holder(
            new Gold())))
    .whatHolder()?  

103 The gold plate.

What is the value of new Shallot(
    new Radish(
        new Holder(
            new Integer(52))))
    .whatHolder()?  

104 An Integer, whose underlying int is 52.

What type of values do the methods (i.e., the function) of whatHolder produce?

They produce rods, plates, and integers. And it looks like they can produce a lot more.

Is there a simple way of saying what type of values they produce?

They always produce an Object, which is also the type of the field of Holder.

Here is the abstract method whatHolder.

abstract Object whatHolder()

If we add this method to Kebab, then we must add a method definition to each of the four variants.
What is the value of
new Holder(
  new Integer(52))
  .whatHolder()?

108 new Integer(52).

What is the value of
new Holder(
  new Sword())
  .whatHolder()?

109 new Sword().

What is the value of
new Holder(b)
  .whatHolder()
if b is some object?

110 It is b.

Define the concrete method that goes into
the space labeled Holder.

111 With these kinds of hints, it's easy.

```
Object whatHolder() {
  return o; }
```

What is the value of
new Radish(
  new Shallot(
    new Shrimp(
      new Holder(
        new Integer(52))))))
  .whatHolder()?

112 new Integer(52).

What is the value of
new Shallot(
  new Shrimp(
    new Holder(
      new Integer(52))))
  .whatHolder()?

113 new Integer(52).
What is the value of
  new Shrimp(
      new Holder(
          new Integer(52)))
    .whatHolder();

114 new Integer(52).

Does that mean that the value of
  new Radish(
      new Shallot(
          new Shrimp(
              new Holder(
                  new Integer(52))))))
    .whatHolder();
is the same as
  new Shallot(
      new Shrimp(
          new Holder(
              new Integer(52))))
    .whatHolder(),
which is the same as
  new Shrimp(
      new Holder(
          new Integer(52)))
    .whatHolder(),
which is the same as
  new Holder(
      new Integer(52))
    .whatHolder();?

112 Yes, all four have the same answer:
  new Integer(52).

Here is the method for Shallot.

116 They are all the same.

Object whatHolder() {
    return k.whatHolder();
}

Shallot

Write the methods of whatHolder for Shrimp and Radish.

Object whatHolder() {
    return k.whatHolder();
}

Shrimp

Object whatHolder() {
    return k.whatHolder();
}

Radish
Here is the datatype and one of its variants. There are only three left.

```java
abstract class KebabD {
    abstract boolean isVeggie();
    abstract Object whatHolder();
}

class Shallot extends KebabD {
    KebabD k;
    Shallot(KebabD _k) {
        k = _k;
    }

    boolean isVeggie() {
        return k.isVeggie();
    }

    Object whatHolder() {
        return k.whatHolder();
    }
}

class Shrimp extends KebabD {
    KebabD k;
    Shrimp(KebabD _k) {
        k = _k;
    }

    boolean isVeggie() {
        return false;
    }

    Object whatHolder() {
        return k.whatHolder();
    }
}

class Radish extends KebabD {
    KebabD k;
    Radish(KebabD _k) {
        k = _k;
    }

    boolean isVeggie() {
        return k.isVeggie();
    }

    Object whatHolder() {
        return k.whatHolder();
    }
}
```

Collect the remaining variants.

Are there any other KebabD foods besides Shallot, Shrimp, and Radish? No, these are the only kinds of foods on a KebabD.
Can we add more foods?  

Let's define another Kebab\textsuperscript{D}.  

```
class Pepper extends Kebab\textsuperscript{D} {
    Kebab\textsuperscript{D} k;
    Pepper(Kebab\textsuperscript{D} .k) {
        k = .k;
    }

    boolean isVeggie() {
        return k.isVeggie();
    }
    Object whatHolder() {
        return k.whatHolder();
    }
}
```

Why does it include \texttt{isVeggie} and \texttt{whatHolder} methods?

A concrete class that extends Kebab\textsuperscript{D} must define these two methods. That's what the abstract specifications say. We can define as many Kebab\textsuperscript{D}s as we wish.

```
class Zucchini extends Kebab\textsuperscript{D} {
    Kebab\textsuperscript{D} k;
    Zucchini(Kebab\textsuperscript{D} .k) {
        k = .k;
    }

    boolean isVeggie() {
        return k.isVeggie();
    }
    Object whatHolder() {
        return k.whatHolder();
    }
}
```

Is it obvious how the new methods work?  

Totally. In both cases \texttt{isVeggie} just checks the rest of the Kebab\textsuperscript{D}, because green peppers and zucchini are vegetables. Similarly, \texttt{whatHolder} returns whatever holder belongs to the rest of the Kebab\textsuperscript{D}.

And then there were six.  

```
new ManhattanPt(3,4)
```

```
new ManhattanPt(1,5)
```

Which of these points is closer to the origin:  

Yes, now Kebab\textsuperscript{D} has six variants.

The second one, because its distance to the origin is 6 while the first point's distance is 7.

Good. Which of the following points is closer to the origin:  

The first one, clearly. Its distance to the origin is 5, but the second distance is 13.

```
new CartesianPt(3,4)
```

```
new CartesianPt(12,5)
```

Methods to Our Madness
We added the method `closerToO` to `CartesianPt`. It consumes another `CartesianPt` and determines whether the constructed or the consumed point is closer to the origin.

```java
class CartesianPt extends Point {
    int x;
    int y;
    CartesianPt(int _x, int _y) {
        x = _x;
        y = _y;
    }
    int distanceToO() {
        return \sqrt{x^2 + y^2};
    }
    boolean closerToO(CartesianPt p) {
        return distanceToO() <= p.distanceToO();
    }
}
```

Add the corresponding method to `ManhattanPt`.

```java
class ManhattanPt extends Point {
    int x;
    int y;
    ManhattanPt(int _x, int _y) {
        x = _x;
        y = _y;
    }
    int distanceToO() {
        return x + y;
    }
    boolean closerToO(ManhattanPt p) {
        return distanceToO() <= p.distanceToO();
    }
}
```

The definitions are nearly identical. The method for `ManhattanPt` consumes a `ManhattanPt` and determines which of those two points is closer to the origin.

---

What is the value of

```java
new ManhattanPt(3,4).
closerToO(new ManhattanPt(1,5))
```

1. This is the two character symbol \(\ll\).

What is the value of

```java
new ManhattanPt(1,5).
closerToO(new ManhattanPt(3,4))
```

1. true, obviously.

What is the value of

```java
new CartesianPt(12,5).
closerToO(new CartesianPt(3,4))
```

1. false, and true is the value of

```java
new CartesianPt(3,4).
closerToO(new CartesianPt(12,5)).
```

So finally, what is the value of

```java
new CartesianPt(3,4).
closerToO(new ManhattanPt(1,5))
```

1. That's nonsense.
The method closerToO can only consume CartesianPts, not ManhattanPts.

We could replace
\((\text{CartesianPt} \ p)\)
by
\((\text{Point}^D \ p)\)
in the definition of closerToO for CartesianPt.

Yes, because the definition of Point\(^D\) obliges every variant to provide a method named distanceToO.

Every CartesianPt is a Point\(^D\), and every ManhattanPt is a Point\(^D\), too.

Here is the improved CartesianPt.

```java
class CartesianPt extends Point\(^D\) {
    int x;
    int y;
    CartesianPt(int \_x, int \_y) {
        x = \_x;
        y = \_y;
    }

    int distanceToO() {
        return Math.sqrt(x*x + y*y);
    }

    boolean closerToO(Point\(^D\) \ p) {
        return distanceToO() \leq \ p.distanceToO();
    }
}
```

Improve the definition of ManhattanPt.

```java
class ManhattanPt extends Point\(^D\) {
    int x;
    int y;
    ManhattanPt(int \_x, int \_y) {
        x = \_x;
        y = \_y;
    }

    int distanceToO() {
        return x + y;
    }

    boolean closerToO(Point\(^D\) \ p) {
        return distanceToO() \leq \ p.distanceToO();
    }
}
```

Is the definition of closerToO in CartesianPt the same as the one in ManhattanPt?

Yes, they are identical.

Methods to Our Madness

39
Correct, and therefore we can add a copy to the abstract class \texttt{Point} and delete the definitions from the variants.

\begin{verbatim}
abstract class \texttt{Point} { 
    boolean \texttt{closerToO}(\texttt{Point} \texttt{p}) { 
        return 
        \texttt{distanceToO}() \leq \texttt{p.distanceToO}(); 
    } 

    abstract int \texttt{distanceToO}(); 
}
\end{verbatim}

Looks correct.

What else do the two point variants have in common?

Yes. It's tricky, but here is a start.

\begin{verbatim}
abstract class \texttt{Point} { 
    int \texttt{x};
    int \texttt{y};
    \texttt{Point}(int \texttt{x}, \texttt{int} \texttt{y}) { 
        \texttt{x} = \texttt{x};
        \texttt{y} = \texttt{y}; 
    }

    boolean \texttt{closerToO}(\texttt{Point} \texttt{p}) { 
        return 
        \texttt{distanceToO}() \leq \texttt{p.distanceToO}(); 
    } 

    abstract int \texttt{distanceToO}(); 
}
\end{verbatim}

Their fields. Shouldn't we lift them, too?

This not only lifts \texttt{x} and \texttt{y}, it also introduces a new constructor.

Absolutely. And we need to change how a \texttt{CartesianPt} is built. Define \texttt{ManhattanPt}.

\begin{verbatim}
class \texttt{CartesianPt extends Point} { 
    \texttt{CartesianPt}(\texttt{int} \texttt{x}, \texttt{int} \texttt{y}) { 
        \texttt{super}(\texttt{x}, \texttt{y}); 
    }

    int \texttt{distanceToO}() { 
        return \sqrt{\texttt{x}^2 + \texttt{y}^2}; 
    } 
}
\end{verbatim}

Mimicking this change is easy. But what does \texttt{super(x,y)} mean?

\begin{verbatim}
class \texttt{ManhattanPt extends Point} { 
    \texttt{ManhattanPt}(\texttt{int} \texttt{x}, \texttt{int} \texttt{y}) { 
        \texttt{super}(\texttt{x}, \texttt{y}); 
    }

    int \texttt{distanceToO}() { 
        return \texttt{x} + \texttt{y}; 
    } 
}
\end{verbatim}
The expressions `super(...) in the constructors `CartesianPt` and `ManhattanPt` create a `PointD` with the appropriate fields, and the respective constructor guarantees that the point becomes a `CartesianPt` or a `ManhattanPt`. 

That’s simple.

Do we now have everything that characterizes `PointD`s in the datatype?

Yes, and those things that distinguish the two variants from each other reside in the corresponding variants.

Is this a long chapter?

Yes, let’s have a short snack break.
3. What's New?
abstract class Pizza\(^D\) {
    Pizza
}

class Crust extends Pizza\(^D\) {
    Crust
}

class Cheese extends Pizza\(^D\) {
    Pizza\(^D\) p;
    Cheese(Pizza\(^D\) p) {
        p = .p;
    }
    Cheese
}

class Olive extends Pizza\(^D\) {
    Pizza\(^D\) p;
    Olive(Pizza\(^D\) p) {
        p = .p;
    }
    Olive
}

class Anchovy extends Pizza\(^D\) {
    Pizza\(^D\) p;
    Anchovy(Pizza\(^D\) p) {
        p = .p;
    }
    Anchovy
}

1. Looks like good toppings. Let’s add Sausage.

class Sausage extends Pizza\(^D\) {
    Pizza\(^D\) p;
    Sausage(Pizza\(^D\) p) {
        p = .p;
    }
    Sausage
}

What’s New?
Here is our favorite pizza:

\[
\text{new Anchovy}(
\text{new Olive}(
\text{new Anchovy}(
\text{new Anchovy}(
\text{new Cheese}(
\text{new Crust}()))))).
\]

\[\text{This looks too salty.}\]

How about removing the anchovies?

\[\text{That would make it less salty.}\]

Let's remove them. What is the value of

\[
\text{new Anchovy}(
\text{new Olive}(
\text{new Anchovy}(
\text{new Anchovy}(
\text{new Cheese}(
\text{new Crust}()))))).
\]

\[\text{It should be a cheese and olive pizza, like this:}\]

\[
\text{new Olive}(
\text{new Cheese}(
\text{new Crust}()))).
\]

\[\text{A better name for these methods would be \text{removeAnchovy}.}\]

\[\text{but then our definitions wouldn't fit into these columns.}\]

What is the value of

\[
\text{new Sausage}(
\text{new Olive}(
\text{new Anchovy}(
\text{new Sausage}(
\text{new Cheese}(
\text{new Crust}()))))).
\]

\[\text{It should be a cheese, sausage, and olive pizza, like this:}\]

\[
\text{new Sausage}(
\text{new Olive}(
\text{new Sausage}(
\text{new Cheese}(
\text{new Crust}())))).
\]

\[\text{Does \text{newA} belong to the datatype \text{PizzaD} and its variants?}\]

\[\text{Yes, and it produces them, too.}\]
Define the methods that belong to the five variants. Here is a start.

\[
\text{abstract Pizza}^D \text{ remA}();
\]

Pizza

\[
\text{Pizza}^D \text{ remA}() \{
\text{return new Crust}();
\}
\]

Crust

Define the two methods that belong to Olive and Sausage. We’ve eaten the cheese already.

\[
\text{Pizza}^D \text{ remA}() \{
\text{return new Cheese}(p.\text{remA}());
\}
\]

Cheese

\[
\text{Pizza}^D \text{ remA}() \{
\text{return new Olive}(p.\text{remA}());
\}
\]

Olive

\[
\text{Pizza}^D \text{ remA}() \{
\text{return new Sausage}(p.\text{remA}());
\}
\]

Sausage

The Olive and Sausage methods are similar to the Cheese method.

Explain why we use new Cheese . . . .
new Olive . . . ., and new Sausage . . . .
when we define these methods.

The methods remA must produce a Pizza\(^D\), so let’s use new Crust(), the simplest Pizza\(^D\), for the method in Anchovy.

\[
\text{Pizza}^D \text{ remA}() \{
\text{return new Crust}();
\}
\]

Anchovy

\[
\text{Pizza}^D \text{ remA}() \{
\text{return new Olive}(p.\text{remA}());
\}
\]

Olive

\[
\text{Pizza}^D \text{ remA}() \{
\text{return new Sausage}(p.\text{remA}());
\}
\]

Sausage

The cheese, the olives, and the sausages on the pizzas must be put back on top of the pizza that \(p.\text{remA}()\) produces.

Yes, and now the methods of \(\text{remA}\) produce pizzas without any anchovies on them.

What’s New? 45
Let's try it out on a small pizza:

```
new Anchovy(
    new Crust()))
.remove().
```

That's easy. The object is an Anchovy. So the answer is `new Crust()`.

Is

```
new Crust()
```

like

```
new Anchovy(
    new Crust()))
```

without anchovy?

Absolutely, but what if we had more anchovies?

No problem. Here is an example:

```
new Anchovy(
    new Anchovy(
        new Crust()))
.remove().
```

That's easy again. As before, the object is an Anchovy so that the answer must still be `new Crust()`.

Okay, so what if we had an olive and cheese on top:

```
new Olive(
    new Cheese(
        new Anchovy(
            new Anchovy(
                new Crust())))))
.remove().
```

Well, this object is an Olive and its p stands for

```
new Cheese(
    new Anchovy(
        new Anchovy(
            new Crust()))).
```

Then, what is the value of

```
new Olive(p, remove())
```

where p stands for

```
new Cheese(
    new Anchovy(
        new Anchovy(
            new Crust()))).
```

It is the pizza that

```
new Cheese(
    new Anchovy(
        new Anchovy(
            new Crust()))).
.remove().
```

produces, with an olive added on top.
What is the value of
   new Cheese(
   new Anchovy(
   new Anchovy(
   new Crust())))
   .remA()?

16 It is
   new Cheese(p.remA()),
   where p stands for
   new Anchovy(
   new Anchovy(
   new Crust()))).

And what is the value of
   new Cheese(
   new Anchovy(
   new Anchovy(
   new Crust())))
   .remA()?

17 It is the pizza that
   new Anchovy(
   new Anchovy(
   new Crust())))
   .remA() produces, with cheese added on top.

Do we know the value of
   new Anchovy(
   new Anchovy(
   new Crust())))
   .remA()?

18 Yes, we know that it produces new Crust().

Does that mean that new Crust() is the
   answer?

19 No, we still have to add cheese and an olive.

Does it matter in which order we add those
   two toppings?

20 Yes, we must first add cheese, producing
   new Cheese(
   new Crust())
   and then we add the olive.

So what is the final answer?

21 It is
   new Olive(
   new Cheese(
   new Crust())).
Let's try one more example:

```java
new Cheese()
    new Anchovy(
        new Cheese(
            new Crust()))
    .remA();
```

What kind of pizza should this make?

---

Check it out!

The object is an instance of `Cheese` so the value is

```java
new Cheese(p.remA());
```

where `p` stands for

```java
new Anchovy(
    new Cheese(
        new Crust()));
```

---

Doesn't that mean that the result is

```java
new Cheese(
    new Anchovy(
        new Cheese(
            new Crust()))
    .remA());
```

---

What about

```java
new Anchovy(
    new Cheese(
        new Crust()))
    .remA();
```

---

And the answer is

```java
new Crust();
```

---

Does that mean the final answer is

```java
new Cheese(
    new Crust());
```

---

Chapter 3
What do we want? A double-cheese pizza like
   new Cheese(
       new Cheese(
           new Crust()))),
   because that's what it means to remove anchovies and nothing else.

Which remA method do we need to change to get the cheese back? The one in Anchovy.

PizzaP remA() {
    return p.remA();
} Anchovy

Does this remA still belong to PizzaP? Yes, and it still produces them.

The Third Bit of Advice
When writing a function that returns values of a datatype, use new to create these values.

We could add cheese on top of the anchovies. Yes, that would hide their taste, too.

What kind of pizza is easy, it adds cheese on top of each anchovy:
   new Olive(
       new Anchovy(
           new Cheese(
               new Anchovy(
                   new Crust()))))

   .topAnchovyC1()?

1 A better name for these methods would be topAnchovyWithCheese.

Did you notice the underlines? Yes, they show where we added cheese.

What's New? 49
And what is
new Olive(
    new Cheese(
        new Sausage(
            new Crust())))
    .topAwC();}

34 Here we don’t add any cheese, because the
pizza does not contain any anchovies:
    new Olive(
        new Cheese(
            new Sausage(
                new Crust()))).

Define the remaining methods.

35 We expect you to know some of the answers.

abstract PizzaD topAwC();

Pizza

PizzaD topAwC() {
    return new Cheese(p.topAwC()); }
    Cheese

Crust

PizzaD topAwC() {
    return new Olive(p.topAwC()); } Olive

Sausage

Take a look at this method.

PizzaD topAwC() {
    return p.topAwC(); } Anchovy

36 With that definition, topAwC would give the
same results as remA. The method topAwC
in Anchovy must put the anchovy back on
the pizza and top it with cheese:

PizzaD topAwC() {
    return
    new Cheese(
        new Anchovy(p.topAwC())); } Anchovy
How many layers of cheese are in the result of

(new Olive(
    new Anchovy(
    new Cheese(
    new Anchovy(
    new Crust()))))
).remA() 
.topAwC()?

One, because remA removes all the anchovies, so that topAwC doesn’t add any cheese.

How many occurrences of cheese are in the result of

(new Olive(
    new Anchovy(
    new Cheese(
    new Anchovy(
    new Crust()))))
).topAwC() .remA()?

Three, because topAwC first adds cheese for each anchovy. Then remA removes all the anchovies:

(new Olive(
    new Cheese(
    new Cheese(
    new Crust())))).

Perhaps we should replace every anchovy with cheese.

We just did something like that.

Is it true that for each anchovy in x 

x.topAwC().remA()

adds some cheese?

Yes, and it does more. Once all the cheese is added, the anchovies are removed.

So 

x.topAwC().remA()

is a way to substitute all anchovies with cheese by looking at each topping of a pizza and adding cheese on top of each anchovy and then looking at each topping again, including all the new cheese, and removing the anchovies.

Aha!
Here is a different description:
“The methods look at each topping of a pizza and replace each anchovy with cheese.”

Define the methods that match this description. Call them $\textit{subAbC}$.

Here is the abstract method.

\begin{verbatim}
abstract Pizza $\textit{subAbC}()$:

Pizza
\end{verbatim}

Here is a skeleton.

\begin{verbatim}
Pizza $\textit{subAbC}()$
|\textit{return new Crust();} |
\hline
Crust

Pizza $\textit{subAbC}()$
|\textit{return new Cheese($p$.\textit{subAbC}());} |
\hline
Cheese

Pizza $\textit{subAbC}()$
|\textit{return new Olive($p$.\textit{subAbC}());} |
\hline
Olive

Pizza $\textit{subAbC}()$
|\textit{return \_\_\_\_\_\_;} |
\hline
Anchovy

Pizza $\textit{subAbC}()$
|\textit{return new Sausage($p$.\textit{subAbC}());} |
\hline
Sausage
\end{verbatim}

\footnote{A better name for these methods would be \texttt{substituteAnchovyByCheese}.}

Does this skeleton look familiar?

\begin{itemize}
\item Yes, this skeleton looks just like those of $\textit{topAwC}$ and $\textit{remA}$.
\end{itemize}

Define the method that belongs in $\textit{Anchovy}$.

\begin{verbatim}
Pizza $\textit{subAbC}()$
|\textit{return new Cheese($p$.\textit{subAbC}());} |
\hline
Anchovy
\end{verbatim}

\footnote{Here it is.}
Collection time.\textsuperscript{1}

\begin{verbatim}
abstract class Pizza\textsuperscript{D} {
    abstract Pizza\textsuperscript{D} remA();
    abstract Pizza\textsuperscript{D} topAuC();
    abstract Pizza\textsuperscript{D} subAbC();
}

class Crust extends Pizza\textsuperscript{D} {
    Pizza\textsuperscript{D} remA() {
        return new Crust();}
    Pizza\textsuperscript{D} topAuC() {
        return new Crust();
    }
    Pizza\textsuperscript{D} subAbC() {
        return new Crust();
    }
}

class Cheese extends Pizza\textsuperscript{D} {
    Pizza\textsuperscript{D} p;
    Cheese(Pizza\textsuperscript{D} .p) {
        p = .p;
    }
    Pizza\textsuperscript{D} remA() {
        return new Cheese(p.remA());
    }
    Pizza\textsuperscript{D} topAuC() {
        return new Cheese(p.topAuC());
    }
    Pizza\textsuperscript{D} subAbC() {
        return new Cheese(p.subAbC());
    }
}

class Sausage extends Pizza\textsuperscript{D} {
    Pizza\textsuperscript{D} p;
    Sausage(Pizza\textsuperscript{D} .p) {
        p = .p;
    }
    Pizza\textsuperscript{D} remA() {
        return new Sausage(p.remA());
    }
    Pizza\textsuperscript{D} topAuC() {
        return new Sausage(p.topAuC());
    }
    Pizza\textsuperscript{D} subAbC() {
        return new Sausage(p.subAbC());
    }
}

class Olive extends Pizza\textsuperscript{D} {
    Pizza\textsuperscript{D} p;
    Olive(Pizza\textsuperscript{D} .p) {
        p = .p;
    }
    Pizza\textsuperscript{D} remA() {
        return new Olive(p.remA());
    }
    Pizza\textsuperscript{D} topAuC() {
        return new Olive(p.topAuC());
    }
    Pizza\textsuperscript{D} subAbC() {
        return new Olive(p.subAbC());
    }
}

class Anchovy extends Pizza\textsuperscript{D} {
    Pizza\textsuperscript{D} p;
    Anchovy(Pizza\textsuperscript{D} .p) {
        p = .p;
    }
    Pizza\textsuperscript{D} remA() {
        return p.remA();
    }
    Pizza\textsuperscript{D} topAuC() {
        return new Anchovy(p.topAuC());
    }
    Pizza\textsuperscript{D} subAbC() {
        return new Anchovy(p.subAbC());
    }
}
\end{verbatim}

\textsuperscript{1} This is similar to the interpreter and composite patterns [4].

The classes are getting larger.

\begin{verbatim}
class Olive extends Pizza\textsuperscript{D} {
    Pizza\textsuperscript{D} p;
    Olive(Pizza\textsuperscript{D} .p) {
        p = .p;
    }
    Pizza\textsuperscript{D} remA() {
        return new Olive(p.remA());
    }
    Pizza\textsuperscript{D} topAuC() {
        return new Olive(p.topAuC());
    }
    Pizza\textsuperscript{D} subAbC() {
        return new Olive(p.subAbC());
    }
}
\end{verbatim}

\begin{verbatim}
class Anchovy extends Pizza\textsuperscript{D} {
    Pizza\textsuperscript{D} p;
    Anchovy(Pizza\textsuperscript{D} .p) {
        p = .p;
    }
    Pizza\textsuperscript{D} remA() {
        return p.remA();
    }
    Pizza\textsuperscript{D} topAuC() {
        return new Anchovy(p.topAuC());
    }
    Pizza\textsuperscript{D} subAbC() {
        return new Anchovy(p.subAbC());
    }
}
\end{verbatim}

\begin{verbatim}
class Sausage extends Pizza\textsuperscript{D} {
    Pizza\textsuperscript{D} p;
    Sausage(Pizza\textsuperscript{D} .p) {
        p = .p;
    }
    Pizza\textsuperscript{D} remA() {
        return new Sausage(p.remA());
    }
    Pizza\textsuperscript{D} topAuC() {
        return new Sausage(p.topAuC());
    }
    Pizza\textsuperscript{D} subAbC() {
        return new Sausage(p.subAbC());
    }
}
\end{verbatim}

\begin{verbatim}
class Olive extends Pizza\textsuperscript{D} {
    Pizza\textsuperscript{D} p;
    Olive(Pizza\textsuperscript{D} .p) {
        p = .p;
    }
    Pizza\textsuperscript{D} remA() {
        return new Olive(p.remA());
    }
    Pizza\textsuperscript{D} topAuC() {
        return new Olive(p.topAuC());
    }
    Pizza\textsuperscript{D} subAbC() {
        return new Olive(p.subAbC());
    }
}
\end{verbatim}

\begin{verbatim}
class Anchovy extends Pizza\textsuperscript{D} {
    Pizza\textsuperscript{D} p;
    Anchovy(Pizza\textsuperscript{D} .p) {
        p = .p;
    }
    Pizza\textsuperscript{D} remA() {
        return p.remA();
    }
    Pizza\textsuperscript{D} topAuC() {
        return new Anchovy(p.topAuC());
    }
    Pizza\textsuperscript{D} subAbC() {
        return new Anchovy(p.subAbC());
    }
}
\end{verbatim}

\begin{verbatim}
class Sausage extends Pizza\textsuperscript{D} {
    Pizza\textsuperscript{D} p;
    Sausage(Pizza\textsuperscript{D} .p) {
        p = .p;
    }
    Pizza\textsuperscript{D} remA() {
        return new Sausage(p.remA());
    }
    Pizza\textsuperscript{D} topAuC() {
        return new Sausage(p.topAuC());
    }
    Pizza\textsuperscript{D} subAbC() {
        return new Sausage(p.subAbC());
    }
}
\end{verbatim}

What's New?  53
Let’s add more Pizza$^D$ foods.

Here is one addition: Spinach.

```java
class Spinach extends Pizza$^D$ {
    Pizza$^D$ p;
    Spinach(Pizza$^D$ p) {
        p = p;
    }
    Pizza$^D$ remA() {
        return new Spinach(p, remA());
    }
    Pizza$^D$ topAwC() {
        return new Spinach(p, topAwC());
    }
    Pizza$^D$ subAbC() {
        return new Spinach(p, subAbC());
    }
}
```

Yes, we must define three concrete methods for each variant of Pizza$^D$.

Do we need to change Pizza$^D$, Crust, Cheese, Olive, Anchovy, or Sausage?

No. When we add variants to the datatypes we have defined, we don’t need to change what we have.

Isn’t that neat?

Yes, this is a really flexible way of defining classes and methods. Unfortunately, the more things we want to do with Pizza$^D$’s, the more methods we must add.

True enough. And that means cluttered classes. Is there a better way to express all this?

That would be great, because these definitions are painful to the eye. But we don’t know of a better way to organize these definitions yet.

Don’t worry. We are about to discover how to make more sense out of such things.

Great.

And now you can replace anchovy with whatever pizza topping you want.

We will stick with anchovies.

Chapter 3
4.
Come to Our Carousel
Wasn’t this last collection overwhelming?  

1. It sure was. We defined seven classes and each contained three method definitions.

Could it get worse?  

2. It sure could. For everything we want to do with PizzaD, we must add a method definition to each class.

Why does that become overwhelming?  

3. Because it becomes more and more difficult to understand the rationale for each of the methods in a variant and what the relationship is between methods of the same name in the different variants.

Correct. Let’s do something about it. Take a close look at this visitor class.

```
class OnlyOnionsV {
    boolean forSkewer() {
        return true;
    }
    boolean forOnion(ShishD s) {
        return s.onlyOnions();
    }
    boolean forLamb(ShishD s) {
        return false;
    }
    boolean forTomato(ShishD s) {
        return false;
    }
}
```

4. These methods look familiar. Have we seen them before?

\[ V \] This superscript is a reminder that the class is a visitor class. Lower superscripts when you enter this kind of definition in a file: OnlyOnionsV.

Almost. Each of them corresponds to an onlyOnions method in one of the variants of ShishD.

5. That’s right. The major difference is that they are all in one class, a visitor, whose name is OnlyOnionsV.

Is onlyOnions different from OnlyOnionsV?  

6. The former is used to name methods, the latter names a class.

Come to Our Carousel
And that’s the whole point.  

What point?

We want all the methods in one class.  

What methods?

These methods that would have the same name if we placed them into the variants of a datatype in one class.  

If we could do that, it would be much easier to understand what action these methods perform.

That’s what we are about to do. We are going to separate the action from the datatype.

It’s about time.

What is the difference between the method onlyOnions in the Onion variant and the method forOnion in the visitor OnlyOnionsV?

Everything following return is the same.

Right. What is the difference?

The difference is that onlyOnions in Onion is followed by () and that forOnion in OnlyOnionsV is followed by (ShishV s).

Yes, that is the difference. Are the other for methods in OnlyOnionsV related to their counterparts in the same way?

Indeed, they are.

It is time to discuss the boring part.

What boring part?

The boring part tells us how to make all of this work.

True, we still don’t know how to make ShishV and its variants work with this visitor class, which contains all the action.
Now take a look at this.

```java
abstract class ShishD {
    OnlyOnionsV ooFn = new OnlyOnionsV();
    abstract boolean onlyOnions();
}
```

```java
class Skewer extends ShishD {
    boolean onlyOnions() {
        return ooFn.forSkewer();
    }
}
```

```java
class Onion extends ShishD {
    ShishD s;
    Onion(ShishD.s) {
        s = s;
    }
    boolean onlyOnions() {
        return ooFn.forOnion(s);
    }
}
```

```java
class Lamb extends ShishD {
    ShishD s;
    Lamb(ShishD.s) {
        s = s;
    }
    boolean onlyOnions() {
        return ooFn.forLamb(s);
    }
}
```

```java
class Tomato extends ShishD {
    ShishD s;
    Tomato(ShishD.s) {
        s = s;
    }
    boolean onlyOnions() {
        return ooFn.forTomato(s);
    }
}
```

This is a strange set of definitions. All the `onlyOnions` methods in the variants look alike. Each of them uses an instance of `OnlyOnionsV`, which is created in the datatype, to invoke a `for` method with a matching name.
What does the `forOnion` method in `Onion` consume?  

That's what "consumption" is all about. And what does the `forSkewer` method in `Skewer` consume?  

So what does the `(Shish^P s)` mean in the definition of `forOnion`?  

Very good. The notation `(Shish^P s)` means that `forOnion` consumes a `Shish^P` and that between `{ and `}, `s` stands for that shish.  

That makes sense and explains `s.onlyOnions()`.  

Here are our words: "`s` is a `Shish^P`, and therefore `s.onlyOnions()` determines whether what is below the onion is also edible by an onion lover."  

You knew we wouldn't let you down: "`ooFn.forOnion` says that we want to use the method we just described. It consumes a `Shish^P`, and `s` is the `Shish^P` that represents what is below the onion."  

It is still true.  

We start with the `onlyOnions` method in `Onion`, but it immediately uses the `forOnion` method on the rest of the shish.
And what does the forOnion method do? It checks whether the rest of this shish is edible by onion lovers.

How does it do that? It uses the method onlyOnions on s.

Isn’t that where we started from? Yes, we’re going round and round.

Welcome to the carousel. Fortunately, the shish shrinks as it goes around, and when we get to the skewer we stop.

And then the ride is over and we know that for this example the answer is true. That’s exactly it.

Do we need to remember that we are on a carousel? No! Now that we understand how the separation of data and action works, we only need to look at the action part to understand how things work.

Is one example enough? No, let’s look at some of the other actions on shishes and pizzas.

Let’s look at isVegetarian next. Here is the beginning of the protocol.¹

```java
abstract class Shish {  
    OnlyOnions ooFn = new OnlyOnions();  
    isVegetarian ivFn = new isVegetarian();  
    abstract boolean onlyOnions();  
    abstract boolean isVegetarian();  
}
```

¹ The American Heritage Dictionary defines protocol as "the form of ceremony and etiquette observed by diplomats and heads of state." For us, a protocol is an agreement on how classes that specify a datatype and its variants interact with classes that realize functions on that datatype.

Come to Our Carousel
Write the rest! We must add two lines to each variant, and they are almost the same as those for `ooFn`.

```java
class Skewer extends ShishD {
    boolean onlyOnions() {
        return ooFn.forSkewer();
    }
    boolean isVegetarian() {
        return ivFn.forSkewer();
    }
}

class Onion extends ShishD {
    ShishD s;
    Onion(ShishD _s) {
        s = _s;
    }
    boolean onlyOnions() {
        return ooFn.forOnion(s);
    }
    boolean isVegetarian() {
        return ivFn.forOnion(s);
    }
}

class Lamb extends ShishD {
    ShishD s;
    Lamb(ShishD _s) {
        s = _s;
    }
    boolean onlyOnions() {
        return ooFn.forLamb(s);
    }
    boolean isVegetarian() {
        return ivFn.forLamb(s);
    }
}

class Tomato extends ShishD {
    ShishD s;
    Tomato(ShishD _s) {
        s = _s;
    }
    boolean onlyOnions() {
        return ooFn.forTomato(s);
    }
    boolean isVegetarian() {
        return ivFn.forTomato(s);
    }
}
```
That’s why we call this part boring. True, there’s very little to think about. It could be done automatically.

How do we define the visitor? Does that refer to the class that contains the actions?

Yes, that one. Define the visitor. It is like OnlyOnions\(^Y\) except for the method forTomato.

```java
class isVegetarian\(^Y\) {
    boolean forSkeuer() {
        return true;
    }
    boolean forOnion(Shish\(^P\) s) {
        return s.isVegetarian();
    }
    boolean forLamb(Shish\(^P\) s) {
        return false;
    }
    boolean forTomato(Shish\(^P\) s) {
        return s.isVegetarian();
    }
}
```

Are we moving fast? Yes, but there are only a few interesting parts. The protocol is always the same and boring; the visitor is always closely related to what we saw in chapter 2.

How about a tea break? Instead of coffee?

The Fourth Bit of Advice

*When writing several functions for the same self-referential datatype, use visitor protocols so that all methods for a function can be found in a single class.*

Come to Our Carousel 63
Is

new Anchovy()
new Olive()
new Anchovy()
new Cheese()
new Crust(())

a shish kebab?

39. No, it's a pizza.

abstract class PizzaD { }

class Crust extends PizzaD { }

class Cheese extends PizzaD {
  PizzaD p;
  Cheese(PizzaD .p) {
    p = .p;
  }
}

class Olive extends PizzaD {
  PizzaD p;
  Olive(PizzaD .p) {
    p = .p;
  }
}

class Anchovy extends PizzaD {
  PizzaD p;
  Anchovy(PizzaD .p) {
    p = .p;
  }
}

class Sausage extends PizzaD {
  PizzaD p;
  Sausage(PizzaD .p) {
    p = .p;
  }
}

40. We can define the protocol for the methods that belong to PizzaD and its extensions: remA, topAweC, and subAbC.
Great! Here is the abstract portion of the protocol.

```
abstract class PizzaD {
    RemA^V remFn = new RemA^V();
    TopAwC^V topFn = new TopAwC^V();
    SubAbC^V subFn = new SubAbC^V();
    abstract PizzaD^ remA();
    abstract PizzaD^ topAwC();
    abstract PizzaD^ subAbC();
}
```

And here are some variants.

```
class Crust extends PizzaD {
    PizzaD^ remA() {
        return remFn.forCrust();
    }
    PizzaD^ topAwC() {
        return topFn.forCrust();
    }
    PizzaD^ subAbC() {
        return subFn.forCrust();
    }
}
```

```
class Cheese extends PizzaD {
    PizzaD^ p;
    Cheese(PizzaD^ .p) {
        p = .p;
    }
    PizzaD^ remA() {
        return remFn.forCheese(p); }
    PizzaD^ topAwC() {
        return topFn.forCheese(p); }
    PizzaD^ subAbC() {
        return subFn.forCheese(p); }
}
```

Define the rest.

How innovative! The variants are totally mindless, now.

```
class Olive extends PizzaD {
    PizzaD^ p;
    Olive(PizzaD^ .p) {
        p = .p;
    }
    PizzaD^ remA() {
        return remFn.forOlive(p); }
    PizzaD^ topAwC() {
        return topFn.forOlive(p); }
    PizzaD^ subAbC() {
        return subFn.forOlive(p); }
}
```

```
class Anchovy extends PizzaD {
    PizzaD^ p;
    Anchovy(PizzaD^ .p) {
        p = .p;
    }
    PizzaD^ remA() {
        return remFn.forAnchovy(p); }
    PizzaD^ topAwC() {
        return topFn.forAnchovy(p); }
    PizzaD^ subAbC() {
        return subFn.forAnchovy(p); }
}
```

```
class Sausage extends PizzaD {
    PizzaD^ p;
    Sausage(PizzaD^ .p) {
        p = .p;
    }
    PizzaD^ remA() {
        return remFn.forSausage(p); }
    PizzaD^ topAwC() {
        return topFn.forSausage(p); }
    PizzaD^ subAbC() {
        return subFn.forSausage(p); }
}
```

Come to Our Carousel
We are all set.

Okay, here is RemA^V

```java
class RemA^V {
    Pizza^D forCrust() {
        return new Crust();
    }
    Pizza^D forCheese(Pizza^D p) {
        return new Cheese(p, remA());
    }
    Pizza^D forOlive(Pizza^D p) {
        return new Olive(p, remA());
    }
    Pizza^D forAnchoy(Pizza^D p) {
        return p, remA();
    }
    Pizza^D forSausage(Pizza^D p) {
        return new Sausage(p, remA());
    }
}
```

Define TopAwC^V.

By now, even this is routine.

```java
class TopAwC^V {
    Pizza^D forCrust() {
        return new Crust();
    }
    Pizza^D forCheese(Pizza^D p) {
        return new Cheese(p, topAwC());
    }
    Pizza^D forOlive(Pizza^D p) {
        return new Olive(p, topAwC());
    }
    Pizza^D forAnchoy(Pizza^D p) {
        return new Cheese(p, topAwC());
    }
    Pizza^D forSausage(Pizza^D p) {
        return new Sausage(p, topAwC());
    }
}
```

The last one, SubAbC^V, is a piece of cake.

And we thought we were working with a pizza pie.

```java
class SubAbC^V {
    Pizza^D forCrust() {
        return new Crust();
    }
    Pizza^D forCheese(Pizza^D p) {
        return new Cheese(p, subAbC());
    }
    Pizza^D forOlive(Pizza^D p) {
        return new Olive(p, subAbC());
    }
    Pizza^D forAnchoy(Pizza^D p) {
        return new Cheese(p, subAbC());
    }
    Pizza^D forSausage(Pizza^D p) {
        return new Sausage(p, subAbC());
    }
}
```
5. Objects Are People, Too
Have we seen this kind of definition before?  

abstract class Pie\textsuperscript{D} { 
  Pie
} 

class Bot extends Pie\textsuperscript{D} { 
  Bot
} 

class Top extends Pie\textsuperscript{D} { 
  Object \_t; 
  Pie\textsuperscript{D} \_r; 
  Top(Object \_t, Pie\textsuperscript{D} \_r) { 
    \_t = \_t; 
    \_r = \_r; 
  } 
} 

1 Better names for these classes would be Pizza\textsuperscript{Pie\textsuperscript{D}}, Bottom and Topping, respectively.

Yes, still more pizza, but this one is different.  

Yes, it includes only one variant for adding toppings to a pizza, and toppings are Objects.

What kind of toppings can we put on these kinds of pizza?  

Any kind, because Object is the class of all objects. Here are some fish toppings.

abstract class Fish\textsuperscript{D} {}

class Anchovy extends Fish\textsuperscript{D} {}

class Salmon extends Fish\textsuperscript{D} {}

class Tuna extends Fish\textsuperscript{D} {}
Nice datatype. Is
new Top(new Anchovy()).
new Top(new Tuna()).
new Top(new Anchovy()).
new Bot(()))
a pizza pie?

It is a pizza pie, and so is
new Top(new Tuna()).
new Top(new Integer(42)).
new Top(new Anchovy()).
new Top(new Integer(5)).
new Bot(()))

What is the value of
new Top(new Salmon()).
new Top(new Anchovy()).
new Top(new Tuna()).
new Top(new Anchovy()).
new Bot(()))

It is this fishy pizza pie:
new Top(new Salmon()).
new Top(new Tuna()).
new Bot(()).

Is it true that the value of
new Top(new Salmon()).
new Top(new Tuna()).
new Bot(()))

Yes. The pizza that comes out is the same as
the one that goes in, because there are no
anchovies on that pizza.

Does remA belong to Pie^D?

Yes, and it produces pizza pies.

Define the protocol for RemA^V. We provide
the abstract part.

RemA^V remFu = new RemA^V();
abstract Pie^D remA();

Pie
remA()

Pie
remA()

This is easy by now.

Pie^D remA() {
return remFu.forBot(); }

Bot
remA()

Pie^D remA() {
return remFu.forTop(t,x); }

Top
Great. Isn’t that easy? 

Easy and boring.

What part of this exercise differs from datatype to datatype?

10 Determining how many fields a variant contains. In our case, we had zero and two.

Anything else?

11 No, from that we know that $raFn.forBot$ is followed by () and $raFn.forTop$ by $(t,r)$.

Why $(t,r)$?

12 Because these are the fields of Top.

Let’s define the visitor $RemA^V$.

```java
class RemA^V {
    Pie^P forBot() {
        return _____; }
    Pie^P forTop(Object t,Pie^P r) {
        if (new Anchovy().equals(t))
            return _____;
        else
            return _____; }
    }
```

13 Here are some guesses.

```java
class RemA^V {
    Pie^P forBot() {
        return new Bot(); }
    Pie^P forTop(Object t,Pie^P r) {
        if (new Anchovy().equals(t))
            return r.remA();
        else
            return new Top(t.r.remA()); }
    }
```

Great guesses! What does

if $(expr_1)$
    return $expr_2$;
else
    return $expr_3$;
mean?

14 We guess:

“This produces the value of either $expr_2$ or $expr_3$, depending on whether or not $expr_1$ is determined to be true or false, respectively.”

And what does

$new Anchovy().equals(t)$
mean?

15 We could guess:

“This expression determines whether $t$ is equal to $new Anchovy().$”

Not yet. It depends on what equals means.

16 What?

*Objects Are People, Too*
What is the value of

\[ \text{new Anchovy().equals(new Anchovy())?} \]

The "Not yet." implies that the value is false.

---

Yes! And what is the value of

\[ \text{new Anchovy().equals(new Tuna())?} \]

false. Because no anchovy is a tuna.

---

The class Object contains a method called equals. This method compares one Object to another, and it always returns false.\(^1\)

\(^1\) Not always. We explain the correct answer in chapter 10.

---

We must define it anew\(^1\) for all classes whose instances we wish to compare.

\(^1\) In Java, redefining a method is called "overriding."

---

For Fish\(^D\) and its variants it works like this.

```java
abstract class Fish\(^D\) {}  
  
  class Anchovy extends Fish\(^D\) {
    public boolean equals(Object o) {
      return (o instanceof Anchovy);
    }
  }

  class Salmon extends Fish\(^D\) {
    public boolean equals(Object o) {
      return (o instanceof Salmon);
    }
  }

  class Tuna extends Fish\(^D\) {
    public boolean equals(Object o) {
      return (o instanceof Tuna);
    }
  }
```

---

Assuming that

\[ (o \text{ instanceof Tuna}) \]

is true when o is an instance of Tuna, these method definitions are obvious.

\(^1\) The class Object is defined in a separate package called java.lang.Object. Overriding methods that reside in other packages requires the word public.
Aren’t they? Is every value constructed with `new` an instance of `Object`?

Yes. Every such value is an `Object`, because every class extends `Object` directly or indirectly.

If class `A` extends `B`, is every value created by `new A(...)` an instance of class `B`?

Yes, and of the class that `B` extends and so on.

Now, what is the value of

`new Anchovy().equals(new Anchovy())`?

24 true,

because `new Anchovy()` is an instance of `Anchovy`.

Yet the value of

`new Anchovy().equals(new Tuna())`

is still false.

25 Of course, because an anchovy is never a tuna.

Could we have written `RemA^\nu` without using `equals`?

Absolutely, `instanceof` is enough.

```java
class RemA^\nu {
    Pie^D forBot() {
        return new Bot();
    }
    Pie^D forTop(Object t,Pie^D r) {
        if (t instanceof Anchovy) return r.remA();
        else return new Top(t,r.remA());
    }
}
```

Why haven’t we defined it this way?

Easy, because we want to generalize `RemA^\nu` so that it works for any kind of fish topping.

27 We can do that, but when we use the methods of the more general visitor, we need to say which kind of fish we want to remove.

What are good names for the more general methods and visitor?

28 How about `remFish` and `RemFish^\nu`?

`Objects Are People, Too` 73
How do we use \texttt{remFish}? We give it a \texttt{Fish}. 

Add the protocol for \texttt{RemFish}$. We designed the abstract portion.

\begin{verbatim}
RemFish$ rffn = new RemFish$();
abstract Pie$ remFish(Fish$ f);
\end{verbatim}

The rest is routine.

\begin{verbatim}
Pie$ remFish(Fish$ f) {
return rffn.forBot(f);
}
\end{verbatim}

\begin{verbatim}
Pie$ remFish(Fish$ f) {
return rffn.forTop(t,r,f);
}
\end{verbatim}

Where do \textit{(f)} and \textit{(t,r,f)} come from? The \textit{f} stands for the \texttt{Fish} we want to remove in both cases. The \textit{t} and the \textit{r} are the fields of \texttt{Top}; \texttt{Bot} doesn’t have any.

Let’s define \texttt{RemFish}$ and its two methods. Instead of comparing the top layer \textit{t} of the pizza to \texttt{Anchovy}, we now determine whether it equals the \texttt{Fish}, which is the additional value consumed by the method.

\begin{verbatim}
class RemFish$ {
Pie$ forBot(Fish$ f) {
return new Bot();
}
Pie$ forTop(Object t,Pie$ r,Fish$ f) {
if (f.equals(t))
return r.remFish(f);
else
return new Top(t,r,remFish(f));
}
}
\end{verbatim}

If we add another kind of fish to our datatype, what would happen to the definition of \texttt{RemFish}$? Nothing, we just have to remember to add \texttt{equals} to the new variant.
Let's try it out with a short example:

```java
new Top(new Anchovy(),
new Bot())
.remFish(new Anchovy());
```

34 The object is a topping, so we use `forTop` from `RemFish<V>.

Yes. What values does `forTop` consume? 35 It consumes three values: `new Anchovy()`, which is `t`, the top-most layer of the pizza; `new Bot()`, which is `r`, the rest of the pizza; and `new Anchovy()`, which is `f`, the Fish to be removed.

And now? 36 Now we need to determine the value of

```java
if (f.equals(t))
    return r.remFish(f);
else
    return new Top(t,r.remFish(f));
```

where `t`, `r`, and `f` stand for the values just mentioned.

So? 37 Given what `f` and `t` stand for, `f.equals(t)` is true. Hence, we must determine the value of `r.remFish(f)`.

What is the value of `new Bot()` 38 It is the same as `forBot(f)`, where `f` is `new Anchovy()`.

`.remFish(new Anchovy())?`

What does `forBot` in `RemFish<V>` produce? 39 It produces `new Bot()`, no matter what `f` is.

All clear? 40 Ready to move on, after snack time.

*Objects Are People, Too*
Does

```plaintext
new Top(new Integer(2).
new Top(new Integer(3).
new Top(new Integer(2).
new Bot(1)))
```

... look familiar?

---

What does `remInt` do?

'Yes, it looks like what we just evaluated.

---

It removes `Integer` from pizza pies just as `remFish` removes fish from pizza pies.

---

Who defined `equals` for `Integer`?

'`The Machine decided

```plaintext
new Integer(0). equals(new Integer(0))
```

to be true, and the rest was obvious.'

---

Define the visitor `RemInt`.

'Wonderful! We do the interesting thing first. This visitor is almost identical to `RemFish`.

We just need to change the type of what the two methods consume.'

```plaintext
class RemInt {
    Pie for Bot(Integer i) {
        return new Bot(); }
    Pie for Top(Object t, Pie r, Integer i) {
        if (t.equals(i))
            return r.remInt(t);
        else
            return new Top(t.r.remInt(i));
    }
}
```

---

Does it matter that this definition uses `i` and not `f`?

'No, `i` is just a better name than `f`, no other reason. As long as we do such substitutions systematically, we are just fine.'

---

Where is the protocol?

'It is so simple, let's save it for later.'

---

76
Can we remove Integers from Pie's?

Yes.

Can we remove Fish from Pie's?

Yes, and we use nearly identical definitions.

Let's combine the two definitions.

If we use Object instead of the underlined Integer above, everything works out.

Why?

Because everything constructed with new is an Object.

Just do it!

It's done.

```java
class RemV {
    Pie forBot(Object o) {
        return new Bot();
    }
    Pie forTop(Object t, Pie r, Object o) {
        if (o.equals(t))
            return r.rem(o);
        else
            return new Top(t, r.rem(o));
    }
}
```

Should we do the protocol for all these visitors?

Now?

You never know when it might be useful, even if it does not contain any interesting information.

Let's just consider RemV.

Why not RemFishV and RemAV and RemIntV?

They are unnecessary once we have RemV.

Objects Are People, Too
Here is the abstract portion of PieP.

```java
abstract class PieP {
    RemV remV = new RemV();
    abstract PieP rem(Object o);
}
```

And here are the pieces for Bot and Top.

```java
class Bot extends PieP {
    PieP rem(Object o) {
        return remV.forBot(o); }
}

class Top extends PieP {
    Object t;
    PieP r;
    Top(Object ,t,PieP ,r) {
        t = ,t;
        r = ,r; }
    PieP rem(Object o) {
        return remV.forTop(t,r,o); }
}
```

Let’s remove some things from pizza pies:

```java
new Top(new Integer(2),
new Top(new Integer(3),
new Top(new Integer(2),
new Bot())))
.rem(new Integer(3)).
```

Works like a charm with the same result as before.

And how about

```java
new Top(new Anchovy(),
new Bot())
.rem(new Anchovy());?
```

Ditto.

Next:

```java
new Top(new Anchovy(),
new Top(new Integer(3),
new Top(new Zero(),
new Bot())))
.rem(new Integer(3)).
```

No problem. This, too, removes 3 and leaves the other layers alone:

```java
new Top(new Anchovy(),
new Top(new Zero(),
new Bot()));
```
What is the value of
new Top(new Anchovy(),
new Top(new Integer(3),
new Top(new Zero(),
new Bot()))).
.rem(new Zero())?

Oops. The answer is
new Top(new Anchovy(),
new Top(new Integer(3),
new Top(new Zero(),
new Bot()))).

What's wrong with that?

We expected it to remove new Zero() from the pizza.

And why didn't it?

Because equals for NumD's uses Object's equals, which always produces false — as we discussed above when we introduced equals.

Always?

Unless we define it anew for those classes whose instances we wish to compare.

Here is the version of NumD (including OneMoreThan) with its own equals. Define the new Zero variant.

abstract class NumD {
}

class OneMoreThan extends NumD {
    NumD predecessor;
    OneMoreThan(NumD p) {
        predecessor = p;
    }

    public boolean equals(Object o) {
        if (o instanceof OneMoreThan)
            return predecessor.equals((OneMoreThan)o).predecessor;
        else
            return false;
    }
}

Adding equals to Zero is easy. We use instanceof to determine whether the consumed value is a new Zero().

class Zero extends NumD {
    public boolean equals(Object o) {
        return (o instanceof Zero);
    }
}

But what is the underlining of ((OneMoreThan)o) about? Wouldn't it have been sufficient to write o.predecessor?

1 In Java, this is called (downward) casting, because OneMoreThan extends NumD.

Objects Are People, Too 79
No. What is the type of \( o \)?

64 Object, according to \((\text{Object } o)\), which is what declares the type of \( o \).

So what is \( o \).\textit{predecessor}?

65 Nonsense.

Correct. What do we know after \texttt{if} has determined that 

66 We know that \( o \)'s type is \texttt{Object} and that it is an instance of \texttt{OneMoreThan}.

\( o \) \texttt{instanceof OneMoreThan} 

is \texttt{true}?

Precisely. So what does \(((\texttt{OneMoreThan})o)\) do?

67 It converts the type of \( o \) from \texttt{Object} to \texttt{OneMoreThan}.

What is \(((\texttt{OneMoreThan}) o)\)'s type?

68 Its type is \texttt{OneMoreThan}, and now it makes sense to write 

\(((\texttt{OneMoreThan}) o).\textit{predecessor}\).

Are \( o \) and \(((\texttt{OneMoreThan}) o)\) interchangeable?

69 The underlying object is the same. But no, the two expressions are not interchangeable, because the former's type is \texttt{Object}, whereas the latter's is \texttt{OneMoreThan}.

Is this complicated?

70 Someone has been drinking too much coffee.

Did you also notice the 

\texttt{predecessor}

\texttt{.equals}(

\(((\texttt{OneMoreThan}) o).\texttt{predecessor}\)

in \texttt{equals} for \texttt{OneMoreThan}? 

71 How do the two uses of \texttt{predecessor} differ?

The first one, \texttt{predecessor}, refers to the \texttt{predecessor} field of the instance of \texttt{OneMoreThan} on which we are using \texttt{equals}. And that field might not be a \texttt{OneMoreThan}.

72 So the second one, \(((\texttt{OneMoreThan}) o).\texttt{predecessor}\), refers to the \texttt{predecessor} field of the instance of \texttt{OneMoreThan} consumed by \texttt{equals}.

80 Chapter 5
Yes. Are these two objects equal? 73 If they are similar\(^1\) to the same \texttt{int}, they are equal. But most of the time, they are not.

\(^{1}\) Check chapter \texttt{1} for "similar."

Time for lunch? 74 That's just in time.

Did you have a good lunch break? 75 Yes, thank you.

Now what is the value of 
\begin{verbatim}
new Top(new Anchovy(),
new Top(new Integer(3),
new Top(new Zero(),
new Bot())))
.rem(new Zero())?
\end{verbatim}

76 Now we get 
\begin{verbatim}
new Top(new Anchovy(),
new Top(new Integer(3),
new Bot()))
\end{verbatim}
which is precisely what we want.

And why? 77 Because \texttt{equals} now knows how to compare \texttt{Num}D's.

Do we always add \texttt{equals} to a class? 78 No, only if we need it.

Do we need \texttt{equals} when we want to substitute one item for another on a pizza pie? 79 Yes, we do.

What is the value of 
\begin{verbatim}
new Top(new Anchovy(),
new Top(new Tuna(),
new Top(new Anchovy(),
new Bot())))
.substFish(new Salmon(),
new Anchovy())?
\end{verbatim}

80 It is the same pizza pie with all the anchovies replaced by salmon: 
\begin{verbatim}
new Top(new Salmon(),
new Top(new Tuna(),
new Top(new Salmon(),
new Bot())))
\end{verbatim}

What kind of values does \texttt{substFish} consume? 81 It consumes two \texttt{fish} and works on \texttt{Pie}D's.

\textit{Objects Are People, Too}
And what does it produce? It always produces a Pie^D.

What is the value of new Top(new Integer(3).
new Top(new Integer(2).
new Top(new Integer(3).
new Bot()));)
.substInt(new Integer(5).
new Integer(3));

It is the same pizza pie with all 3s replaced by 5s:
new Top(new Integer(5).
new Top(new Integer(2).
new Top(new Integer(5).
new Bot()));)

What kind of values does substInt consume? It consumes two integers and works on Pie^D's.

And what does it produce? It always produces a Pie^D.

We can define SubstFish^V.

```java
class SubstFish^V {
    Pie^D forBot(Fish^D n,Fish^D o) {
        return new Bot();
    }
    Pie^D forTop(Object t,
    Pie^D r,
    Fish^D n,
    Fish^D o) {
        if (o.equals(t))
            return new Top(t,r,substFish(n,o));
        else
            return new Top(t,r,substFish(n,o));
    }
}
```

Define SubstInt^V.

Did we forget the boring parts? Yes, because there is obviously a more general version like Rem^V.

To get from SubstFish^V to SubstInt^V, we just need to substitute Fish^D by Integer everywhere and "Fish" by "Int" in the class and method names.

```java
class SubstInt^V {
    Pie^D forBot(Integer n,Integer o) {
        return new Bot();
    }
    Pie^D forTop(Object t,
    Pie^D r,
    Integer n,
    Integer o) {
        if (o.equals(t))
            return new Top(t,r,substInt(n,o));
        else
            return new Top(t,r,substInt(n,o));
    }
}
```
Yes, we call it \( \text{Subst}^V \). Define it.

\[
\text{class Subst}^V \{
\text{Pie}^P \text{ rem}(\text{Object } n, \text{Object } o) \{ \\
\text{return new } \text{Bot}(); \}
\text{Pie}^P \text{ forBot}(\text{Object } t). \\
\text{Pie}^P \text{ r,} \\
\text{Object } n, \\
\text{Object } o) \{ \\
\text{if (o.equals(t))} \\
\text{return new } \text{Top}(n,r,\text{subst}(n,o)); \\
\text{else} \\
\text{return new } \text{Top}(t,r,\text{subst}(n,o)); \}
\}
\]

Now it is time to add the protocol for \( \text{Subst}^V \) to \( \text{Pie}^P \). Here are the variants.

\[
\text{class Bot extends Pie}^P \{ \\
\text{Pie}^P \text{ rem}(\text{Object } o) \{ \\
\text{return remFn,forBot(o);} \}
\text{Pie}^P \text{ subst}(\text{Object } n, \text{Object } o) \{ \\
\text{return substFn,forBot(n,o);} \}
\}
\]

\[
\text{class Top extends Pie}^P \{ \\
\text{Object } t; \\
\text{Pie}^P \text{ r;} \\
\text{Top( Object } .t, \text{Pie}^P \text{ .r) \{ \\
\text{t = .t;} \\
\text{r = .r;} \}
\text{Pie}^P \text{ rem}(\text{Object } o) \{ \\
\text{return remFn,forTop(t,r,o);} \}
\text{Pie}^P \text{ subst}(\text{Object } n, \text{Object } o) \{ \\
\text{return substFn,forTop(t,r,n,o);} \}
\}
\]

So?

\[
\text{abstract class Pie}^P \{ \\
\text{Rem}^V \text{ remFn = new Rem}^V(); \\
\text{Subst}^V \text{ substFn = new Subst}^V(); \\
\text{abstract Pie}^P \text{ rem}(\text{Object } o); \\
\text{abstract Pie}^P \text{ subst}(\text{Object } n, \text{Object } o); \\
\}
\]

That was some heavy lifting.
Are protocols truly boring?  

But, of course they are not. We just didn’t want to spend much time on them. Let’s take a closer look at the last one we defined in the previous chapter.

```
abstract class PieD {
    Rem^V remFn = new Rem^V();
    Subst^V substFn = new Subst^V();
    abstract PieD rem(Object o);
    abstract PieD subst(Object n,Object o);
}
```

Okay, here are the variants again.

```
class Bot extends PieD {
    PieD rem(Object o) {
        return remFn.forBot(o);
    }
    PieD subst(Object n,Object o) {
        return substFn.forBot(n,o);
    }
}
```

```
class Top extends PieD {
    Object t;
    PieD r;
    Top(Object t,PieD r) {
        t = .t;
        r = .r;
    }
    PieD rem(Object o) {
        return remFn.forTop(t,r,o);
    }
    PieD subst(Object n,Object o) {
        return substFn.forTop(t,r,n,o);
    }
}
```

What is the difference between rem and subst in PieD?  

The first one consumes one Object, the second one consumes two.

What is the difference between rem and subst in the Bot variant?  

Simple: rem asks for the forBot service from remFn and hands over the Object it consumes; subst asks for the forBot service from substFn and hands over the two Objects it consumes.

What is the difference between rem and subst in the Top variant?  

Simpler: rem asks for the forTop service from remFn and hands over the field values and the Object it consumes; subst asks for the forTop service from substFn and hands over the field values and the two Objects it consumes.
And that is all there is to the methods in the variants of a protocol.

But remFn and substFn defined in the datatype are still a bit mysterious.

Let’s not create remFn and substFn in the datatype.

```
abstract class PieD {
    abstract PieD rem(RemV remFn, Object o);
    abstract PieD subst(SubstV substFn, Object n, Object o);
}
```

This looks like an obvious modification. The new rem and subst now consume a remFn and a substFn, respectively. Can they still find forBot and forTop, their corresponding carousel partners?

Yes, it is a straightforward trade-off. Instead of adding a remFn field and a substFn field to the datatype, we now have rem or subst consume such values. What kind of values are consumed by rem and subst?

The definition of the datatype says that they are a RemV and a SubstV, respectively. And every RemV defines forBot and forTop, and so does every SubstV.

Here is how it changes Top.

```
class Top extends PieD {
    Object t;
    PieD r;
    Top(Object t, PieD r) {
        t = t;
        r = r;
    }
    PieD rem(RemV remFn, Object o) {
        return remFn.forTop(t, r, o);
    }
    PieD subst(SubstV substFn, Object n, Object o) {
        return substFn.forTop(t, r, n, o);
    }
}
```

In the same manner. We just need to change each concrete method’s description of what it consumes. The rest remains the same.

```
class Bot extends PieD {
    PieD rem(RemV remFn, Object o) {
        return remFn.forBot(o);
    }
    PieD subst(SubstV substFn, Object n, Object o) {
        return substFn.forBot(n, o);
    }
}
```

How does it affect Bot?
That's right. Nothing else changes in the variants. Instead of relying on fields of the datatype, we use what is consumed.

Like what?

Where are they used?

Yes. Here is $\text{Rem}^\triangledown$.

```java
class Rem$\triangledown$ {
    Pie$\triangledown$ forBot(Object o) {
        return new Bot();
    }
    Pie$\triangledown$ forTop(Object t,
        Pie$\triangledown$ r,
        Object o) {
        if (o.equals(t))
            return r.rem(this,o);
        else
            return new Top(t,r.rem(this,o));
    }
}
```

Modify $\text{Subst}^\triangledown$ accordingly.

```
class Subst$\triangledown$ {
    Pie$\triangledown$ forBot(Object n,
        Object o) {
        return new Bot();
    }
    Pie$\triangledown$ forTop(Object t,
        Pie$\triangledown$ r,
        Object n,
        Object o) {
        if (o.equals(t))
            return new Top(n,r.subst(this,n,o));
        else
            return new Top(t,r.subst(this,n,o));
    }
}
```

That takes all the fun out of it.

What is this all about?

Yes, what about it. Copying is easy.

Understanding is more difficult. The word this refers to the object itself.

Which object?

How did we get here?

The protocol is that $\text{rem}$ in $\text{Bot}$ and $\text{Top}$ asks for the $\text{forBot}$ and $\text{forTop}$ methods of $\text{remFn}$, respectively.

*Boring Protocols*
How does that happen?

17 It happens with

\( \text{remFn, forBot}(...) \)

and

\( \text{remFn, forTop}(...) \),

respectively.

Correct. And now \textit{forBot} and \textit{forTop} can refer to the object \textit{remFn} as \textbf{this}.

18 Oh, so inside the methods of \textit{Rem} \( ^{V} \), \textbf{this} stands for precisely that instance of \textit{Rem} \( ^{V} \) that allowed us to use those methods in the first place. And that must mean that when we use \( r \text{.rem}(\textbf{this}, ...) \) in \textit{forTop}, it tells \textit{rem} to use the same instance over again.

That’s it. Tricky?  

19 Not really, just self-referential.

Why?

20 Because \textbf{this} is a \textit{Rem} \( ^{V} \), and it is exactly what we need to complete the job.

What is the value of  

new \text{Top}(new \text{Anchovy}());

new \text{Top}(new \text{Integer}(3),

new \text{Top}(new \text{Zero}(),

new \text{Bot}()));

.\text{rem}(new \text{Rem} \( ^{V} \),

new \text{Zero}());

21 We did the same example in the preceding chapter, and the result remains the same.

And how does the underlined part relate to what we did there?

22 It creates a \textit{Rem} \( ^{V} \) object, which corresponds to the \textit{remFn} in the old \textit{Pie} \( ^{D} \).

What is the value of  

new \text{Top}(new \text{Integer}(3),

new \text{Top}(new \text{Integer}(2),

new \text{Top}(new \text{Integer}(3),

new \text{Bot}()));

.\text{subst}(new \text{Subst} \( ^{V} \),

new \text{Integer}(5),

new \text{Integer}(3));

23 We did the same example in the preceding chapter, and the result remains the same.
And how does the underlined part relate to what we did there? 24 It creates a Subst\textsuperscript{V} object, which corresponds to the remFn in the old Pie\textsuperscript{D}.

So what is the underlined part about? 25 We changed the methods in Pie\textsuperscript{D}, which means that we must also change how it is used.

Ready for the next protocol? 26 Let’s grab a quick snack.

How about some ice cream? 27 Cappuccino crunch sounds great. The more coffee, the better.

Take a look at subst in Top and at forTop in Subst\textsuperscript{V}. What happens to the values that they consume? 28 Nothing really. They get handed back and forth, though forTop compares o to t.

Is the handing back and forth necessary? 29 We don’t know any better way, yet.

Here is a way to define Subst\textsuperscript{V} that avoids the handing back and forth of these extra values.

```java
class Subst\textsuperscript{V} {
    Object n;
    Object o;
    Subst\textsuperscript{V}(Object _n, Object _o) {
        n = _n;
        o = _o;
    }
}
```

Pie\textsuperscript{D} forBot() {
    return new Bot();
}

Pie\textsuperscript{D} forTop(Object t, Pie\textsuperscript{D} r) {
    if (o.equals(t))
        return new Top(n, r.subst(this));
    else
        return new Top(t, r.subst(this));
}
```

1 In functional programming, a visitor with fields is called a closure (or a higher-order function), which would be the result of applying a curried version of subst.
How do we create a Subst<sup>V</sup>?  

We use  
\[
\text{new Subst<sup>V</sup>(new Integer(5), new Integer(3))}. 
\]

What does that do?  

It creates a Subst<sup>V</sup> whose methods know how to substitute new Integer(5) for all occurrences of new Integer(3) in Pie<sup>V</sup>.

How do the methods know that without consuming more values?  

The values have now become fields of the Subst<sup>V</sup> object to which the methods belong. They no longer need to be consumed.

Okay, so how would we substitute all new Integer(3) with new Integer(5) in 
\[
\text{new Top(new Integer(3), new Integer(2), new Top(new Integer(3), new Bot()))}? 
\]

We write  
\[
\text{new Top(new Integer(3), new Integer(2), new Top(new Integer(3), new Bot()))}. 
\]

\[
\text{.subst(new Subst<sup>V</sup>(new Integer(5), new Integer(3)))}. 
\]

And if we want to substitute all new Integer(2) with new Integer(7) in the same pie?  

We write  
\[
\text{new Top(new Integer(3), new Top(new Integer(2), new Top(new Integer(3), new Bot())))}. 
\]

\[
\text{.subst(new Subst<sup>V</sup>(new integer(7), new Integer(2)))}. 
\]

Does all that mean we have to change the protocol, too?  

Of course, because the methods subst in the Bot and Top variants consume only one value now.
That's right. Here are the datatype and its Bot variant. Define the Top variant.

```java
abstract class PieD {
    abstract PieD rem(remV remFn);
    abstract PieD subst(substV substFn);
}
```

```java
class Bot extends PieD {
    PieD rem(remV remFn) {
        return remFn.forBot();
    }
    PieD subst(substV substFn) {
        return substFn.forBot();
    }
}
```

```java
class Top extends PieD {
    Object t;
    PieD r;
    Top(Object .t, PieD .r) {
        t = .t;
        r = .r;
    }
    PieD rem(remV remFn) {
        return remFn.forTop(t, r);
    }
    PieD subst(substV substFn) {
        return substFn.forTop(t, r);
    }
}
```

Is there anything else missing?

We haven't defined RemV for this new protocol. But it is simple and hardly worth our attention.

What is the difference between rem and subst in Bot?

Not much. The name of the respective values they consume and the corresponding types.

What is the difference between rem and subst in Top?

Not much. The name of the respective values they consume and the corresponding types.

Can we eliminate the differences?

It is easy to make them use the same names. It doesn't matter whether rem is defined as it is or as:

```java
PieD rem(remV substFn) {
    return substFn.forTop(t, r);
}
```

True, because substFn is just a name for a value we don't know yet. But how can we make the types the same?

Both RemV and SubstV are visitors that contain the same method names and those methods consume and produce the same types of values. We can think of them as extensions of a common abstract class.

_Boring Protocols_ 91
Yes! Do it!

Here it is.

```java
abstract class PieVisitorD {
    abstract PieD forBot();
    abstract PieD forTop(Object t, PieD r);
}
```

Great job, except that we will use interface for specifying visitors like these.

```java
interface PieVisitorD {
    PieD forBot();
    PieD forTop(Object t, PieD r);
}
```

I This superscript is a reminder that the name refers to an interface. Lower superscripts when you enter this kind of definition in a file: PieVisitorI.

Okay, that doesn't seem to be a great difference. Can a class extend an interface the way it extends an abstract class?

No. A class implements an interface; it does not extend it.

Now that we have an interface that describes the type of the values consumed by rem and subst, can we make their definitions even more similar?

Yes, we can. Assuming we can use interfaces like abstract classes, we can write

```java
PieD rem(PieVisitorD pvFn) {
    return pvFn.forTop(t, r);
}
```

and

```java
PieD subst(PieVisitorD pvFn) {
    return pvFn.forTop(t, r);
}
```

in Top.

Correct. What is the difference between rem and subst, now?

There isn't any. We can use the same name for both, as long as we remember to use it whenever we would have used rem or subst.

What is a good name for this method?

The method accepts a visitor and asks for its services, so we call it accept.
And what is a better name for `peFn`?

Now we can simplify the protocol. Here is the new `RemV`.

```java
class RemV implements PieVisitorT {
    Object o;
    RemV(Object _o) {
        _o = o;
    }

    public PieP forBot() {
        return new Bot();
    }
    public PieP forTop(Object t,PieP r) {
        if (_o.equals(t))
            return r.accept(this);
        else
            return new Top(t,r.accept(this));
    }
}
```

Supply the protocol.

Did you notice the two underlined occurrences of `public`?

Yes, what about them?

When we define a `class` that implements an `interface`, we need to add the word `public` to the left of the method definitions.

Why?

It’s a way to say that these are the methods that satisfy the obligations imposed by the `interface`.

Looks weird, but let’s move on.

Correct. They are just icing.

Okay, we still won’t forget them.

_Boring Protocols_
Now define the new $\text{Subst}^V$.

55 Here it is.

```java
class $\text{Subst}^V$ implements $\text{PieVisitor}^T$
{
    Object $n$;
    Object $o$;
    $\text{Subst}^V$(Object $_.n$, Object $_.o$) {
        $n$ = $_.n$;
        $o$ = $_.o$;
    }

    public $\text{Pie}^D$ $\text{forBot}()$
    {
        return new $\text{Bot}()$;
    }

    public $\text{Pie}^D$ $\text{forTop}(\text{Object } t, \text{Pie}^D r)$
    {
        if ($o$.equals($t$))
            return
            new $\text{Top}(n, r$.accept($this$));
        else
            return
            new $\text{Top}(t, r$.accept($this$));
    }
}
```

56 Here is our picture.

Draw a picture of the interface $\text{PieVisitor}^T$ and all the classes: $\text{Pie}^D$, Bot, Top, $\text{Rem}^V$, and $\text{Subst}^V$.

Why is there a line, not an arrow, from $\text{Subst}^V$ to $\text{PieVisitor}^T$?

57 The $\text{Subst}^V$ visitor implements $\text{PieVisitor}^T$.

58 It tells us the name of the method that connects the datatype to the visitors.

And the dashed line?

57 The $\text{Subst}^V$ visitor implements $\text{PieVisitor}^T$.

It doesn't extend it. Arrows mean "extends," lines mean "implements."

Chapter 6
What is the value of
\[
\text{new Top(new Anchovy(),}
\text{new Top(new Tuna(),}
\text{new Top(new Anchovy(),}
\text{new Top(new Tuna(),}
\text{new Top(new Anchovy(),}
\text{new Bot()))))}
\]
\[
...\text{.accept(new LtdSubst}^V\text{(2,}
\text{new Salmon(),}
\text{new Anchovy()))?)}
\]

Easy:
\[
\text{new Top(new Salmon(),}
\text{new Top(new Tuna(),}
\text{new Top(new Anchovy(),}
\text{new Top(new Tuna(),}
\text{new Top(new Anchovy(),}
\text{new Bot()))))}
\]

Explain what LtdSubst" produces.\(^1\)

\(^1\) A better name is LimitedSubstitution, and that is how we pronounce it.

The methods of LtdSubst" replace one fish on a pie by another as many times as specified by the first value consumed by LtdSubst".

Good. Define LtdSubst".

That's easy. We have such a flexible protocol that we only need to define the essence now.

```java
class LtdSubst" implements PieVisitor\(^T\) {
    int c;
    Object n;
    Object o;
    LtdSubst"(int .c, Object .n, Object .o) {
        c = .c;
        n = .n;
        o = .o;
    }
    
    public Pie\(^D\) forBot() {
        return new Bot();
    }
    public Pie\(^D\) forTop(Object t, Pie\(^D\) r) {
        if (c == 0)
            return new Top(t, r);
        else
            if (o.equals(t))
                return new Top(n, r.accept(this));
            else
                return new Top(t, r.accept(this));
    }
}
```

---

Boring Protocols 95
What is the value of
new Top(new Anchovy()),
   new Top(new Tuna()),
   new Top(new Anchovy()),
   new Top(new Anchovy(),
new Bot()));
   .accept(new LtdSubst^V(2,
   new Salmon(),
   new Anchovy()));

Oops, there are too few anchovies on this pizza pie:
new Top(new Salmon()),
new Top(new Tuna()),
new Top(new Anchovy(),
new Top(new Salmon(),
new Bot()));

How come?

Because e, the counting field, never changes.

Why doesn’t e ever change?

Because this, the LtdSubst^V that performs the substitutions, never changes.

Can we fix this?

We can’t change this, but we can replace this with a new LtdSubst^V that reflects the change.

If e stands for the current count, how do we create a LtdSubst^V that shows that we have just substituted one fish by another.

Simple, we use
new LtdSubst^V(c - 1,n,o)
in place of this.

The Sixth Bit of Advice

When the additional consumed values change for a self-referential use of a visitor, don’t forget to create a new visitor.
Define the new and improved version of `LstSubstV`.

Voilà.

```java
class LstSubstV implements PieVisitorV {
    int c;
    Object n;
    Object o;
    LstSubstV(int _c, Object _n, Object _o) {
        c = _c;
        n = _n;
        o = _o;
    }

    public PieV forBot() {
        return new Bot();
    }

    public PieV forTop(Object t, PieV r) {
        if (c == 0)
            return new Top(t, r);
        else
            if (o.equals(t))
                return new Top(n,
                               r.accept(
                               new LstSubstV(c - 1, n, o)));
            else
                return new Top(t,
                               r.accept(
                               this));
    }
}
```

How does this differ from `new LstSubstV(c - 1, n, o)`?

They are two different `LstSubstV`'s. One replaces `c` occurrences of `o` by `n` in a pizza pie, and the other one replaces only `c - 1` of them.

How do you feel about protocols now?

They are exciting. Let’s do more.

---

*Boring Protocols*
7. Oh My!
Is
   new Flat(new Apple(),
   new Flat(new Peach(),
   new Bud()))
a flat Tree^D?  

1. Yes.

Is
   new Flat(new Pear(),
   new Bud())
a flat Tree^D?  

2. Yes, it is also a flat Tree^D.

And how about
   new Split(
   new Bud(),
   new Flat(new Fig(),
   new Split(
   new Bud(),
   new Bud()))))?

3. No, it is split, so it can’t be flat.

Here is one more example:
   new Split(
   new Split(
   new Bud(),
   new Flat(new Lemon(),
   new Bud()))),
   new Flat(new Fig(),
   new Split(
   new Bud(),
   new Bud()))).

4. No, it isn’t flat either.

Is it flat?

5. Unless there is anything else to Tree^D, it’s totally clear.

Is the difference between flat trees and split trees obvious now?

6. Okay, let’s.

Good. Then let’s move on.
Here are some fruits.

```java
abstract class FruitD {
}

class Peach extends FruitD {
    public boolean equals(Object o) {
        return (o instanceof Peach);
    }
}

class Apple extends FruitD {
    public boolean equals(Object o) {
        return (o instanceof Apple);
    }
}

class Pear extends FruitD {
    public boolean equals(Object o) {
        return (o instanceof Pear);
    }
}

class Lemon extends FruitD {
    public boolean equals(Object o) {
        return (o instanceof Lemon);
    }
}

class Fig extends FruitD {
    public boolean equals(Object o) {
        return (o instanceof Fig);
    }
}
```

It does not differ too much from what we have seen before.

```java
abstract class TreeD {
}

class Bud extends TreeD {
}

class Flat extends TreeD {
    FruitD f;
    TreeD t;
    Flat(FruitD.f, TreeD.t) {
        f = .f;
        t = .t;
    }
}

class Split extends TreeD {
    TreeD l;
    TreeD r;
    Split(TreeD.l, TreeD.r) {
        l = .l;
        r = .r;
    }
}
```

Let's say all TreeD's are either flat, split, or bud. Formulate a rigorous description for TreeD's.

Did you notice that we have redefined the method equals in the variants of FruitD? *That probably means that we will need to compare fruits and other things.*

Do TreeD's variants contain equals? *No, which means we won't compare them, but we could.*
How does the datatype $\text{Tree}^D$ differ from all the other datatypes we have seen before?

Let’s add a visitor interface whose methods produce booleans.

```java
interface bTreeVisitor^T {
    boolean forBud();
    boolean forFlat(Fruit^D f, Tree^D t);
    boolean forSplit(Tree^D l, Tree^D r);
}
```

Here is the new datatype definition.

```java
abstract class Tree^D {
    abstract boolean accept(bTreeVisitor^T ask);
}
```

Revise the variants.

```java
class Bud extends Tree^D {
    boolean accept(bTreeVisitor^T ask) {
        return ask.forBud();
    }
}
```

```java
class Flat extends Tree^D {
    Fruit^D f;
    Tree^D t;
    Flat(Fruit^D f, Tree^D t) {
        f = f;
        t = t;
    }
    boolean accept(bTreeVisitor^T ask) {
        return ask.forFlat(f, t);
    }
}
```

```java
class Split extends Tree^D {
    Tree^D l;
    Tree^D r;
    Split(Tree^D l, Tree^D r) {
        l = l;
        r = r;
    }
    boolean accept(bTreeVisitor^T ask) {
        return ask.forSplit(l, r);
    }
}
```

That just means extending what we have with one method each.

But isn’t $\text{bTreeVisitor}^T$ a pretty unusual name?

Yes, it is. Hang in there, we need unusual names for unusual interfaces. Here b reminds us that the visitor’s methods produce booleans.

Oh My!
How many methods does the definition of \texttt{blsFlat}$^V$ contain, assuming it implements \texttt{bTreeVisitor}$^T$?

Three, because it works with \texttt{Tree}$^D$'s, and the datatype definition for \texttt{Tree}$^D$'s has three variants.

What type of values do the methods of \texttt{blsFlat}$^V$ produce?

\texttt{booleans}.

What visitor does \texttt{blsFlat}$^V$ remind us of?

\texttt{OnlyOnions}$^V$.

Here is a skeleton for \texttt{blsFlat}$^V$.

Here’s easy now.

\begin{verbatim}
class blsFlat$^V$ implements bTreeVisitor$^T$ {
  public
    boolean forBud() {
      return _______ ;
    }
  public
    boolean forFlat(Fruit$^D$ f, Tree$^D$ t) {
      return _______ ;
    }
  public
    boolean forSplit(Tree$^D$ l, Tree$^D$ r) {
      return _______ ;
    }
}
\end{verbatim}

Fill in the blanks.

Define the \texttt{blsSplit}$^V$ visitor, whose methods check whether a \texttt{Tree}$^D$ is constructed with \texttt{Split} and \texttt{Bud} only.

\begin{verbatim}
class blsSplit$^V$ implements bTreeVisitor$^T$ {
  public
    boolean forBud() {
      return true;
    }
  public
    boolean forFlat(Fruit$^D$ f, Tree$^D$ t) {
      return t.accept(this);
    }
  public
    boolean forSplit(Tree$^D$ l, Tree$^D$ r) {
      return false;
    }
}
\end{verbatim}

Here is the easy part.

\begin{verbatim}
class blsSplit$^V$ implements bTreeVisitor$^T$ {
  public
    boolean forBud() {
      return true;
    }
  public
    boolean forFlat(Fruit$^D$ f, Tree$^D$ t) {
      return false;
    }
  public
    boolean forSplit(Tree$^D$ l, Tree$^D$ r) {
      _______ 
    }
}
\end{verbatim}
What is difficult about the last line?

We need to check whether both $l$ and $r$ are split trees.

Isn't that easy?

Yes, we just use the methods of blsSplit\textsuperscript{Y} on $l$ and $r$.

And then?

Then we need to know that both are true.

If

$L.accept(this)$
is true, do we need to know whether

$r.accept(this)$
is true?

Yes, because if both are true, we have a split tree.

If

$L.accept(this)$
is false, do we need to know whether

$r.accept(this)$
is true?

No, then the answer is false.

Finish the definition of blsSplit\textsuperscript{Y} using

if (...) return ... else return ...

```
class blsSplit\textsuperscript{Y} implements bTreeVisitor\textsuperscript{T} {
    public
    boolean forBud() {
        return true;
    }

    public
    boolean forFlat(Fruit\textsuperscript{D} f, Tree\textsuperscript{D} t) {
        return false;
    }

    public
    boolean forSplit(Tree\textsuperscript{D} l, Tree\textsuperscript{D} r) {
        if ($L.accept(this)$)
            return $r.accept(this)$;
        else
            return false;
    }
}
```

1 We could have written the if ... as return $l.accept(this)$ && $r.accept(this)$

Oh My!
Give an example of a $\text{Tree}^D$ for which the methods of $\text{bIsSplit}^Y$ respond with true.

There is a trivial one:

\[
\text{new Bud()}. 
\]

How about one with five uses of Split?

Here is one:

\[
\begin{align*}
\text{new Split(} \\
\text{new Bud(),} \\
\text{new Split(} \\
\text{new Bud(),} \\
\text{new Bud())),} \\
\text{new Split(} \\
\text{new Bud(),} \\
\text{new Split(} \\
\text{new Bud(),} \\
\text{new Bud()))).}
\end{align*}
\]

Does this $\text{Tree}^D$ have any fruit?

No.

Define the $\text{bHasFruit}^Y$ visitor.

Here it is.

\begin{verbatim}
class bHasFruit$^Y$
    implements bTreeVisitor$^Y$
    {
public
    boolean forBud() {
        return false; }
public
    boolean forFlat(Fruit$^D$ f,Tree$^D$ t) {
        return true; }
public
    boolean forSplit(Tree$^D$ l,Tree$^D$ r) {
        if (l.accept(this))
            return true;
        else
            return r.accept(this); }
    }
\end{verbatim}

\footnote{We could have written the if \ldots as
\[\text{return l.accept(this) || r.accept(this)}\]}

104

Chapter 7
What is the height of
new Split(
  new Split(
    new Bud(),
    new Flat(new Lemon(),
    new Bud())),
  new Flat(new Fig(),
  new Split(
    new Bud(),
    new Bud())))?

What is the height of
new Split(
  new Bud(),
  new Flat(new Lemon(),
  new Bud()))?

What is the height of
new Flat(new Lemon(),
  new Bud())?

What is the height of
new Bud()?

So what is the height of a Tree^D?  
32 Just as in nature, the height of a tree is the distance from the beginning to the highest bud in the tree.

Do the methods of iHeight^v work on a Tree^D?  
33 Yes, and they produce an int.

Is that what the i in front of Height is all about?  
34 It looks like i stands for int, doesn’t it?

Oh My!

105
What is the value of
\[
\text{new Split(}
\text{new Split(}
\text{new Bud(),}
\text{new Bud()},
\text{new Flat(new Fig()),}
\text{new Flat(new Lemon()),}
\text{new Flat(new Apple()),}
\text{new Bud()}))
\text{.accept(new \text{Height}^V())?}
\]

Why is the height 4?  
Because the value of
\[
\text{new Split(}
\text{new Bud(),}
\text{new Bud()}\text{.accept(new \text{Height}^V())}
\]
is 1; the value of
\[
\text{new Flat(new Fig()),}
\text{new Flat(new Lemon()),}
\text{new Flat(new Apple()),}
\text{new Bud()}\text{.accept(new \text{Height}^V())}
\]
is 3; and the larger of the two numbers is 3.

And how do we get from 3 to 4?  
We need to add one to the larger of the numbers so that we don’t forget that the original Tree^P was constructed with Split and those two Tree^P’s.

∪ picks the larger of two numbers, \(x\) and \(y\).  
Oh, that’s nice. What kind of methods does \text{Height}^V define?

\text{Height}^V’s methods measure the heights of the Tree^P’s to which they correspond.

\footnote{When you enter this in a file, use}\textsf{Math.max}(x, y)\footnote{Math is a class that contains \textsf{max} as a (static) method.}

Now that’s a problem.
Why?

We defined only interfaces that produce booleans in this chapter.

So what?

The methods of iHeight\(^{V}\) produce ints, which are not booleans.

Okay, so let's define a visitor interface that produces ints.

It's almost the same as bTreeVisitor\(^{T}\).

```java
interface iTreeVisitor\(^{T}\) {
    int forBud();
    int forFlat(Fruit\(^{D}\) f, Tree\(^{D}\) t);
    int forSplit(Tree\(^{D}\) l, Tree\(^{D}\) r);
}
```

Yes, and once we have that we can add another accept method to Tree\(^{D}\).

Does that mean we can have two methods with the same name in one class?

```java
abstract class Tree\(^{D}\) {
    abstract boolean accept(bTreeVisitor\(^{T}\) ask);
    abstract int accept(iTreeVisitor\(^{T}\) ask);
}
```

We can have two methods with the same name in the same class as long as the types of the things they consume are distinct.

bTreeVisitor\(^{T}\) is indeed different from iTreeVisitor\(^{T}\), so we can have two versions of accept in Tree\(^{D}\).

Add the new accept methods to Tree\(^{D}\)’s variants. Start with the easy one.

It is easy.

```java
class Bud extends Tree\(^{D}\) {
    boolean accept(bTreeVisitor\(^{T}\) ask) {
        return ask.forBud();
    }
    int accept(iTreeVisitor\(^{T}\) ask) {
        return ask.forBud();
    }
}
```

Oh My!
The others are easy, too. We duplicate `accept`.

```java
class Flat extends TreeD {
    FruitD f;
    TreeD t;
    Flat(FruitD f, TreeD t) {
        f = f;
        t = t;
    }

    boolean accept(bTreeVisitorT ask) {
        return ask.forFlat(f, t);
    }
    int accept(iTreeVisitorT ask) {
        return ask.forFlat(f, t);
    }
}
```

We must also change the type of what the new `accept` method consumes and produces.

```java
class Split extends TreeD {
    TreeD l;
    TreeD r;
    Split(TreeD l, TreeD r) {
        l = l;
        r = r;
    }

    boolean accept(bTreeVisitorT ask) {
        return ask.forSplit(l, r);
    }
    int accept(iTreeVisitorT ask) {
        return ask.forSplit(l, r);
    }
}
```

Here is `iHeightV`.

```java
class iHeightV implements iTreeVisitorT {
    public int forBud() {
        return 0;
    }
    public int forFlat(FruitD f, TreeD t) {
        return 1;
    }
    public int forSplit(TreeD l, TreeD r) {
        return (l.accept(this) + 1) + r.accept(this);
    }
}
```

Complete these methods.

That's easy now.

```java
class iHeightV implements iTreeVisitorT {
    public int forBud() {
        return 0;
    }
    public int forFlat(TreeD f, TreeD t) {
        return t.accept(this) + 1;
    }
    public int forSplit(TreeD l, TreeD r) {
        return l.accept(this) + r.accept(this) + 1;
    }
}
```

What is the value of

```java
new Split(
    new Bud(),
    new Bud())
.accept(new iHeightV())
```

1, of course.

And why is it 1?

Because

```java
    new Bud().accept(new iHeightV())
```

is 0, the larger of 0 and 0 is 0, and one more is 1.
What is the value of
 new Split(
   new Split(
     new Flat(new Fig()),
     new Bud()),
   new Flat(new Fig()),
   new Bud())),
 new Flat(new Fig()),
 new Flat(new Lemon()),
 new Flat(new Apple()),
 new Bud())))
 .accept(
   new tSubst\textsuperscript{V}(
     new Apple(),
     new Fig()));

If the visitor tSubst\textsuperscript{V} substitutes apples for figs, here is what we get:
 new Split(
   new Split(
     new Flat(new Apple()),
     new Bud()),
   new Flat(new Apple()),
   new Bud())),
 new Flat(new Apple()),
 new Flat(new Lemon()),
 new Flat(new Apple()),
 new Bud()));

Correct. Define the tSubst\textsuperscript{V} visitor.

What's the problem?

It's like Subst\textsuperscript{V} and SubstInt\textsuperscript{V} from the end of chapter 5, but we can't do it just yet.

Its methods produce Tree\textsuperscript{P}s, neither ints nor booleans, which means that we need to add yet another interface.

```java
interface tTreeVisitor\textsuperscript{P} {
    Tree\textsuperscript{P} forBud();
    Tree\textsuperscript{P} forFlat(Fruit\textsuperscript{P} f, Tree\textsuperscript{P} t);
    Tree\textsuperscript{P} forSplit(Tree\textsuperscript{P} l, Tree\textsuperscript{P} r);
}
```

Good job. How about the datatype Tree\textsuperscript{P}.

Easy. Here is the abstract one.

```java
abstract class Tree\textsuperscript{P} {
    abstract boolean accept(tTreeVisitor\textsuperscript{P} ask);
    abstract int accept(tTreeVisitor\textsuperscript{P} ask);
    Tree\textsuperscript{P} accept(tTreeVisitor\textsuperscript{P} ask);
}
```

Oh My!
Define the variants of $\text{Tree}^D$.

```java
54 No problem.
class Bud extends Tree$^D$ {
    boolean accept(bTreeVisitor$^T$ ask) {
        return ask.forBud();
    }
    int accept(iTreeVisitor$^T$ ask) {
        return ask.forBud();
    }
    Tree$^D$ accept(tTreeVisitor$^T$ ask) {
        return ask.forBud();
    }
}

class Flat extends Tree$^D$ {
    Fruit$^D$ f;
    Tree$^D$ t;
    Flat(Fruit$^D$.f,Tree$^D$.t) {
        f = .f;
        t = .t;
    }
    boolean accept(bTreeVisitor$^T$ ask) {
        return ask.forFlat(f,t);
    }
    int accept(iTreeVisitor$^T$ ask) {
        return ask.forFlat(f,t);
    }
    Tree$^D$ accept(tTreeVisitor$^T$ ask) {
        return ask.forFlat(f,t);
    }
}

class Split extends Tree$^D$ {
    Tree$^D$ l;
    Tree$^D$ r;
    Split(Tree$^D$.l,Tree$^D$.r) {
        l = .l;
        r = .r;
    }
    boolean accept(bTreeVisitor$^T$ ask) {
        return ask.forSplit(l,r);
    }
    int accept(iTreeVisitor$^T$ ask) {
        return ask.forSplit(l,r);
    }
    Tree$^D$ accept(tTreeVisitor$^T$ ask) {
        return ask.forSplit(l,r);
    }
}
```
Then define $t\text{Subst}^V$.

That's easy, too. It has two fields, one for the new $\text{Fruit}^D$ and one for the old one, and the rest is straightforward.

```java
class tSubstV implements tTreeVisitor
{
    FruitD n;
    FruitD o;
    tSubstV(FruitD _n,FruitD _o) {
        n = _n;
        o = _o;
    }

    public TreeD forBud() {
        return new Bud();
    }

    public TreeD forFlat(FruitD f,TreeD t) {
        if (o.equals(f))
            return new Flat(n,t.accept(this));
        else
            return new Flat(f,t.accept(this));
    }

    public TreeD forSplit(TreeD l,TreeD r) {
        return new Split(l.accept(this),r.accept(this));
    }
}
```

Here is a $\text{Tree}^D$ that has three Figs:

```
new Split(
    new Split(
        new Flat(new Fig(),
            new Bud()),
        new Flat(new Fig(),
            new Bud())),
    new Flat(new Fig(),
        new Flat(new Lemon(),
            new Flat(new Apple(),
                new Bud()))));
```

Now define $i\text{Occurs}^V$, whose methods count how often some $\text{Fruit}^D$ occurs in a tree.

Even the visitors are no longer interesting.

```java
class iOccursV implements iTreeVisitor
{
    FruitD a;
    iOccursV(FruitD _a) {
        a = _a;
    }

    public int forBud() {
        return 0;
    }

    public int forFlat(FruitD f,TreeD t) {
        if (f.equals(a))
            return t.accept(this) + 1;
        else
            return t.accept(this);
    }

    public int forSplit(TreeD l,TreeD r) {
        return 
            l.accept(this) + r.accept(this);
    }
}
```

Oh My!
Do you like your fruit with yogurt?  

We prefer coconut sorbet.

Is it disturbing that we have three nearly identical versions of accept in TreeD’s and its variants?  

Copying definitions is always bad. If we make a mistake and copy a definition, we copy mistakes. If we modify one, it’s likely that we might forget to modify the other.

Can we avoid it?  

If boolean and int were classes, we could use Object for boolean, int, and TreeD. Unfortunately, they are not.

Remember Integer and Boolean? They make it possible.  

Yes, Boolean is the class that corresponds to boolean, and Integer corresponds to int.

Here is the interface for a protocol that produces Object in place of boolean, int, and TreeD.

```java
interface TreeVisitorD {
    Object forBud();
    Object forFlat(FruitD f, TreeD t);
    Object forSplit(TreeD l, TreeD r);
}
```

Here is the datatype and the Bud variant.

```java
abstract class TreeD {
    abstract
    Object accept(TreeVisitorD ask);
}
```

```java
class Bud extends TreeD {
    Object accept(TreeVisitorD ask) {
        return ask.forBud();
    }
}
```

Define the remaining variants of TreeD.

Here they are.

```java
class Flat extends TreeD {
    FruitD f;
    TreeD t;
    Flat(FruitD f, TreeD t) {
        f = f;
        t = t;
    }
}
```

```java
Object accept(TreeVisitorD ask) {
    return ask.forFlat(f, t);
}
```

```java
class Split extends TreeD {
    TreeD l;
    TreeD r;
    Split(TreeD l, TreeD r) {
        l = l;
        r = r;
    }
}
```

```java
Object accept(TreeVisitorD ask) {
    return ask.forSplit(l, r);
}
```
Good. Now define \texttt{isFlat$^V$}, an Object producing version of \texttt{isFlat$^V$}.

That’s no big deal.

```java
class isFlat$^V$ implements TreeVisitor$^T$
{
  public Object forBud()
  {
    return new Boolean(true);
  }
  public Object forFlat(Fruit$^D$ f, Tree$^D$ t)
  {
    return t.accept(this);
  }
  public Object forSplit(Tree$^D$ l, Tree$^D$ r)
  {
    return new Boolean(false);
  }
}
```

And how about \texttt{isSplit$^V$}?  

Now that’s different. Here we need a way to determine the underlying \texttt{boolean} of the \texttt{Boolean} that is produced by \texttt{L.accept(this)} in the original definition.

Okay, here it is.

```java
class isSplit$^V$ implements TreeVisitor$^T$
{
  public Object forBud()
  {
    return new Boolean(true);
  }
  public Object forFlat(Fruit$^D$ f, Tree$^D$ t)
  {
    return new Boolean(false);
  }
  public Object forSplit(Tree$^D$ l, Tree$^D$ r)
  {
    if (((Boolean) (L.accept(this))).booleanValue())
      return r.accept(this);
    else
      return new Boolean(false);
  }
}
```

Oh, because \texttt{L.accept(this)} produces an Object, we must first convert\footnote{If Java had parametric polymorphism for methods, no downward cast would be necessary for our visitors (Martin Odersky and Philip Wadler, Fuzz into Java: Translating Theory into Practice, Conference Record on Principles of Programming Languages, 146–159. Paris, 1997).} it to a \texttt{Boolean}. Then we can determine the underlying \texttt{boolean} with the \texttt{booleanValue} method. We have seen this in chapter 5 when we converted an Object to a \texttt{OneMoreThan}.

Will the conversion always work?

Yes, because the Object produced by \texttt{L.accept(this)} is always a \texttt{Boolean}.

---

**The Seventh Bit of Advice**

*When designing visitor protocols for many different types, create a unifying protocol using \texttt{Object}.*

Oh My!  

113
class Occurs$^3$ implements TreeVisitor$^2$

Fruit$^D$ $a$;
Occurs$^7$(Fruit$^D$ .a) {
  $a = .a$;
}

public Object forBud() {
  return new Integer(0);
}

public Object forFlat(Fruit$^D$ $f$, Tree$^D$ $t$) {
  if ($f$.equals($a$))
    return new Integer($(((\text{Integer})$
      $t$.accept(this))$
      \text{.intValue}()$
      + 1);
  else
    return $t$.accept(this);
}

public Object forSplit(Tree$^D$ $l$, Tree$^D$ $r$) {
  return new Integer($(((\text{Integer})$
    $l$.accept(this))$
    \text{.intValue}()$
    +
    $(((\text{Integer})$
      $r$.accept(this))$
      \text{.intValue}());
})
8. Like Father, Like Son
What is the value of 
\((7 + ((4 - 3) \times 5))\)?

1. 12.

What is the value of 
\((+ 7 (\times (-(4 \ 3) 5)))\)?

2. 12, because we have just rewritten the previous expression with prefix operators.

What is the value of
\[
\text{new Plus}(
\text{new Const(new Integer(7)),}
\text{new Prod(
\text{new Diff(
\text{new Const(new Integer(4)),}
\text{new Const(new Integer(3)))},
\text{new Const(new Integer(5)))})}
)\]

3. \text{new Integer(12)}, because we have just rewritten the previous expression using \text{Integer} and constructors.

Where do the constructors come from?

4. A datatype and its variants that represent arithmetic expressions.

Did you like that?

5. So far, so good.

What is the value of 
\((\{7, 5\} \cup (((4 \ \setminus \ {3}) \cap \{5\}))\)?

6. \{7, 5\}.

What is the value of 
\((\cup \{7, 5\} (\cap (\setminus \{4 \ 3\} \{5\}))\)?

7. \{7, 5\}, we just went from infix to prefix notation.

What is the value of 
\((+ \{7, 5\} \times (-(4 \ 3) \{5\}))\)?

8. \{7, 5\}, we just renamed the operators.
What is the value of

\[
\begin{align*}
\text{new Plus(} & \\
\text{new Const(new Empty(} & \\
\text{.add(new Integer(7))} & \\
\text{.add(new Integer(5)))).} & \\
\text{new Prod(} & \\
\text{new Diff(} & \\
\text{new Const(new Empty(} & \\
\text{.add(new Integer(4)))).} & \\
\text{new Const(new Empty(} & \\
\text{.add(new Integer(3)))).} & \\
\text{new Const(new Empty(} & \\
\text{.add(new Integer(5)))).}
\end{align*}
\]

9 new Empty()
   .add(new Integer(7))
   .add(new Integer(5)),
because we have just rewritten the
previous expression using the
constructors.

Where do the constructors come from?

10 A datatype and its variants that represent
set expressions.

Do you still like it?

11 Sure, why not.

Does the arithmetic expression look like the
set expression?

12 Yes, they look the same except for the
constants:
\[
\begin{align*}
\text{new Plus(} & \\
\text{new Const(} & \\
\text{new Diff(} & \\
\text{new Const(} & \\
\text{new Const(} & \\
\text{new Const(} &
\end{align*}
\]

Let's say that an expression is either

\[
\begin{align*}
a \text{ Plus(expr}_1, \text{expr}_2), \\
a \text{ Diff(expr}_1, \text{expr}_2), \\
a \text{ Prod(expr}_1, \text{expr}_2), \text{ or} \\
a \text{ a constant},
\end{align*}
\]

where \text{expr}_1 \text{ and } \text{expr}_2 \text{ stand for arbitrary}
expressions. What should be the visitor
interface?

13 That's a tricky question.

\[
\text{interface ExprVisitor }^D \{ \\
\text{Object forPlus(Expr}^D l, \text{Expr}^D r); \\
\text{Object forDiff(Expr}^D l, \text{Expr}^D r); \\
\text{Object forProd(Expr}^D l, \text{Expr}^D r); \\
\text{Object forConst(Object c);} \\
\}
\]
Good answer. Here is the datatype now.

```
abstract class ExprD {
    abstract
    Object accept(ExprVisitor ask);
}
```

Define the variants of the datatype and equip them with an `accept` method that produces Objects.

```
class Plus extends ExprD {
    ExprD l;
    ExprD r;
    Plus(ExprD \_l,ExprD \_r) {
        l = \_l;
        r = \_r;
    }
    Object accept(ExprVisitor ask) {
        return ask.forPlus(l,r);
    }
}
```

```
class Diff extends ExprD {
    ExprD l;
    ExprD r;
    Diff(ExprD \_l,ExprD \_r) {
        l = \_l;
        r = \_r;
    }
    Object accept(ExprVisitor ask) {
        return ask.forDiff(l,r);
    }
}
```

```
class Prod extends ExprD {
    ExprD l;
    ExprD r;
    Prod(ExprD \_l,ExprD \_r) {
        l = \_l;
        r = \_r;
    }
    Object accept(ExprVisitor ask) {
        return ask.forProd(l,r);
    }
}
```

```
class Const extends ExprD {
    Object c;
    Const(Object \_c) {
        c = \_c;
    }
    Object accept(ExprVisitor ask) {
        return ask.forConst(c);
    }
}
```

*Like Father, Like Son*
Can we now define a visitor whose methods determine the value of an arithmetic expression?

15 Yes, we can. It must have four methods, one per variant, and it is like Occurs\(^3\) from the previous chapter.

How do we add

new Integer(3)
and
new Integer(2)?

16 We have done this before. We use the method intValue to determine the ints that correspond to the Integers, and then add them together.

But what is the result of

new Integer(3).intValue() +
new Integer(2).intValue()?

17 An int. what else?

How do we turn that into an Integer?

18 We use new Integer(...).

Okay, so here is a skeleton of IntEval\(^3\).

```java
class IntEval\(^3\) implements ExprVisitor\(^2\) {
    public Object forPlus(Expr\(^\textsuperscript{D}\) l, Expr\(^\textsuperscript{D}\) r) {
        return plus(l.accept(this),
                   r.accept(this));}
    public Object forDiff(Expr\(^\textsuperscript{D}\) l, Expr\(^\textsuperscript{D}\) r) {
        return diff(l.accept(this),
                   r.accept(this));}
    public Object forProd(Expr\(^\textsuperscript{D}\) l, Expr\(^\textsuperscript{D}\) r) {
        return prod(l.accept(this),
                   r.accept(this));}
    public Object forConst(Object c) {
        return c; }
    Object plus(int l, int r) {
        return l + r; }
    Object diff(int l, int r) {
        return l - r; }
    Object prod(int l, int r) {
        return l * r; }
}
```

19 That's an interesting skeleton. It contains five different kinds of blanks and two of them occur three times each. But we can see the bones only. Where is the beef?
How does forPlus work?

It consumes two Exp

20

's, determines their respective values, and places them.

How are the values represented?

As Objects, because we are using our most general kind of (and most recent) visitor.

So what kind of values must plus consume?

Objects,

22

because that’s what

l.accept(this)

and

r.accept(this)

produce.

What must we put in the first and second blanks?

Object.

Can we add Objects?

24

No, we must convert them to Integers first and extract their underlying ints.

Can we convert all Objects to Integers?

25

No, but all Objects produced by IntEval

are made with new Integer(...), so that this conversion always succeeds.

Is that true? What is the value of

new Plus(

26

new Const(new Empty()),

new Const(new Integer(5)))

.accept(new IntEval

())?

Wow. At some level, this is nonsense.

Correct, so sometimes the conversion may fail, because we use an instance of IntEval

27

on nonsensical arithmetic expressions.

Like Father, Like Son
We agree to avoid such arithmetic expressions.\footnote{In other words, we have unsafe evaluators for our expressions. One way to make them safe is to add a method that checks whether constants are instances of the proper class and that raises an exception. [Chapter 7]. An alternative is to define a visitor that type checks the arithmetic expressions we wish to evaluate.}

And their set expressions, too.

If we want to add \( l \) and \( r \), we write

\[
\text{new Integer}((\text{Integer})l).intValue() + ((\text{Integer})r).intValue()).
\]

Complete the definition now.

\[
\begin{align*}
\text{class IntEval}&\text{ implements ExprVisitor}\{ \\
\text{public }&\text{ Object forPlus(Expr } l, \text{ Expr } r) \{ \\
&\text{return plus(l.accept(this).} \\
&\text{r.accept(this)); } \\
\text{public }&\text{ Object forDiff(Expr } l, \text{ Expr } r) \{ \\
&\text{return diff(l.accept(this).} \\
&\text{r.accept(this)); } \\
\text{public }&\text{ Object forProd(Expr } l, \text{ Expr } r) \{ \\
&\text{return prod(l.accept(this).} \\
&\text{r.accept(this)); } \\
\text{public }&\text{ Object forConst(Expr } c) \{ \\
&\text{return c; } \\
&\text{Object plus(Object } l, \text{ Object } r) \{ \\
&\text{return new Integer}((\text{Integer})l).intValue() + ((\text{Integer})r).intValue()); } \\
\text{Object diff(Object } l, \text{ Object } r) \{ \\
&\text{return new Integer}((\text{Integer})l).intValue() \\
&\text{-(((Integer)r).intValue())); } \\
\text{Object prod(Object } l, \text{ Object } r) \{ \\
&\text{return new Integer}((\text{Integer})l).intValue() \\
&\text{*(((Integer)r).intValue()); } \\
\end{align*}
\]

Now it's easy. Here we go.
That one was pretty easy, wasn’t it?  

Yes. Let’s implement an ExprVisitor for sets.

What do we need to implement one for sets?  

We certainly need methods for \textit{plusing}, \textit{diffing}, and \textit{proding} sets.

That’s correct, and here is everything.

\begin{verbatim}
abstract class SetD {
    SetD add(Integer i) {
        if (mem(i))
            return this;
        else
            return new Add(i,this); }

    abstract boolean mem(Integer i);
    abstract SetD plus(SetD s);
    abstract SetD diff(SetD s);
    abstract SetD prod(SetD s);
}
\end{verbatim}

Whoa.

Explain the method in the nested box in your own words.

We use our words: “As its name says, \textit{add} adds an element to a set. If the element is a \textit{member} of the set, the set remains the same; otherwise, a \textit{new} set is constructed with \textit{Add}.”

Why is this so tricky?

Constructors always construct, and \textit{add} does not always construct.

Do we need to understand that?

Not now, but feel free to absorb it when you have the time.
Define the variants `Empty` and `Add for Set^D`. Here we go.

```java
class Empty extends Set^D {
    boolean mem(Integer i) {
        return false;
    }

    Set^D plus(Set^D s) {
        return s;
    }

    Set^D diff(Set^D s) {
        return new Empty();
    }

    Set^D prod(Set^D s) {
        return new Empty();
    }
}

class Add extends Set^D {
    Integer i;
    Set^D s;

    Add(Integer .i, Set^D .s) {
        i = .i;
        s = .s;
    }

    boolean mem(Integer n) {
        if (i.equals(n))
            return true;
        else
            return s.mem(n);
    }

    Set^D plus(Set^D t) {
        return s.plus(t.add(i));
    }

    Set^D diff(Set^D t) {
        if (i.mem(i))
            return s.diff(t);
        else
            return s.diff(t).add(i);
    }

    Set^D prod(Set^D t) {
        if (i.mem(i))
            return s.prod(t).add(i);
        else
            return s.prod(t);
    }
}
```
Do we need to understand these definitions? 37 Not now, but feel free to think about them when you have the time. We haven’t even used visitors to define operations for union, set-difference, and intersection, but we trust you can.

What do we have to change in IntEval\textsuperscript{V} to obtain SetEval\textsuperscript{V}, an evaluator for set expressions? 38 Not much, just \texttt{plus}, \texttt{diff}, and \texttt{prod}.

How should we do that? 39 Oh, that’s a piece of pie. We just copy the definition of IntEval\textsuperscript{V} and replace its \texttt{plus}, \texttt{diff}, and \texttt{prod} methods.

That’s the worst way of doing that. 40 What?

Why should we throw away more than half of what we have? 41 That’s true. If we copied the definition and changed it, we would have identical copies of \texttt{forPlus}, \texttt{forDiff}, \texttt{forProd}, and \texttt{forConst}. We should reuse this definition.\textsuperscript{1}

\textsuperscript{1} Sometimes we do not have license to see the definitions, so copying might not even be an option.

Yes, and we are about to show you better ways. How do we have to change \texttt{plus}, \texttt{diff}, and \texttt{prod}? 42 That part is easy:

\begin{verbatim}
Object plus(Object l, Object r) {
    return ((Set\textsuperscript{D})l).plus((Set\textsuperscript{D})r);
}

and

Object diff(Object l, Object r) {
    return ((Set\textsuperscript{D})l).diff((Set\textsuperscript{D})r);
}

and

Object prod(Object l, Object r) {
    return ((Set\textsuperscript{D})l).prod((Set\textsuperscript{D})r);
}
\end{verbatim}
Very good, and if we define SetEval\textsuperscript{V} as an extension of IntEval\textsuperscript{V}, that's all we have to put inside of SetEval\textsuperscript{V}.

```java
class SetEval\textsuperscript{V} extends IntEval\textsuperscript{V} {
    Object plus(Object l, Object r) {
        return ((Set\textsuperscript{D})l).plus((Set\textsuperscript{D})r);
    }
    Object diff(Object l, Object r) {
        return ((Set\textsuperscript{D})l).diff((Set\textsuperscript{D})r);
    }
    Object prod(Object l, Object r) {
        return ((Set\textsuperscript{D})l).prod((Set\textsuperscript{D})r);
    }
}
```

44 Now that's much easier than copying and modifying.

Is it like equals?  

44 Yes, when we include equals in our class definitions, we override the one in Object. Here, we override the methods plus, diff, and prod as we extend IntEval\textsuperscript{V}.

How many methods from IntEval\textsuperscript{V} are overridden in SetEval\textsuperscript{V}?  

45 Three.

How many methods from IntEval\textsuperscript{V} are not overridden in SetEval\textsuperscript{V}?  

46 Four: forPlus, forDiff, forProd, and forConst.

Does SetEval\textsuperscript{V} implement ExprVisitor\textsuperscript{T}?  

47 It doesn't say so.

Does SetEval\textsuperscript{V} extend IntEval\textsuperscript{V}?  

48 It says so.

Does IntEval\textsuperscript{V} implement ExprVisitor\textsuperscript{T}?  

49 It says so.

Does SetEval\textsuperscript{V} implement ExprVisitor\textsuperscript{T}?  

50 By implication.

126  Chapter 8
That's correct. What is the value of
\[
\text{new Prod(}
\text{  new Const(new Empty())}
\text{    .add(new Integer(7))),}
\text{  new Const(new Empty())}
\text{    .add(new Integer(3)))}
\text{  .accept(new SetEvalP())?}
\]
51 Interesting question. How does this work now?

What type of value is
\[
\text{new Prod(}
\text{  new Const(new Empty())}
\text{    .add(new Integer(7))),}
\text{  new Const(new Empty())}
\text{    .add(new Integer(3)))?}
\]
52 It is a Prod and therefore an \( \text{Expr}^D \).

And what does \text{accept} consume?
\[
\text{new SetEvalP(), forProd(}
\text{  new Const(new Empty())}
\text{    .add(new Integer(7))),}
\text{  new Const(new Empty())}
\text{    .add(new Integer(3)))?}
\]
53 An instance of \( \text{SetEvalP} \), but its type is \( \text{ExprVisitor}^T \).

What is
\[
\text{new SetEvalP(), forProd(}
\text{  new Const(new Empty())}
\text{    .add(new Integer(7))),}
\text{  new Const(new Empty())}
\text{    .add(new Integer(3)))?}
\]
54 That's what we need to determine the value of next, because it is \( \text{ask forProd(l,r)} \),
with \( \text{ask, l, t, and r replaced by what they stand for} \).

Where is the definition of \( \text{SetEvalP} \)'s method \( \text{forProd} \)?
\[
\text{It is in \text{IntEvalP}.}
\]
55

Suppose we had the values of
\[
\text{new Const(new Empty())}
\text{  .add(new Integer(7)))}
\text{  .accept(this)}
\]
and
\[
\text{new Const(new Empty())}
\text{  .add(new Integer(3)))}
\text{  .accept(this).}
\]
What would we have to evaluate next?

56 If their values were \( A \) and \( B \), we would have to determine the value of \( \text{prod(A,B)} \).

\text{Like Father, Like Son}
Isn’t that strange? Why?

So far, we have always used a method on a particular object. That’s true. What is the object with which we use $prod(A,B)$?

It is this object. Oh, does that mean we should evaluate $new\ SetEval^V().prod(A,B)$?

Absolutely. If the use of a method omits the object, we take the one that we were working with before. That clarifies things.

Good. And now what? Now we still need to determine the values of $new\ Const(new\ Empty().\ add(new\ Integer(7))).\ accept(this)$ and $new\ Const(new\ Empty().\ add(new\ Integer(3))).\ accept(this)$.

The values are obviously $new\ Empty().\ add(new\ Integer(7))$ and $new\ Empty().\ add(new\ Integer(3))$. It, too, is in $\text{IntEval}^V$.

Where is the definition of $forConst$ that determines these values? Where does $prod$ come from?

Here is the next expression in our sequence: $new\ SetEval^V().\ prod(new\ Empty().\ add(new\ Integer(7)), new\ Empty().\ add(new\ Integer(3)))$. The object is an instance of $SetEval^V$, which overrides the $prod$ method in $\text{IntEval}^V$ with its own.
What next?  

64 Next we need to determine the value of
   \[(\text{Set}^D)(\text{new Empty}())
   \cdot \text{add}(\text{new Integer}(7)))
   \cdot \text{prod}(\text{Set}^D)\text{new Empty}()
   \cdot \text{add}(\text{new Integer}(3)),\]

because it is
   \[(\text{Set}^D)(\text{l.accept(this)})
   \cdot \text{prod}(\text{Set}^D)\text{r.accept(this))}\]

with \text{l.accept(this)} and \text{r.accept(this)}
replaced by their respective values.

Is
   \text{new Empty}().\text{add}(\text{new Integer}(7))

an instance of \text{Set}^D?  

65 Of course it is, but the type of \text{l.accept(this)},
which is where it comes from, is \text{Object}.

And how about
   \text{new Empty}().\text{add}(\text{new Integer}(3))?  

66 It’s the same.

And that is why the method must contain a
conversion from \text{Object} to \text{Set}^D's.  

67 This example makes the need for conversions
obvious again.

Time for the last question. Where does this
\text{prod} come from now?  

68 This one belongs to \text{Set}^D or more precisely
its \text{Empty} and \text{Add} variants.

And what does \text{prod} do?  

69 It determines the intersection of one \text{Set}^D
with another \text{Set}^D, but didn’t we agree that
the previous question was the last question
on that topic?

We overrode that, too.  

70 Thanks, guys.

Is it natural that \text{SetEval}^V extends \text{IntEval}^V?  

71 No, not at all.

\textit{Like Father, Like Son}
Why did we do that? Because we defined `IntEvalV` first.\footnote{Sometimes we may need to extend classes that are used in several different programs. Unless we wish to maintain multiple copies of the same class, we should extend it. Java is object-oriented, so it may also be the case that we acquire the object code of a class and its interface, but not its source text. If we wish to enrich the functionality of this kind of class, we must also extend it.}

But just because something works, it doesn't mean it's rational. Yes, let's do better. We have defined all these classes ourselves, so we are free to rearrange them any way we want.

What distinguishes `IntEvalV` from `SetEvalV`? The methods `plus`, `diff`, and `prod`.

What are the pieces that they have in common? They share the methods `forPlus`, `forDiff`, `forProd`, and `forConst`.

Good. Here is how we express that.

```java
abstract class EvalP {
    implements ExprVisitorT {
        public Object forPlus(ExprP l, ExprP r) {
            return plus(l.accept(this),
                      r.accept(this));
        }
        public Object forDiff(ExprP l, ExprP r) {
            return diff(l.accept(this),
                        r.accept(this));
        }
        public Object forProd(ExprP l, ExprP r) {
            return prod(l.accept(this),
                        r.accept(this));
        }
        public Object forConst(Object c) {
            return c;
        }
    abstract Object plus(Object l, Object r);
    abstract Object diff(Object l, Object r);
    abstract Object prod(Object l, Object r);
    }
```
Yes, we can think of it as a datatype for EvalD visitors that collects all the common elements as concrete methods. The pieces that differ from one variant to another are specified as abstract methods.

We define IntEvalV extending EvalD.

```java
class IntEvalV extends EvalD {
  Object plus(Object l, Object r) {
    return
    new Integer((Integer)l).intValue() +
    ((Integer)r).intValue();
  }
  Object diff(Object l, Object r) {
    return
    new Integer((Integer)l).intValue() -
    ((Integer)r).intValue();
  }
  Object prod(Object l, Object r) {
    return
    new Integer((Integer)l).intValue() *
    ((Integer)r).intValue();
  }
}
```

Define SetEvalV.

It is basically like the original but extends EvalD, not IntEvalV.

```java
class SetEvalV extends EvalD {
  Object plus(Object l, Object r) {
    return ((SetD)l).plus((SetD)r);
  }
  Object diff(Object l, Object r) {
    return ((SetD)l).diff((SetD)r);
  }
  Object prod(Object l, Object r) {
    return ((SetD)l).prod((SetD)r);
  }
}
```

Is it natural for two evaluators to be on the same footing?

Much more so than one extending the other.

Time for supper.

If you are neither hungry nor tired, you may continue.

Like Father, Like Son
Remember Subst^V from chapter 6?  

```java
class Subst^V implements PieVisitor^T {
    Object n;
    Object o;
    Subst^V(Object .n, Object .o) {
        n = .n;
        o = .o;
    }
    public Pie^D forBot() {
        return new Bot();
    }
    public Pie^D forTop(Object t, Pie^D r) {
        if (o.equals(t))
            return new Top(n, r.accept(this));
        else
            return new Top(t, r.accept(this));
    }
}
```

Yes, and LtdSubst^V, too.

```java
class LtdSubst^V implements PieVisitor^T {
    int c;
    Object n;
    Object o;
    LtdSubst^V(int .c, Object .n, Object .o) {
        c = .c;
        n = .n;
        o = .o;
    }
    public Pie^D forBot() {
        return new Bot();
    }
    public Pie^D forTop(Object t, Pie^D r) {
        if (c == 0)
            return new Top(t, r);
        else
            if (o.equals(t))
                return new Top(n, r.accept(
                    new LtdSubst^V(c - 1, n, o)));
            else
                return new Top(t, r.accept(this));
    }
}
```

---

**What do the two visitors have in common?**

Many things: n, o, and forBot.

---

**Where do they differ?**

They differ in forTop, but LtdSubst^V also has an extra field.

---

**And where do we put the pieces that two classes have in common?**

We put them into an abstract class.

---

**What else does the abstract class contain?**

It specifies the pieces that are different if they are needed for all extensions.

---

132  Chapter 8
Define the abstract class $\text{Subst}^D$, which contains all the common pieces and specifies what a concrete pie substituter must contain in addition.

```java
abstract class $\text{Subst}^D$
    implements PieVisitor$^D$
    {
        Object $n$;
        Object $o$;
        public Pie$^D$ forBot() {
            return new Bot();
        }
        public
        abstract Pie$^D$ forTop(Object $t$, Pie$^D$ $r$);
    }
```

It's not a big deal, except for the fields.

We can define $\text{Subst}^V$ by extending $\text{Subst}^D$.

```java
class $\text{Subst}^V$ extends $\text{Subst}^D$
    {
        $\text{Subst}^V$(Object ..$n$, Object ..$o$) {
            $n$ = ..$n$;
            $o$ = ..$o$;
        }
        public Pie$^D$ forTop(Object $t$, Pie$^D$ $r$) {
            if (o.equals(t))
                return new Top($n$, r.accept(this));
            else
                return new Top($t$, r.accept(this));
        }
    }
```

Define $\text{LtdSubst}^V$.

```java
class $\text{LtdSubst}^V$ extends $\text{Subst}^D$
    {
        int $c$;
        $\text{LtdSubst}^V$(int ..$c$, Object ..$n$, Object ..$o$) {
            $n$ = ..$n$;
            $o$ = ..$o$;
            $c$ = ..$c$;
        }
        public Pie$^D$ forTop(Object $t$, Pie$^D$ $r$) {
            if ($c$ == 0)
                return new Top($t$, $r$);
            else
                if (o.equals(t))
                    return new Top($n$,
                        r.accept(
                            new $\text{LtdSubst}^V$(c - 1, ..$n$, ..$o$)));
                else
                    return new Top($t$, r.accept(this));
        }
    }
```

It also extends $\text{Subst}^D$.

Do the two remaining classes still have things in common?

No, but the constructors have some overlap. Shouldn't we lift the $\text{Subst}^V$ constructor into $\text{Subst}^D$, because it holds the common elements?

Like Father, Like Son
Was that first part easy?

As pie.
That’s neat. How about some art work?  

*Is this called a pie chart?*

---

No, but the picture captures the important relationships.

*Fine.*

---

*Is it also possible to define \( \text{LtdSubst}^Y \) as an extension of \( \text{Subst}^Y \)?*

*It may even be better. In some sense, \( \text{LtdSubst}^Y \) just adds a service to \( \text{Subst}^Y \); it counts as it substitutes.*

---

*If \( \text{LtdSubst}^Y \) is defined as an extension of \( \text{Subst}^Y \), what has to be added and what has to be changed?*

*As we just said, \( c \) is an addition and for \( \text{Top} \) is different.*

---

**The Eighth Bit of Advice**

*When extending a class, use overriding to enrich its functionality.*

---

*Like Father, Like Son*
Here is the good old definition of $\text{Subst}^\text{V}$ from chapter 6 one more time.

```java
class Subst^V implements PieVisitor^T {
    Object n;
    Object o;
    Subst^V(Object n, Object o) {
        n = n;
        o = o;
    }

    public Pie^D forBot() {
        return new Bot();
    }

    public Pie^D forTop(Object t, Pie^D r) {
        if (o.equals(t))
            return new Top(n, r.accept(this));
        else
            return new Top(t, r.accept(this));
    }
}
```

Define $\text{LtdSubst}^\text{V}$ as an extension of $\text{Subst}^\text{V}$.

```java
class LtdSubst^V extends Subst^V {
    int c;
    LtdSubst^V(int c, Object n, Object o) {
        super(n, o);
        c = c;
    }

    public Pie^D forTop(Object t, Pie^D r) {
        if (c == 0)
            return new Top(t, r);
        else
            if (o.equals(t))
                return new Top(n, r.accept(
                    new LtdSubst^V(c - 1, n, o)));
            else
                return new Top(t, r.accept(this));
    }
}
```

The rest follows naturally, just as with the evaluators and the previous version of these two classes.

Let's draw a picture.

![Diagram](attachment:image.png)

You deserve a super-deluxe pizza now.

Fine, and don't forget to use lines, rather than arrows, for `implements`.

It's already on its way.
9.
Be a Good Visitor
Remember Point\textsuperscript{D}? If not, here is the datatype with one additional method, \textit{minus}. We will talk about \textit{minus} when we need it, but for now, just recall Point\textsuperscript{D}’s variants.

\begin{verbatim}
abstract class Point\textsuperscript{D} {
    int x;
    int y;
    Point\textsuperscript{D}(int \_x, int \_y) {
        x = \_x;
        y = \_y;
    }

    boolean closerToO(Point\textsuperscript{D} p) {
        return distanceToO() \leq p.distanceToO();
    }

    Point\textsuperscript{D} minus(Point\textsuperscript{D} p) {
        return new CartesianPt(x - p.x, y - p.y);
    }

    abstract int distanceToO();
}
\end{verbatim}

1. It has been a long time since we discussed the datatype Point\textsuperscript{D} and its variants, but they are not that easy to forget.

\begin{verbatim}
class CartesianPt extends Point\textsuperscript{D} {
    CartesianPt(int \_x, int \_y) {
        super(\_x, \_y);
    }

    int distanceToO() {
        return Math.Sqrt(x\^2 + y\^2);
    }
}
\end{verbatim}

\begin{verbatim}
class ManhattanPt extends Point\textsuperscript{D} {
    ManhattanPt(int \_x, int \_y) {
        super(\_x, \_y);
    }

    int distanceToO() {
        return x + y;
    }
}
\end{verbatim}

Good. Take a look at this extension of ManhattanPt.

\begin{verbatim}
class ShadowedManhattanPt extends ManhattanPt {
    int \_Delta_x;
    int \_Delta_y;
    ShadowedManhattanPt(int \_x, int \_y, int \_Delta_x, int \_Delta_y) {
        super(\_x, \_y);
        \_Delta_x = \_Delta_x;
        \_Delta_y = \_Delta_y;
    }

    int distanceToO() {
        return super.distanceToO() + \_Delta_x + \_Delta_y;
    }
}
\end{verbatim}

What is unusual about the constructor?

2. It uses \Delta_x = \_Delta_x; 
\Delta_y = \_Delta_y; in addition to super(\_x, \_y).

\textit{Be a Good Visitor}
And what does that mean? ³ By using super on the first two values consumed, the constructor creates a ShadowedManhattanPt with proper x and y fields. The rest guarantees that this newly created point also contains values for the two additional fields.

Okay. So what is a ShadowedManhattanPt? ⁴ It is a ManhattanPt with two additional fields: \( \Delta_x \) and \( \Delta_y \). These two represent the information that determines how far the shadow is from the point with the fields \( x \) and \( y \).

Is this a ShadowedManhattanPt: new ShadowedManhattanPt(2,3,1.0)? ⁵ Yes.

What is unusual about \( \text{distanceToO} \)? ⁶ Unlike any other method we have seen before, it contains the word super. So far, we have only seen it used in constructors. What does it mean?

Here, super.distanceToO refers to the method definition of \( \text{distanceToO} \) that is relevant in the class that ShadowedManhattanPt extends. ⁷ Okay. That means we just add \( x \) and \( y \) when we evaluate super.distanceToO().

Correct. But what would we have done if ManhattanPt had not defined \( \text{distanceToO} \)? ⁸ Then we would refer to the definition in the class that ManhattanPt extends, right?

Yes, and so on. What is the value of new ShadowedManhattanPt(2,3,1.0).distanceToO()? ⁹ It is 6, because \( 2 + 3 \) is 5, and then we have to add 1 and 0.
Precisely. Now take a look at this extension of CartesianPt.

```
class ShadowedCartesianPt
    extends CartesianPt {
    int Δx;
    int Δy;
    ShadowedCartesianPt(int .x,
                          int .y,
                          int .Δx,
                          int .Δy) {
        super(.x,.y);
        Δx = .Δx;
        Δy = .Δy;
    }

    int distanceToO() {
        return super.distanceToO()
               + [\sqrt{Δx^2 + Δy^2} ];
    }
}
```

What is unusual about the constructor?

Is this a ShadowedCartesianPt:
```
new ShadowedCartesianPt(12,5,3,4)?
```

And what is the value of
```
new ShadowedCartesianPt(12,5,3,4)
  .distanceToO();
```

It is 18, because the distance of the Cartesian point (12,5) is 13, and then we add 5, because that is the value of
\[ \sqrt{Δx^2 + Δy^2} \]
with Δx replaced by 3 and Δy replaced by 4.

What do we expect?
```
17, obviously.
```

*Be a Good Visitor* 141
Why 17?  

Because we need to think of this point as if it were

```
new CartesianPt(15.9).
```

We need to add \( \Delta_x \) to \( x \) and \( \Delta_y \) to \( y \) when we think of a `ShadowedCartesianPt`.  

And indeed, the value of

```
new CartesianPt(15.9)
.distanceToO()
```

is 17.

Does this explain how `distanceToO` should measure the distance of a `ShadowedCartesianPt` to the origin?  

Completely. It should make a new `CartesianPt` by adding the corresponding fields and should then measure the distance of that new point to the origin.

Revise the definition of `ShadowedCartesianPt` accordingly.

Okay:

```java
class ShadowedCartesianPt
    extends CartesianPt {
    int \( \Delta_x \);
    int \( \Delta_y \);
    ShadowedCartesianPt(int \( x \),
                        int \( y \),
                        int \( \Delta_x \),
                        int \( \Delta_y \)) {
        super(\( x \), \( y \));
        \( \Delta_x = \Delta_x \);
        \( \Delta_y = \Delta_y \); }
    int distanceToO() {
        return
        new CartesianPt(\( x + \Delta_x \), \( y + \Delta_y \))
        .distanceToO();
    }
}
```

Do we still need the new `CartesianPt` after `distanceToO` has determined the distance?  

No, once we have the distance, we have no need for this point.  

\(^1\) And neither does Java. Object-oriented languages manage memory so that programmers can focus on the difficult parts of design and implementation.
Correct. What is the value of
new CartesianPt(3, 4)
  .closerToO(
    new ShadowedCartesianPt(1, 5, 1, 2))?

true,
because the distance of the CartesianPt to
the origin is 5, while that of the
ShadowedCartesianPt is 7.

How did we determine that value?

That's obvious.

Is the rest of this chapter obvious, too?

What?

That was a hint that now is a good time to
take a break.

Oh. Well, that makes the hint obvious.

Come back fully rested. You will more than
need it.

Fine.

Are sandwiches square meals for you?

They can be well-rounded.

Here are circles and squares.

```java
class Circle extends Shape {
  int r;
  Circle(int r) {
    r = r;
  }

  boolean accept(ShapeVisitor ask) {
    return ask.forCircle(r);
  }
}

class Square extends Shape {
  int s;
  Square(int s) {
    s = s;
  }

  boolean accept(ShapeVisitor ask) {
    return ask.forSquare(s);
  }
}
```

Then this must be the datatype that goes
with it.

```java
abstract class Shape {
  abstract
    boolean accept(ShapeVisitor ask);
}
```

Be a Good Visitor
Very good. We also need an interface, and here it is.

```java
interface ShapeVisitor<T {
    boolean forCircle(int r);
    boolean forSquare(int s);
    boolean forTrans(PointD q, ShapeD s);
}
```

Yes and we will need this third variant.

```java
class Trans<
    extends ShapeD {
    PointD q;
    ShapeD s;
    Trans(PointD _q, ShapeD _s) {
        q = _q;
        s = _s;
    }

    boolean accept(ShapeVisitor<T ask) {
        return ask.forTrans(q, s);
    }
}
```

1 A better name is Translation.

Let’s create a circle.

No problem:

```java
new Circle(10).
```

How should we think about that circle?

We should think about it as a circle with radius 10.

Good. So how should we think about new Square(10)?

Well, that’s a square whose sides are 10 units long.

Where are our circle and square located?

What does that mean?
Suppose we wish to determine whether some CartesianPt is inside of the circle? In that case, we must think of the circle as being drawn around the origin.

And how about the square? There are many ways to think about the location of the square.

Pick one. Let's say the square's southwest corner sits on the origin.

That will do. Is the CartesianPt with x coordinate 10 and y coordinate 10 inside the square? Yes, it is, but barely.

And how about the circle? Certainly not, because the circle's radius is 10, but the distance of the point to the origin is 14.

Are all circles and squares located at the origin? We have no choice so far, because Circle and Square only contain one field each: the radius and the length of a side, respectively.

This is where Trans comes in. What is new Trans( new CartesianPt(5,6), new Circle(10))? Aha. With Trans we can place a circle of radius 10 at a point like new CartesianPt(5,6).

How do we place a square's southwest corner at new CartesianPt(5,6)? Also with Trans: new Trans( new CartesianPt(5,6), new Square(10)).

Is new CartesianPt(10,10) inside either the circle or the square that we just referred to? It is inside both of them.

Be a Good Visitor
How do we determine whether some point is inside a circle?

41 If the circle is located at the origin, it is simple. We determine the distance of the point to the origin and whether it is smaller than the radius.

How do we determine whether some point is inside a square?

42 If the square is located at the origin, it is simple. We check whether the point’s $x$ coordinate is between 0 and $s$, the length of the side of the square.

Is that all?

43 No, we also need to do that for the $y$ coordinate.

Aren’t we on a roll?

44 We have only done the easy stuff so far. It is not clear how to check these things when the circle or the square are not located at the origin.

Let’s take a look at our circle around new CartesianPt(5,6) again. Can we think of this point as the origin?

45 We can if we translate all other points by an appropriate amount.

By how much?

46 By 5 in the $x$ direction and 6 in the $y$ direction, respectively.

How could we translate the points by an appropriate amount?

47 We could subtract the appropriate amount from each point.

Is there a method in Point$^D$ that accomplishes that?

48 Yes. Is that why we included minus in the new definition of Point$^D$?
Indeed. And now we can define the visitor HasPtV, whose methods determine whether some Shape has a Point inside of it.

```java
class HasPtV implements ShapeVisitorT {
    Point p;
    HasPtV(Point p) {
        p = p;
    }

    public boolean forCircle(int r) {
        return p.distanceToO() <= r;
    }
    public boolean forSquare(int s) {
        if (p.x <= s)
            return (p.y <= s);
        else
            return false;
    }
    public boolean forTrans(Point q, Shape s) {
        return s.accept(
            new HasPtV(p.minus(q));
        )
    }
}
```

The three methods put into algebra what we just discussed.

1 We could have written the if as
```
return (p.x <= s) && (p.y <= s);
```

What is the value of
```
new Circle(10)
```
```
   .accept(
      new HasPtV(new CartesianPt(10,10)));
```

We said that this point wasn’t inside of that circle, so the answer is false.

Good. And what is the value of
```
new Square(10)
```
```
   .accept(
      new HasPtV(new CartesianPt(10,10)));
```

true.

Let’s consider something a bit more interesting. What is the value of
```
new Trans(
    new CartesianPt(5.6),
    new Circle(10))
   .accept(
      new HasPtV(new CartesianPt(10,10)));
```

We already considered that one, too. The value is true, because the circle’s origin is at
```
new CartesianPt(5.6).
```

Be a Good Visitor
Right. And how about this:

```java
new Trans(
    new CartesianPt(5,4),
    new Trans(
        new CartesianPt(5,6),
        new Circle(10))
).accept(
    new HasPtY(new CartesianPt(10,10)))?
```

53 Now that is tricky. We used Trans twice, which we should have expected given Trans's definition.

But what is the value?

54 First, we have to find out whether

```java
new Trans(
    new CartesianPt(5,6),
    new Circle(10))
.accept(
    new HasPtY(new CartesianPt(5,6)))
```

is true or false.

And then?

55 Second, we need to look at

```java
new Circle(10)
.accept(
    new HasPtY(new CartesianPt(0,0)))
```

but the value of this is obviously true.

Very good. Can we nest Trans three times?

56 Ten times, if we wish, because a Trans contains a Shape\textsuperscript{b}, and that allows us to nest things as often as needed.

Ready to begin?

57 What? Wasn’t that it?

No. The exciting part is about to start.

58 We are all eyes.

How can we project a cube of cheese to a piece of paper?

59 It becomes a square, obviously.

And the orange on top?

60 A circle, Transed appropriately.
Can we think of the two objects as one?  

We can, but we have no way of saying that a circle and a square belong together.

Here is our way.

```java
class Union extends ShapeD {
    ShapeD s;
    ShapeD t;
    Union(ShapeD .s,ShapeD .t) {
        s = .s;
        t = .t;
    }
    boolean accept(ShapeVisitorT ask) {
        return _______; }
}
```

That looks obvious after the fact. But why is there a blank in `accept`?

What do we know from Circle, Square, and Trans about `accept`?

We know that a `ShapeVisitorT` contains one method each for the Circle, Square, and Trans variants. And each of these methods consumes the fields of the respective kinds of objects.

So what should we do now?

We need to change `ShapeVisitorT` so that it specifies a method for the Union variant in addition to the methods for the existing variants.

Correct, except that we won't allow ourselves to change `ShapeVisitorT`.

Why can't we change it?

In that case, we're stuck.

Just to make the problem more interesting.

*Be a Good Visitor*
We would be stuck, but fortunately we can extend interfaces. Take a look at this.

```java
interface UnionVisitor2
    extends ShapeVisitor2 {
    boolean forUnion(Shape s, Shape t);
}
```

Which means that we extend interfaces the way we extend classes.

Basically, this extension produces an interface that contains all the obligations (i.e., names of methods and what they consume and produce) of ShapeVisitor2 and the additional one named forUnion.

1 Unlike a class, an interface can actually extend several other interfaces. A class can implement several different interfaces.

Does that mean accept in Union should receive a UnionVisitor2, so that it can use the forUnion method?

Yes it should, but because UnionVisitor2 extends ShapeVisitor2, it is also a ShapeVisitor2.

We have been here before. Our accept method must consume a ShapeVisitor2 and fortunately every UnionVisitor2 implements a ShapeVisitor2, too. But if we know that accept consumes a UnionVisitor2, we can convert the ShapeVisitor2 to a UnionVisitor2 and invoke the forUnion method.

Perfect reasoning. Here is the completed definition of Union.

```java
class Union extends Shape2 {
    Shape2 s;
    Shape2 t;
    Union(Shape2 s, Shape2 t) {
        s = s;
        t = t;
    }

    boolean accept(ShapeVisitor2 ask) {
        return ((UnionVisitor2)ask).forUnion(s, t);
    }
}
```

And it makes complete sense.
Let's create a Union shape.

That's trivial.

```java
new Trans(
    new CartesianPt(12,2),
    new Union(
        new Square(10),
        new Trans(
            new CartesianPt(4,4),
            new Circle(5)))).
```

That's an interesting shape. Should we check whether

```java
new CartesianPt(12,16)
```

is inside?

Could it be a UnionVisitor? We can't. HasPtV is only a ShapeVisitor, it is not a UnionVisitor.

Define UnionHasPtV, which extends HasPtV with an appropriate method `forUnion`.

Here it is. Its method checks whether the point is in one or the other part of a union. The other methods come from HasPtV.

```java
class UnionHasPtV extends HasPtV {
    UnionHasPtV(PointD p) {
        super(p);
    }

    boolean forUnion(ShapeD s, ShapeD t) {
        if (s.accept(this))
            return true;
        else
            return t.accept(this);
    }
}
```

We could have written the if... as

```java
return s.accept(this) || t.accept(this);
```

Does UnionHasPtV contain `forUnion`? Of course, we just put it in.

*Be a Good Visitor*
Is `UnionHasPt^V` a `UnionVisitor^T`? It provides the required methods: `forCircle`, `forSquare`, `forTrans`, and `forUnion`.

Correct, but unfortunately we have to add three more words to make this explicit.

```java
class UnionHasPt^V
    extends HasPt^V
    implements UnionVisitor^T {
        UnionHasPt^V(Point^D _p) {
            super(_p); }

        public boolean forUnion(Shape^D s,Shape^D t) {
            if (s.accept(this))
                return true;
            else
                return t.accept(this); }
    }
```

The first two additional words have an obvious meaning. They explicitly say that this visitor provides the services of `UnionVisitor^T`. And, as we have said before, the addition of `public` is necessary, because this visitor implements an interface.

Good try. Let's see whether it works. What should be the value of

```java
new Trans(
    new CartesianPt(3,7),
    new Union(
        new Square(10),
        new Circle(10)))
    .accept(
    new UnionHasPt^V(
        new CartesianPt(13,17)))?
```

We know how `forTrans` works, so we're really asking whether

```java
new CartesianPt(10,10)
```

is inside the Union shape.

So?

Which means that we're asking whether

```java
new CartesianPt(10,10)
```

is inside of

```java
new Square(10)
```

or inside of

```java
new Circle(10),
```
Okay. And what should be the answer? It should be true.

Let’s see whether the value of
  new Trans(
    new CartesianPt(3,7),
    new Union(
      new Square(10),
      new Circle(10)))
  .accept(
    new UnionHasPtV(
      new CartesianPt(13,17)))

is true?

Usually we start by determining what kind of object we are working with.

And? It’s a Shape\textsuperscript{D}.

How did we construct this shape? With Trans.

Which method should we use on it? \textit{forTrans}, of course.

Where is \textit{forTrans} defined? It is defined in HasPtV.

So what should we do now? We should determine the value of
  new Union(
    new Square(10),
    new Circle(10))
  .accept(
    new HasPtV(
      new CartesianPt(10,10))).

What type of object is
  new Union(
    new Square(10),
    new Circle(10))?

It’s a Shape\textsuperscript{D}.
How did we construct this \texttt{Shape}?

With \texttt{Union}.

So which method should we use on it?

\texttt{for\_Union}, of course.

How do we find the appropriate \texttt{for\_Union} method?

In \texttt{accept}, which is defined in \texttt{Union}, we confirm that

\begin{verbatim}
new HasPtV{
    new CartesianPt(10,10))
\end{verbatim}

is a \texttt{UnionVisitor} and then invoke its \texttt{for\_Union}.

Is an instance of \texttt{HasPtV} a \texttt{UnionVisitor}?

No!

Does it contain a method \texttt{for\_Union}?

No!

Then what is the value of

\begin{verbatim}
new Union(
    new Square(10),
    new Circle(10))
    .accept(
    new HasPtV(
        new CartesianPt(10,10))))
\end{verbatim}

It doesn't have a value. We are stuck.\footnote{A Java program raises a \texttt{RuntimeException}, indicating that the attempt to confirm the \texttt{UnionVisitor\_Visitor} of the object failed. More specifically, we would see the following when running the program:

\begin{verbatim}
java.lang.ClassCastException: UnionHasPtV
at Union.accept(\ldots java\ldots)
at UnionHasPtV.forextra(\ldots java\ldots)
at Tree.accept(\ldots java\ldots)
\end{verbatim}}

What do we do next?

Relax. Read a novel. Take a nap.

Which of those is best?

You guessed it: whatever you did is best.

We should have prepared this extension in a better way.

\footnote{How could we have done that?}
Here is the definition of $\text{HasPt}^V$ that we should have provided if we wanted to extend it without making changes.

```
class HasPt$^V$ implements ShapeVisitor$^T$ {
  Point$^D$ p;
  HasPt$^V$(Point$^D$ _p) {
    p = _p;
  }

  ShapeVisitor$^T$ newHasPt(Point$^D$ p) {
    return new HasPt$^V$(p);
  }

  public boolean forCircle(int r) {
    return p.distanceToO() <= r;
  }

  public boolean forSquare(int s) {
    if (p.x <= s)
      return (p.y <= s);
    else
      return false;
  }

  public boolean forTrans(Point$^D$ q, Shape$^D$ s) {
    return s.accept(newHasPt(p.minus(q)));}
}
```

How does this definition differ from the previous one?

Good. What does $\text{newHasPt}$ produce? A new $\text{ShapeVisitor}^T$, as its interface implies.

And how does it produce that? By constructing a new instance of $\text{HasPt}^V$.

Is $\text{newHasPt}$ like a constructor? It is virtually indistinguishable from a constructor, which is why it is above the line that separates constructors from methods.

---

*Be a Good Visitor*
Does that mean the new definition of \texttt{HasPt}\$^{1}$ and the previous one are really the same?\footnote{A functional programmer would say that \texttt{newHasPt} and \texttt{HasPt}\$^{V}$ are \textit{\textit{\eta}}-equivalent.}

They are mostly indistinguishable. Both \texttt{forTrans}, the one in the previous and the one in the new definition of \texttt{HasPt}\$^{V}$, produce the same values when they consume the same values.

Very well. But how does that help us with our problem?\footnote{That’s not obvious.}

Can we override \texttt{newHasPt} when we extend \texttt{HasPt}\$^{V}$?\footnote{Yes, we can override any method that we wish to override.}

Let’s override \texttt{newHasPt} in \texttt{UnionHasPt}\$^{V}$.\footnote{When we override it, we need to make sure it produces a \texttt{ShapeVisitor}\$^{7}$.}

That’s true. Should it produce a \texttt{HasPt}\$^{V}$ or a \texttt{UnionHasPt}\$^{V}$?\footnote{The latter. Then \texttt{forTrans} in \texttt{HasPt}\$^{V}$ keeps producing a \texttt{UnionHasPt}\$^{V}$, if we start with a \texttt{UnionHasPt}\$^{V}$.}

Good answer. Should we repeat it?\footnote{Let’s just reread it.}

\textbf{The Ninth Bit of Advice}

\textit{If a datatype may have to be extended, be forward looking and use a constructor-like (overridable) method so that visitors can be extended, too.}
And that’s exactly what we need. Revise the definition of UnionHasPtV.\textsuperscript{1} Here it is.

```java
class UnionHasPtV
    extends HasPtV
    implements UnionVisitor2 {
    UnionHasPtV(PointD p) {
        super(p); }
    ShapeVisitor2 newHasPt(PointD p) {
        return new UnionHasPtV(p); }

    public boolean forUnion(ShapeD s, ShapeD t) {
        if (s.accept(this))
            return true;
        else
            return t.accept(this); }
}
```

\textsuperscript{1} The is an instance of the \textit{factory method} pattern [4].

If we assemble all this into one picture, what do we get?

A drawing that helps our understanding of the relationships among the classes and interfaces.

What does the box mean? Everything outside of the box is what we designed originally and considered to be unchangeable; everything inside is our extension.

\textit{Be a Good Visitor}
Does the picture convey the key idea of this chapter?

No. It does not show the addition of a constructor-like method to HasPt and how it is overridden in UnionHasPt.

Is anything missing?

Square, but that's okay.

Let's see whether this definition works.

What is the value of

new Trans(
    new CartesianPt(3,7),
    new Union(
        new Square(10),
        new Circle(10)))
  .accept(
    new UnionHasPt(
        new CartesianPt(13,17)))?

We remember that the shape was built with Trans.

Which method should we use on it?

forTrans, of course.

Where is forTrans defined?

It is defined in HasPt.

So what should we do now?

We should determine the value of

new Union(
    new Square(10),
    new Circle(10))
  .accept(
    this.newHasPt(
        new CartesianPt(10,10))).

What is this?

The current visitor, of course.

And how does that work?

We determine the value of

this.newHasPt(
    new CartesianPt(10,10))
and then use accept for the rest.
And what do we create?

What is the value of
new Union(
new Square(10),
new Circle(10))
.accept(
new UnionHasPtV(
new CartesianPt(10,10)))?

How do we do that?

Is it true?

Are we happy now?

Is it good to have extensible definitions?

Very well. Does this mean we can put
together flexible and extensible definitions if
we use visitor protocols with these
constructor-like methods?

And why is that?

Are you hungry yet?

The new UnionVisitorV:
new UnionHasPtV(
new CartesianPt(10,10)).

UnionHasPtV also satisfies the interface
ShapeVisitorV, so now we can invoke the
forUnion method.

We first determine the value of
new Square(10)
.accept(
new UnionHasPtV(
new CartesianPt(10,10))).
If it is true, we're done.

It is. So we're done and we got the value we
expected.

Ecstatic.

Yes. People should use extensible definitions
if they want their code to be used more than
once.

Yes, we can and should always do so.

Because no program is ever finished.

Are our meals ever finished?

Be a Good Visitor
10. The State of Things to Come
Have you ever wondered where the pizza pies come from? You should have, because someone needs to make the pie.

Here is our pizza pieman.

```java
class Pieman<M implements Pieman<T {
    PieD p = new Bot();
    public int addTop(Object t) {
        p = new Top(t,p)
    ;
        return occTop(t); }
    public int remTop(Object t) {
        p = (PieD)p.accept(new RemV(t))
    ;
        return occTop(t); }
    public int substTop(Object n, Object o) {
        p = (PieD)p.accept(new SubstV(n,o))
    ;
        return occTop(n); }
    public int occTop(Object o) {
        return ((Integer)p.accept(new OccursV(o))
            .intValue(); }
}
```

M - This superscript is a reminder that the class manages a data structure. Lower superscripts when you enter this kind of definition in a file: Pieman.

How so? Haven’t we seen PieD, Top, and Bot before?

And haven’t we seen visitors like RemV, SubstV, and OccursV for various datatypes? Yes, yes. But what are the stand-alone semicolons about?

Let’s not worry about them for a while. Fine, but they are weird.

The State of Things to Come
Here is the interface for Pieman$^M$.

```java
interface Pieman$^M$ {
    int addTop(Object t);
    int remTop(Object t);
    int substTop(Object n, Object o);
    int occTop(Object o);
}
```

Isn’t it missing $p$?

We don’t specify fields in interfaces. And in any case, we don’t want anybody else to see $p$.

Whatever.

Here are PieVisitor$^T$ and Pie$^P$.

```java
interface PieVisitor$^T$ {
    Object forBot();
    Object forTop(Object t, Pie$^P$ r);
}

abstract class Pie$^P$ {
    abstract
    Object accept(PieVisitor$^T$ ask);
}
```

Define Bot and Top.

```java
class Bot extends Pie$^P$ {
    Object accept(PieVisitor$^T$ ask) {
        return ask.forBot();
    }
}
```

They are very familiar.

```java
class Top extends Pie$^P$ {
    Object t;
    Pie$^P$ r;
    Top(Object .t, Pie$^P$ .r) {
        t = .t;
        r = .r;
    }
    Object accept(PieVisitor$^T$ ask) {
        return ask.forTop(t, r);
    }
}
```
Here is \text{Occurs}^\text{V}. It counts how often some topping occurs on a pie.

```java
class Occurs\textsuperscript{V} implements PieVisitor\textsuperscript{T} {
    Object o;
    Occurs\textsuperscript{V}(Object \_\_o) {
        o = \_\_o;
    }

    public Object forBot() {
        return new Integer(0);
    }

    public Object forTop(Object t, Pie\textsuperscript{D} r) {
        if (t.equals(o))
            return new Integer(((Integer)
                (r.accept(this))
                .intValue()
                + 1);
        else
            return r.accept(this);
    }
}
```

And this little visitor substitutes one good topping for another.

```java
class Subst\textsuperscript{V} implements PieVisitor\textsuperscript{T} {
    Object n;
    Object o;
    Subst\textsuperscript{V}(Object \_\_n, Object \_\_o) {
        n = \_\_n;
        o = \_\_o;
    }

    public Object forBot() {
        return new Bot();
    }

    public Object forTop(Object t, Pie\textsuperscript{D} r) {
        if (o.equals(t))
            return new Top(n, (Pie\textsuperscript{D}) r.accept(this));
        else
            return new Top(t, (Pie\textsuperscript{D}) r.accept(this));
    }
}
```

Great! Now we have almost all the visitors for our pieman. Define \text{Rem}^\text{V}, which removes a topping from a pie.

```java
class Rem\textsuperscript{V} implements PieVisitor\textsuperscript{T} {
    Object o;
    Rem\textsuperscript{V}(Object \_\_o) {
        o = \_\_o;
    }

    public Object forBot() {
        return new Bot();
    }

    public Object forTop(Object t, Pie\textsuperscript{D} r) {
        if (o.equals(t))
            return r.accept(this);
        else
            return new Top(t, (Pie\textsuperscript{D}) r.accept(this));
    }
}
```

We remember that one, too.

---

*The State of Things to Come* 163
Now we are ready to talk. What is the value of

\texttt{new Pieman}^{M}().occ\texttt{Top(new Anchovy())}?

\texttt{11} We first create a Pieman^{M} and then ask how many anchovies occur on the pie.

\texttt{Which pie?} \hspace{1cm} \texttt{12} The pie named \texttt{p} in the new Pieman^{M}.

\texttt{And how many anchovies are on that pie?} \hspace{1cm} \texttt{13} None.

\texttt{And what is the value of} \hspace{1cm} \texttt{14} That’s where those stand-alone semicolons come in again. They were never explained.

\texttt{new Pieman}^{M}().\texttt{addTop(new Anchovy())}? 

\texttt{True. If we wish to determine the value of} \hspace{1cm} \texttt{15} Yes, we must understand that. There is no number \texttt{x} in the world for which \texttt{x = x + 1},

\texttt{new Pieman}^{M}().\texttt{addTop(new Anchovy())}, \hspace{1cm} \texttt{so why should we expect there to be a Java \texttt{p} such that} \texttt{p = new Top(new Anchovy().p)}?

\texttt{we must understand what} \hspace{1cm} \texttt{16} So what does it mean?

\texttt{p = new Top(new Anchovy().p)} ;
\texttt{return occ\texttt{Top(new Anchovy())}}

\texttt{That’s right. But that’s what happens when} \hspace{1cm} \texttt{17} And the change is that \texttt{p} has a new topping, right?

\texttt{you have one too many double espressos.}

\texttt{Here it means that \texttt{p} changes and that future} \hspace{1cm} \texttt{18} Does it begin below the stand-alone

\texttt{references to \texttt{p} reflect the change.} \hspace{1cm} \texttt{semicolon?}

\texttt{When does the future begin?} \hspace{1cm} \texttt{19} It produces the number of anchovies on \texttt{p}.

\texttt{That’s precisely what a stand-alone} \hspace{1cm} \texttt{return occ\texttt{Top(new Anchovy())}} \texttt{produces?}

\texttt{semicolon means. Now do we know what}
And how many are there?  

We added one, so the value is 1.

And now what is the value of  
\[
\text{new Pieman}^{<1}.addTop(\text{new Anchovy}())?
\]

It's 2, isn't it?

No, it's not. Take a close look. We created a  
\text{new pieman}, and that pieman added only one anchovy to his \text{p}.

Oh, isn't there a way to place several requests with the same pieman?

Yes, there is. Take a look at this:  
\[
\text{Pieman}^7 y = \text{new Pieman}^{<1}()
\]

Okay, \(y\) stands for some pieman.

What is the value of  
\[
y.addTop(\text{new Anchovy}())?
\]

1. We know that.

And now what is the value of  
\[
y.subst\{Top(\text{new Tuna}(), \text{new Anchovy}())\}
\]

Still 1. According to the rules of semicolon and '=', this replaces all anchovies on \(p\) with tunas, changes \(p\), and then counts how many tunas are on \(p\).

Correct. So what is the value of  
\[
y.occ\{Top(\text{new Anchovy}())\}
\]

0, because \(y's\) pie no longer contains any anchovies.

Very good. And now take a look at this:  
\[
\text{Pieman}^7 yy = \text{new Pieman}^{<1}()
\]

What is the value of  
\[
yy.addTop(\text{new Anchovy}()) : 
\]
\[
yy.addTop(\text{new Anchovy}()) : 
\]
\[
yy.addTop(\text{new Salmon}()) : 
\]

What are the \[\ldots\] doing at the end?

*The State of Things to Come*
Because this is only half of what we want to look at. Here is the other half:

```java
yy.addTop(new Tuna())
;
yy.addTop(new Tuna())
;
yy.substTop(new Tuna(), new Anchovy())?
```

4. First we add two anchovies, then a salmon, and two tunas. Then we substitute the two anchovies by two tunas. So yy’s pie contains four tunas.

And what is the value of

```java
yy.remTop(new Tuna())
```

after we are through with all that?

29. It’s 0, because remTop first removes all tunas and then counts how many there are left.

Does that mean remTop always produces 0? 29. Yes, it always does.

Now what is the value of

```java
yy.occTop(new Salmon())?
```

31. 1.

And how about

```java
y.occTop(new Salmon())?
```

0, because y and yy are two different piemen.

Is yy the same pieman as before? 33. No, it changed.

So is it the same one? 34. When we eat a pizza pie, we change, but we are still the same.

When we asked yy to substitute all anchovies by tunas, did the pie change?

35. The p in yy changed, nothing else.

Does that mean that anybody can write

```java
yy.p = new Bot()
```

and thus change a pieman like yy?

36. No, because yy’s type is Pieman^, p isn’t available. Only addTop, remTop, substTop, and occTop are visible.
Isn’t it good that we didn’t include \( p \) in \( \text{Pieman}^T \)?

Yes, with this trick we can prevent others from changing \( p \) (or parts of \( p \)) in strange ways. Everything is clear now.

Clear like soup?

Just like chicken soup.

Can we define a different version of \( \text{Subst}^V \) so that it changes toppings the way a pieman changes his pies?

We can’t do that yet.

And that’s what we discuss next. Do you need a break?

No, a cup of coffee will do.

Compare this new \( \text{PieVisitor}^T \) with the first one in this chapter.

```java
interface \text{PieVisitor}^T \{ 
    Object \text{forBot}(\text{Bot that}); 
    Object \text{forTop}(\text{Top that}); 
}\}
```

It isn’t all that different. A \( \text{PieVisitor}^T \) must still provide two methods: \( \text{forBot} \) and \( \text{forTop} \), except that the former now consumes a \text{Bot} and the latter a \text{Top}.

True. Here is the unchanged datatype.

```java
abstract class \text{Pie}^T \{ 
    abstract 
    Object \text{accept}(\text{PieVisitor}^T \text{ ask}); 
}\}
```

```java
class \text{Bot} extends \text{Pie}^T \{ 
    Object \text{accept}(\text{PieVisitor}^T \text{ ask}) \{ 
        \text{return \text{ask forBot(this);}}; 
    \}
}\}
```

Define the \text{Bot} variant.

Is it? Why does it use \textbf{this}?

We only have one instance of \text{Bot} when we use \text{forBot}, namely \textbf{this}, so \text{forBot} is clearly supposed to consume \textbf{this}.

*The State of Things to Come* 167
That’s progress. And that’s what happens in Top, too.

```java
class Top extends PieD {
    Object t;
    PieD r;
    Top(Object .t,PieD .r) {
        t = .t;
        r = .r;
    }
    Object accept(PieVisitor2 ask) {
        return ask.forTop(this);
    }
}
```

Modify this version of OccursV so that it implements the new PieVisitor2.

```java
class OccursV implements PieVisitor2 {
    Object a;
    OccursV(Object .a) {
        a = .a;
    }
    public Object forBot() {
        return new Integer(0);
    }
    public Object forTop(Object t,PieD r) {
        if (t.equals(a))
            return new Integer(((Integer)
                (r.accept(this)))
                .intValue()
                + 1);
        else
            return r.accept(this);
    }
}
```

The forBot method basically stays the same, but forTop changes somewhat.

```java
class OccursV implements PieVisitor2 {
    Object a;
    OccursV(Object .a) {
        a = .a;
    }
    public Object forBot(Bot that) {
        return new Integer(0);
    }
    public Object forTop(Bot that) {
        if (that.t.equals(a))
            return new Integer(((Integer)
                (that.r.accept(this)))
                .intValue()
                + 1);
        else
            return that.r.accept(this);
    }
}
```

How does forBot change?

```java
It now consumes a Bot, which is why we had to add (Bot that) behind its name.
```
How does \textit{forTop} change? It no longer receives the field values of the corresponding \textit{Top}. Instead it consumes the entire object, which makes the two fields available as \textit{that.t} and \textit{that.r}.

And? With that, we can replace the fields \textit{t} and \textit{r} with \textit{that.t} and \textit{that.r}.

Isn’t that easy? This modification of \textit{Ocurs}\textsuperscript{V} certainly is.

Then try \textit{Rem}\textsuperscript{V}. It’s easy; we use the same trick.

```java
class Rem\textsuperscript{V} implements PieVisitor\textsuperscript{T} {
    Object o;
    Rem\textsuperscript{V}(Object _o) {
        o = _o;
    }

    public Object forBot(Bot that) {
        return new Bot();
    }
    public Object forTop(Top that) {
        if (o.equals(that.t))
            return that.r.accept(this);
        else
            return new Top(that.t,
            (Pie\textsuperscript{T})that.r.accept(this));
    }
}
```

Do we need to do \textit{Subst}\textsuperscript{V}? Not really. It should be just like \textit{Rem}\textsuperscript{V}.

And indeed, it is. Happy now? So far, so good. But what’s the point of this exercise?

Oh, Point\textsuperscript{P}s? They will show up later. Seriously.
Here is the point. What is new about this version of Subst\textsuperscript{V}?

```java
class Subst\textsuperscript{V} implements PieVisitor\textsuperscript{F} {
    Object n;
    Object o;
    Subst\textsuperscript{V}(Object _n, Object _o) {
        n = _n;
        o = _o;
    }

    public Object forBot(Bot that) {
        return that;
    }

    public Object forTop(Top that) {
        if (o.equals(that.t)) {
            that.t = n
            ;
            that.r.accept(this)
            ;
            return that;
        } else {
            that.r.accept(this)
            ;
            return that;
        }
    }
}
```

Don’t they say “no news is good news?”

Does this saying apply here, too?

Yes, because we want to define a version of Subst\textsuperscript{V} that modifies toppings without constructing a new pie.

That’s a way of putting it.

What do the methods of Subst\textsuperscript{V} always return?

They always return that, which is the object that they consume.

So how do they substitute toppings?

By changing the that before they return it. Specifically, they change the t field of that to n when it equals o.
What?

Correct. And from here on, that.t holds the new topping. What is that.r.accept(this) about?

In the previous Subst^V, r.accept(this) created a new pie from r with all toppings appropriately substituted. In our new version, that.r.accept(this) modifies the pie r so that below the following semicolon it contains the appropriate toppings.

Is there anything else to say about the new Subst^V?

Not really. It does what it does, which is what we wanted.¹

¹ This is a true instance of the visitor pattern [4]. What we previously called “visitor” pattern instances were simple variations on the theme.

Do we have to change Pieman^M?

No, we didn’t change what the visitors do. we only changed how they do things.

Is it truly safe to modify the toppings of a pie?

Yes, because the Pieman^M manages the toppings of p, and nobody else sees p.

Can we do LtdSubst^V now without creating new instances of LtdSubst^V or Top?

Now that’s a piece of pie.

The Tenth Bit of Advice

When modifications to objects are needed, use a class to insulate the operations that modify objects. Otherwise, beware the consequences of your actions.

The State of Things to Come
Here is a true dessert. It will help us understand what the point of state is.

```java
abstract class PointD {
    int x;
    int y;
    PointD(int _x, int _y) {
        x = _x;
        y = _y;
    }

    boolean closerToO(PointD p) {
        return distanceToO() <= p.distanceToO();
    }

    PointD minus(PointD p) {
        return new CartesianPt(x - p.x, y - p.y);
    }

    abstract int distanceToO();
}
```

The datatype has three extensions.

```java
class CartesianPt extends PointD {
    CartesianPt(int _x, int _y) {
        super(_x, _y);
    }

    int distanceToO() {
        return Math.sqrt(x * x + y * y);
    }
}
```

```java
class ManhattanPt extends PointD {
    ManhattanPt(int _x, int _y) {
        super(_x, _y);
    }

    int distanceToO() {
        return x + y;
    }
}
```

```java
class ShadowedManhattanPt extends ManhattanPt {
    int Δx;
    int Δy;
    ShadowedManhattanPt(int _x, int _y, int _Δx, int _Δy) {
        super(_x, _y);
        Δx = _Δx;
        Δy = _Δy;
    }

    int distanceToO() {
        return super.distanceToO() + Δx + Δy;
    }
}
```

Aren’t we missing a variant? Yes, we are missing ShadowedCartesianPt.

172

Chapter 10
Good enough. We won’t need it. Here is one point:

```java
new ManhattanPt(1,4).
```

If this point represents a child walking down the streets of Manhattan, how do we represent his movement?

Yes. Add to PointD the method `moveBy`, which consumes two ints and changes the fields of a point appropriately.

```java
abstract class PointD {
  int x;
  int y;
  PointD(int _x, int _y) { 
    x = _x;
    y = _y;
  }

  boolean closerToO(PointD p) {
    return distanceToO() <= p.distanceToO();
  }

  PointD minus(PointD p) {
    return new CartesianPt(x - p.x, y - p.y);
  }

  int moveBy(int Δx, int Δy) { 
    x = x + Δx;
    y = y + Δy;
    return distanceToO();
  }

  abstract int distanceToO();
}
```

Now we know how to do this.

Let `ptChild` stand for

```java
new ManhattanPt(1,4).
```

What is the value of `ptChild.distanceToO()`?
What is the value of
\( ptChild.moveBy(2,8) \)?

15.

Good. Now let’s watch a child with a
helium-filled balloon that casts a shadow.
Let \( ptChildBalloon \) be
\( \text{new ShadowedManhattanPt(1.4,1.1)} \).
What is the value of
\( ptChildBalloon.distanceToO() \)?

7.

What is the value of
\( ptChildBalloon.moveBy(2,8) \)?

17, of course.

Did the balloon move, too?

Yes, it just moved along as we moved the point.

Isn’t that powerful?

It sure is. We added one method, used it,
and everything moved.

The more things change, the cheaper our
desserts get.

Yes, but to get to the dessert, we had to
work quite hard.

Correct but now we are through and it is
time to go out and to celebrate with a grand
dinner.

Don’t forget to leave a tip.
Commencement
You have reached the end of your introduction to computation with classes, interfaces, and objects. Are you now ready to tackle a major programming problem? Programming requires two kinds of knowledge: understanding the nature of computation, and discovering the lexicon, features, and idiosyncrasies of a particular programming language. The first of these is the more difficult intellectual task. If you understand the material in this book, you have mastered that challenge. Still, it would be well worth your time to develop a fuller understanding of all the capabilities in Java—this requires getting access to a running Java system and mastering those idiosyncrasies. If you want to understand Java and object-oriented systems in greater depth, take a look at the following books:

References


This is for the loyal Schemers and MLers.

```java
interface T ^T { 
  o -> o ^T apply(T ^T x);
}

interface o -> o ^T { 
  apply(Object x);
}

interface oo -> oo ^T { 
  o -> o ^T apply(oo -> oo ^T x);
}

interface oo -> oo oo ^T { 
  o -> o ^T apply(oo -> oo oo ^T x);
}

class Y implements oo -> oo oo ^T { 
  public o -> o ^T apply(oo -> oo oo ^T f) { 
    return new H(f).apply(new H(f));
  }
}

class H implements T ^T { 
  o -> o ^T f;
  H(oo -> oo ^T f) {
    f = f;
  }
  public o -> o ^T apply(T ^T x) { 
    return f.apply(new G(x));
  }
}

class G implements o -> o ^T { 
  T ^T x;
  G(T ^T .x) {
    x = x;
  }
  public Object apply(Object y) { 
    return (x.apply(x)).apply(y);
  }
}
```

No, we wouldn't forget factorial.

```java
class MkFact implements oo -> oo ^T { 
  public o -> o ^T apply(o -> o ^T fact) { 
    return new Fact(fact);
  }
}

class Fact implements o -> o ^T { 
  o -> o ^T fact;
  Fact(o -> o ^T .fact) {
    fact = fact;
  }
  public Object apply(Object i) { 
    int ini = ((Integer)i).intValue();
    if (ini == 0)
      return new Integer(1);
    else
      return new Integer(
        ini
        *
        ((Integer)
          fact.apply(new Integer(ini - 1)))
        .intValue());
  }
}
```
Copyrighted material

Index
Add, 124
addTop, 161
Anchovy, 43, 53, 64, 65, 69, 72
Apple, 100
Base, 10
bHasFruit\textsuperscript{v}, 104
bIsFlat\textsuperscript{v}, 102
bIsSplit\textsuperscript{v}, 162, 163
Bot, 69, 78, 83, 85, 86, 91, 93, 162, 167
Brass, 29
Bud, 100, 101, 107, 110, 112
bTreeVisitor\textsuperscript{v}, 101
CartesianPt, 5, 13, 37, 38–40, 139, 172
Cheese, 43, 53, 64, 65
Circ\textoe, 143
closerToO, 14, 38–40, 139, 172, 173
Const, 119
Copper, 29
Crust, 43, 53, 64, 65
Dagger, 29
diff, 124
Diff, 119
distanceToO, 14, 38–40, 139, 141, 142, 172
Empty, 124
equals, 72, 79, 100
Eval\textsuperscript{D}, 130
Exp\textsuperscript{D}, 119
ExpVisitor\textsuperscript{D}, 118
Fig, 100
Fish\textsuperscript{D}, 69, 72
Flat, 100, 101, 108, 110, 112
Fruit\textsuperscript{D}, 100
Gold, 29
HasPt\textsuperscript{v}, 147, 155
Holder, 28, 36
iHeight\textsuperscript{v}, 188
IntEval\textsuperscript{v}, 120, 122, 131
iOcurs\textsuperscript{v}, 111
IsFlat\textsuperscript{v}, 113
IsSplit\textsuperscript{v}, 113
isVegetarian, 25, 27
isVegetarian\textsuperscript{v}, 63
isVeggie, 30, 31, 36
iTreeVisitor\textsuperscript{v}, 107
Kebab\textsuperscript{D}, 28, 36
Lamb, 16, 27, 28, 36, 59, 62
Layer\textsuperscript{D}, 10
Lemon, 100
LtdSubst\textsuperscript{v}, 95, 97, 132–136
ManhattanPt, 5, 13, 38, 39, 139, 172
mem, 124
minus, 139
moveBy, 173
newHasPt, 155, 157
Num\textsuperscript{D}, 7, 79
occTop, 161
Ocurs\textsuperscript{v}, 114, 163, 168
Olive, 43, 53, 64, 65
OneMoreThan, 7, 79
Onion, 16, 27, 28, 36, 59, 62
onlyOnions, 18, 27
OnlyOnions\textsuperscript{v}, 57, 59
Peach, 100
Pear, 100
Pepper, 4, 37
Pie\textsuperscript{D}, 69, 78, 83, 85, 86, 91, 93, 162, 167
PieVisitor\textsuperscript{D}, 92, 162, 167
Pieman\textsuperscript{T}, 162
Pieman\textsuperscript{M}, 161
Pizza\textsuperscript{D}, 43, 53, 63–65
Pizza\textsuperscript{v}, 29
plus, 124
Plus, 119
Point\textsuperscript{D}, 5, 13, 40, 139, 172, 173
prod, 124
Prod, 119
| Radish, 28, 36 |
| Rem.4, 45, 49, 70 |
| RemA°, 66, 71, 73 |
| RemFish°, 74 |
| Rem°, 77, 87, 93, 163, 169 |
| RemInt°, 76 |
| remTop, 161 |
| Rod°, 29 |
| Sabre, 29 |
| Sage, 5 |
| Salmon, 69, 72 |
| Salt, 4 |
| Sausage, 43, 53, 63–65 |
| Seasoning°, 4 |
| SetD, 123 |
| SetEval°, 126, 131 |
| ShadowedCartesianPt, 141, 142 |
| ShadowedManhattanPt, 139, 172 |
| Shallot, 28, 36 |
| Shape°, 143 |
| ShapeVisitor°, 144 |
| Shish°, 16, 27, 59, 61 |
| Shrimp, 28, 36 |
| Silver, 29 |
| Skewer, 16, 27, 59, 62 |
| Slice, 10 |
| Spinach, 54 |
| Split, 100, 101, 108, 110, 112 |
| Square, 143 |
| subABC, 52, 53 |
| SubAbC°, 66 |
| SubsD, 133, 134 |
| SubsFish°, 82 |
| Subs°, 83, 87, 89, 94, 132–136, 163, 170 |
| SubsInt°, 82 |
| subSTop, 161 |
| Sword, 29 |
| Thyme, 5 |
| Tomato, 16, 27, 28, 38, 50, 62 |
| Top, 69, 78, 83, 85, 86, 91, 93, 162, 168 |
| topAuC, 50, 53 |
| topAwC°, 66 |
| Trans, 144 |
| Tree°, 100, 101, 107, 109, 112 |
| TreeVisitor°, 112 |
| tSubst°, 111 |
| tTreeVisitor°, 109 |
| Tuna, 69, 72 |
| Union, 149, 150 |
| UnionHasPt°, 151, 152, 157 |
| UnionVisitor°, 150 |
| whatHolder, 33–36 |
| Wood, 29 |
| Zero, 7, 79 |
| Zucchini, 37, |
Java is a new object-oriented programming language for programming the Internet and intelligent appliances. In a very short time it has become one of the most widely used programming languages in education as well as for commercial applications. Design patterns, which have moved object-oriented programming to a new level, provide programmers with a language to communicate with others about their designs. As a result, programs become more readable, reusable, and easily extensible.

Matthias Felleisen and Daniel Friedman use a small subset of Java to introduce pattern-directed program design. With their usual clarity and flair, they gently guide readers through the fundamentals of object-oriented programming and pattern-based design. Readers new to programming, as well as those with some background, will enjoy their learning experience as they work their way through Felleisen and Friedman's lessons.

Matthias Felleisen is Professor of Computer Science at Rice University. Daniel P. Friedman is Professor of Computer Science at Indiana University. They are the authors of The Little Schemer, The Seasoned Schemer, and The Little MLer.

"This is a book of 'why,' not 'how.' If you are interested in the nature of computation and curious about the very idea behind object orientation, this book is for you. This book will engage your brain (if not your tummy). Through its sparkling interactive style, you will learn about three essential OO concepts: interfaces, visitors, and factories. A refreshing change from the 'yet another Java book' phenomenon. Every serious Java programmer should own a copy"—Gary McGraw, Ph.D., Research Scientist at Reliable Software Technologies.

The MIT Press
Massachusetts Institute of Technology • Cambridge, Massachusetts 02142
http://mitpress.mit.edu • FELTP 0-262-56115-8