<table>
<thead>
<tr>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foreword ix</td>
</tr>
<tr>
<td>Preface xi</td>
</tr>
<tr>
<td>Experimenting with Java xiii</td>
</tr>
<tr>
<td>1. Modern Toys 3</td>
</tr>
<tr>
<td>2. Methods to Our Madness 13</td>
</tr>
<tr>
<td>3. What’s New? 43</td>
</tr>
<tr>
<td>4. Come to Our Carousel 57</td>
</tr>
<tr>
<td>5. Objects Are People, Too 69</td>
</tr>
<tr>
<td>6. Boring Protocols 85</td>
</tr>
<tr>
<td>7. Oh My! 99</td>
</tr>
<tr>
<td>8. Like Father, Like Son 117</td>
</tr>
<tr>
<td>9. Be a Good Visitor 139</td>
</tr>
<tr>
<td>10. The State of Things to Come 161</td>
</tr>
<tr>
<td>Commencement 177</td>
</tr>
<tr>
<td>Index 178</td>
</tr>
</tbody>
</table>

Copyrighted material
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 Filman, Robby Findler, Steve Ganz, Paul Graunke, 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 SDT^X. 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.

**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 *italic*. Basic data, including numbers, booleans, and constructors introduced via datatypes are set in *sans serif*. Keywords, e.g., *class*, *abstract*, *return* and *interface* are in *boldface*. When you experiment, you may ignore the typefaces but not the related framenotes. To highlight this role of typefaces, the programs in framenotes are set in a *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.

---

*Bon appétit!*

Matthias Felleisen
Daniel P. Friedman

xii
Here are some hints on how to experiment with Java:\footnote{See Arnold and Gosling [1] 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 `distanceTo0` 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.distanceTo0() );
    }
}
```
If you wish to experiment with a sequence of expressions that modify $y$, as in chapter 10, e.g.,

\[
    y_1 = \ldots \ldots \; ;
\]
\[
    y_2 = \ldots \ldots \; ;
\]
\[
    y_3 = \ldots \ldots \; ;
\]

replace \ldots with

\[
    y_1 = \ldots \ldots + \"\ln \" +
\]
\[
    y_2 = \ldots \ldots + \"\ln \" +
\]
\[
    y_3 = \ldots \ldots 
\]

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

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

4. Finally, compile the file and interpret the class $\text{Main}$. 

xiv
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?  

int.¹

In Java, int stands for “integer.”

Quick, think of another integer!  

How about 19?

What type of value is true?  

boolean.

What type of value is false?  

boolean.

Can you think of another boolean?  

No, that’s all there is to boolean.

What is int?  

A type.

What is 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.

Modern Toys
Draw the picture that characterizes the essential relationships among the following classes.

```java
abstract class SeasoningD {}  
class Salt extends SeasoningD {}  
class Pepper extends SeasoningD {}
```

D - This superscript is a reminder that the class is a datatype. Lower superscripts when you enter this kind of definition in a file: SeasoningD.

---

Yes. We say SeasoningD is a datatype, and Salt and Pepper are its variants.

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

Yes, in a way. Now, is new Salt() a SeasoningD?

Yes, it is, because new Salt() creates an instance of Salt, and every instance of Salt is also a SeasoningD.

And new Pepper()?

It's also a SeasoningD, because new Pepper() creates an instance of Pepper, and every instance of Pepper is also a SeasoningD.

What are abstract, class, and extends?

Easy:

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

Is there any other SeasoningD?

No, because only Salt and Pepper extend SeasoningD.1

---

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 `SeasoningD`. Have we seen a datatype like `SeasoningD` before?

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

Let's define more `SeasoningD`s.

```
class Thyme extends SeasoningD {}
class Sage extends SeasoningD {}
```

We can have lots of `SeasoningD`'s.

And then there were four.

```
21. We can have lots of `SeasoningD`'s.

22. And then there were four.

23. Yes.
```

What is a Cartesian point?

```
What is a Cartesian point?
```

It is basically a pair of numbers.

What is a point in Manhattan?

```
What is a point in Manhattan?
```

An intersection where two city streets meet.

How do `CartesianPt` and `ManhattanPt` differ from Salt and Pepper?

```
abstract class PointD {}

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

class ManhattanPt extends PointD {
    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 `javac` when you enter it in a file.
The underlined occurrences of **CartesianPt** and **ManhattanPt** introduce the constructors of the respective variants.

A constructor is used with **new** to create new instances of a **class**.

When we create a **CartesianPt** like this:
```java
new CartesianPt(2,3),
```
we use the constructor in the definition of **CartesianPt**.

So now we have created a **CartesianPt** whose \(x\) field is 2 and whose \(y\) field is 3. And because **CartesianPt extends Point**\(^7\), it is also a **Point**\(^7\).

Correct. Is this a **ManhattanPt**:
```java
new ManhattanPt(2,3)
```
Yes, and its \(x\) field is 2 and its \(y\) 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 **class** does not contain any fields, as in **Salt** and **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 **new**, always create objects without fields.

An **abstract** class is by definition incomplete, so **new** cannot create an instance from it.

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

```
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 Num\(^D\): new Zero()?

Obviously, just like new Salt() is a Seasoning\(^D\).

Is this a Num\(^D\): new OneMoreThan(new Zero())?

Yes, because OneMoreThan constructs a Num\(^D\) from a Num\(^D\), and every instance of OneMoreThan is also a Num\(^D\).

How does OneMoreThan do that?

We give it new Zero(), which is a Num\(^D\), and it constructs a new Num\(^D\).

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 Num\(^D\), which, at the moment, may be either a Zero or a OneMoreThan.
What is
new OneMoreThan(
    new OneMoreThan(
        new Zero()))?

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

What is
new OneMoreThan( 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 OneMoreThan(
    new Zero())
like
1?

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

And what is
new OneMoreThan(
    new OneMoreThan(
        new OneMoreThan(
            new OneMoreThan(
                new Zero()))))?

44 4.

Are there more Num\(^P\)s than booleans?

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\(^{32}\).
Easy: `new Zero()` is an instance of `Zero` and, by implication, is a `Num^D`, whereas 0 is an `int`. This makes it difficult to compare them, but we can compare them in our minds.

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

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

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

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^D`, which is extended by `Zero`. Indeed, it is of any type that `Num^D` 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.*
What do the following define?

```java
abstract class Layer{
}
```

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

```java
class Slice extends Layer{
    Layer l;
    Slice(Layer p) {
        l = p;
    }
}
```

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

What is `new Base(new Zero())`?

```java
new Base(new Zero())
```

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

And what is `new Base(new Salt())`?

```java
new Base(new Salt())
```

55 It also looks like an instance of `Base`. But how come both

```java
new Base(new Zero())
```

and

```java
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.

```java
new Object()
```

56 Hence, we can use both

```java
new Zero()
```

and

```java
new Salt()
```

for the construction of a `Base`, which requires an `Object`. 

Chapter 1
Is anything else an Object?  

57 We said that only things created with `new` are Objects.1

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

Correct. Is this a `Layer^D`: 

```
new Base(5)
```

58 5 is not created with `new`, so this must be nonsense.

Is this a `Layer^D`: 

```
new Base(false)
```

59 `false` is not created with `new`, so this must be nonsense, too.

Correct again! How about this `Layer^D`: 

```
new Base(new Integer(5))
```

60 This must mean that `Integer` creates an object from an `int`.

Guess how we create a `Layer^D` from `false`?  

61 Easy now: 

```
new Base(
    new Boolean(false)).
```

Is it confusing that we need to connect `int` with `Integer` and `boolean` with `Boolean`?  

62 Too much coffee does that.

Ready for more?  

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

abstract class Point\(^D\) {
    int \textit{x};
    int \textit{y};
    Point \{ \\
        \textit{x} = \textit{x};
        \textit{y} = \textit{y};
    \}
}

class CartesianPt extends Point\(^D\) {
    int \textit{x};
    int \textit{y};
    CartesianPt(int \textit{x},int \textit{y}) {
        \textit{x} = \textit{x};
        \textit{y} = \textit{y};
    }
}

class ManhattanPt extends Point\(^D\) {
    int \textit{x};
    int \textit{y};
    ManhattanPt(int \textit{x},int \textit{y}) {
        \textit{x} = \textit{x};
        \textit{y} = \textit{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 \texttt{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 \texttt{new CartesianPt(3,4)} from the origin?

5, which is \(\sqrt{3^2 + 4^2}\).
Write the methods `distanceToO` using `{`, `}`, `:`, `return`, `int`, `+`, `\sqrt{\cdot}`, and `^2`, 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.

Set what do the methods produce? `ints`, which represent the distances to the origin.

Here they are.

```java
abstract int distanceToO();

Point

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

CartesianPt

int distanceToO() {
    return x + y;
}

ManhattanPt
```

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

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

1 When you enter this in a file, use `(int)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 `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 `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?

```
abstract class ShishD {
    Shish
}

class Skewer extends ShishD {
    Skewer
}

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

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

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

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.
Construct a Shish\textsuperscript{D}.

Yes, every Skewer is also a Shish\textsuperscript{D}. How about another one?

And a third?

Are there only Onions on this Shish\textsuperscript{D}: 
\texttt{new Skewer()}

Are there only Onions on this Shish\textsuperscript{D}:
\texttt{new Onion(new Skewer())}

And how about:
\texttt{new Lamb(new Skewer())}

Is it true that
\texttt{new Onion(new Onion(new Onion(new Skewer())))}
contains only Onions?

And finally:
\texttt{new Onion(new Lamb(new Onion(new Skewer())))}?
Write the methods \textit{onlyOnions}\textsuperscript{1} using \{, \}, (, ), . . . ; \textit{true}, \textit{false}, \textit{return}, and \textbf{boolean}.

\textsuperscript{1} A better name for these methods would be \textit{nothingButOnions}.

And what do they produce? \textbf{booleans}.

Here are the methods.

\textbf{abstract boolean onlyOnions():}

\begin{itemize}
  \item \textbf{Shish} \textbf{boolean onlyOnions()} { \textbf{return} \textit{true}; }
  \item \textbf{Skewer} \textbf{boolean onlyOnions()} { \textbf{return} \textit{false}; }
  \item \textbf{Onion} \textbf{boolean onlyOnions()} { \textbf{return} \textit{s. onlyOnions();} }
  \item \textbf{Lamb} \textbf{boolean onlyOnions()} { \textbf{return} \textit{false}; }
  \item \textbf{Tomato} \textbf{boolean onlyOnions()} { \textbf{return} \textit{false}; }
\end{itemize}

Did you notice the labels in the boxes?

Good. How many methods have we defined? \textit{Five}, but the first one is \textbf{abstract}; the others are concrete.

Of course, you can't write these methods, yet. Okay, you deserve a lollipop for enduring this kind of question again.
Do abstract methods belong to the abstract class?  

Yes, because Shish\(^D\) contains an abstract method called onlyOnions that obligates each variant to define a matching, concrete method.

Does each variant of Shish\(^D\) contain a method called onlyOnions?  

32 Yes, because Shish\(^D\) contains an abstract method called onlyOnions that obligates each variant to define a matching, concrete method.

Is this always the case?  

33 Always.

What do these concrete methods consume?  

34 Nothing, just as the abstract method says.

What do these concrete methods produce?  

35 booleans, just as the abstract method says.

What is the value of 

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

36 true.

And how do we determine the value of 

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

37 We will need to pay attention to the method definitions.

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

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

38 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`? 39

What is the field `s` of the object
`new Onion(
    new Onion(
        new Skewer()))`? 40

It is  
`new Onion(
    new Skewer())`, isn't it?

Does `s` always stand for an `Onion`? 41

No. It has type `Shish`21, and it can stand for any variant of `Shish`: Skewer, Onion, Lamb, or Tomato.

Then what is `s.onlyOnions()`? 42

It should be  
`new Onion(
    new Skewer())
  .onlyOnions()`, right?

Why do we need to know the meaning of  
`new Onion(
    new Skewer())
  .onlyOnions()'? 43

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

How do we determine the answer for  
`new Onion(
    new Skewer())
  .onlyOnions()'? 44

Let's see.
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Which definition of <em>onlyOnions</em> must we use</td>
<td>This object is an instance of <em>Onion</em>, so we need to use the definition of <em>onlyOnions</em> that belongs to the <em>Onion</em> variant.</td>
</tr>
<tr>
<td>to determine the value of</td>
<td></td>
</tr>
<tr>
<td><em>new Onion</em></td>
<td></td>
</tr>
<tr>
<td><em>new Skewer()</em></td>
<td></td>
</tr>
<tr>
<td><em>.onlyOnions()</em></td>
<td></td>
</tr>
<tr>
<td>What follows the word <em>return</em> in the <em>onlyOnions</em> method in <em>Onion</em>?</td>
<td><em>s.onlyOnions()</em></td>
</tr>
<tr>
<td>What is the field <em>s</em> of the object <em>new Onion</em>?</td>
<td><em>new Skewer()</em></td>
</tr>
<tr>
<td><em>new Skewer()</em></td>
<td></td>
</tr>
<tr>
<td>?</td>
<td>It is <em>new Skewer()</em></td>
</tr>
<tr>
<td></td>
<td><em>.onlyOnions</em></td>
</tr>
<tr>
<td></td>
<td>just as we would have expected.</td>
</tr>
<tr>
<td>Then what is <em>s.onlyOnions()</em>?</td>
<td></td>
</tr>
<tr>
<td>Why do we need to know the meaning of <em>new Skewer()</em></td>
<td>Because the answer for <em>new Skewer()</em></td>
</tr>
<tr>
<td><em>.onlyOnions()</em></td>
<td>is also the answer for <em>new Onion</em></td>
</tr>
<tr>
<td><em>new Skewer()</em></td>
<td><em>.onlyOnions</em></td>
</tr>
<tr>
<td><em>new Skewer()</em></td>
<td>which in turn is the answer for <em>new Onion</em></td>
</tr>
<tr>
<td><em>.onlyOnions</em></td>
<td><em>new Onion</em></td>
</tr>
<tr>
<td></td>
<td><em>new Skewer()</em></td>
</tr>
<tr>
<td></td>
<td><em>.onlyOnions</em></td>
</tr>
<tr>
<td>How do we determine the answer for <em>new Skewer()</em>?</td>
<td>We need to determine one more time which version of <em>onlyOnions</em> we must use.</td>
</tr>
<tr>
<td><em>.onlyOnions()</em></td>
<td></td>
</tr>
</tbody>
</table>
Obviously.

new Skewer()

a Skewer?

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.

false, isn’t it?

What is the value of

new Onion( new Lamb( new Skewer()))

.onlyOnions()?

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?

57 `s.onlyOnions()`. 

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

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

Then what is `s.onlyOnions()`?

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

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

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

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

61 We determine which version of `onlyOnions` to use. 

And?

62 We use the one that belongs to Lamb. 

And now what is the answer?

63 false, because false follows the word `return` in the corresponding method definition in Lamb. 

Methods to Our Madness
Yes! The answer for
\texttt{new Onion(}
\texttt{new Lamb(}
\texttt{new Skewer())})
\texttt{.onlyOnions()}

is the same as the answer for
\texttt{new Lamb(}
\texttt{new Skewer())}
\texttt{.onlyOnions()},
which is false.

Describe the methods (i.e., the function) \texttt{onlyOnions} in your own words.

Here are our words:
"The methods determine for a Shish\(^{D}\) whether its contents are edible by an onion lover."

Describe how the methods (i.e., the function) \texttt{onlyOnions} accomplish this.

Here are our words again:
"For each layer of the Shish\(^{D}\), 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 \texttt{new Tomato(}
\texttt{new Skewer())}
a Shish\(^{D}\)?:

Yes.

Is \texttt{new Onion(}
\texttt{new Tomato(}
\texttt{new Skewer())})
a Shish\(^{D}\)?:

The object \texttt{new Tomato(}
\texttt{new Skewer())}
is an instance of Shish\(^{D}\), so we can also wrap an Onion around it.

And how about another Tomato?:
Sure.
Of course, there is no Lamb on it.

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

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

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

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

That’s no big deal now.

abstract boolean isVegetarian();

Shish

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?  

Five: one abstract, the others concrete.

Do abstract methods belong to the abstract class?  

Yes, they always do.

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

Yes, because Shish\textsuperscript{T} contains an abstract method called isVegetarian.

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.

The Second Bit of Advice

*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.*
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?

```java
abstract class Kebab<sup>D</sup> {
    Kebab
}

class Holder extends Kebab<sup>D</sup> {
    Object o;
    Holder(Object o) {
        o = o;
    }
    Holder
}

class Shallot extends Kebab<sup>D</sup> {
    Kebab<sup>D</sup> k;
    Shallot(Kebab<sup>D</sup> k) {
        k = k;
    }
    Shallot
}

class Shrimp extends Kebab<sup>D</sup> {
    Kebab<sup>D</sup> k;
    Shrimp(Kebab<sup>D</sup> k) {
        k = k;
    }
    Shrimp
}

class Radish extends Kebab<sup>D</sup> {
    Kebab<sup>D</sup> k;
    Radish(Kebab<sup>D</sup> k) {
        k = k;
    }
    Radish
}
```

They define a datatype and four variants that are similar in shape to Shish<sup>D</sup>.

Don’t forget the picture.
What is different about them? They are placed onto different Holders.

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

Here are some holders.

```java
abstract class RodD {}

class Dagger extends RodD {}

class Sabre extends RodD {}

class Sword extends RodD {}
```

Are they good ones?

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.

```java
abstract class PlateD {}

class Gold extends PlateD {}

class Silver extends PlateD {}

class Brass extends PlateD {}

class Copper extends PlateD {}

class Wood extends PlateD {}
```

What is

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

It's a Kebab$^D$. 

Methods to Our Madness
Sure it is. It only contains radishes and shallots.

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

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

Let’s define the methods (i.e., the function) which check whether a kebab contains only vegetarian foods, regardless of what Holder it is on.

Write the abstract method isVeggie.

That’s possible now.

abstract boolean isVeggie()

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

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

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

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

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

Except for the names of the methods and fields, the definitions are the same as they were for ShishD.

What is the value of

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

It is true, too.

What is

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

It is a KebabD, 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()?
```

And what is

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

Methods to Our Madness
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.
All the food is on an Integer.

The dagger.

The gold plate.

An Integer, whose underlying int is 52.

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

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

If we add this method to Kebab, then we must add a method definition to each of the four variants.

Methods to Our Madness
What is the value of
new Holder(
    new Integer(52))
.whatHolder()?

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

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

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

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()?

What is the value of
new Shallot(
    new Shrimp(
        new Holder(
            new Integer(52))))
.whatHolder()?
What is the value of
\[
\text{new Shrimp(}
    \text{new Holder(}
        \text{new Integer(52)}))
.\text{whatHolder}()?
\]

Does that mean that the value of
\[
\text{new Radish(}
    \text{new Shallot(}
        \text{new Shrimp(}
            \text{new Holder(}
                \text{new Integer(52)}))))
.\text{whatHolder}()
\]

is the same as
\[
\text{new Shallot(}
    \text{new Shrimp(}
        \text{new Holder(}
            \text{new Integer(52)})))
.\text{whatHolder}(),
\]

which is the same as
\[
\text{new Shrimp(}
    \text{new Holder(}
        \text{new Integer(52)}))
.\text{whatHolder}(),
\]

which is the same as
\[
\text{new Holder(}
    \text{new Integer(52))}
.\text{whatHolder}()?
\]

Yes, all four have the same answer:
\[
\text{new Integer(52)}.
\]

Here is the method for Shallot.

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

Write the methods of `whatHolder` for Shrimp and Radish.

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

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

They are all the same.
Here is the datatype and one of its variants.

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

Collect the remaining variants.

```
class Holder extends KebabD {
    Object o;
    Holder(Object o) {
      o = o;
    }
    boolean isVeggie() {
      return true;
    }
    Object whatHolder() {
      return o;
    }
}
```

```
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();
    }
}
```

There are only three left.

```
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();
    }
}
```

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?

Sure. We did something like that when we added Thyme and Sage to Seasoning.

Let's define another \( \text{Kebab}^D \).

```java
class Pepper extends \( \text{Kebab}^D \) {
    \( \text{Kebab}^D \) \( k \);
    Pepper(\( \text{Kebab}^D \) \_\( k \)) {
        \( k = .k; \)
    }
    boolean \text{isVeggie}(\) {
        return \( k \).\text{isVeggie}();
    }
    Object \text{whatHolder}(\) {
        return \( k \).\text{whatHolder}();
    }
}
```

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

Is it obvious how the new methods work?

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

And then there were six.

Yes, now \( \text{Kebab}^D \) has six variants.

Which of these points is closer to the origin:

- new \( \text{ManhattanPt}(3,4) \)
- new \( \text{ManhattanPt}(1,5) \)?

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:

- new \( \text{CartesianPt}(3,4) \)
- new \( \text{CartesianPt}(12,5) \)?

The first one, clearly. Its distance to the origin is 5, but the second distance is 13.
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 Math.sqrt(x * x + y * y);
    }

    boolean closerToO(CartesianPt 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.

```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 two characters `≤` and `≥` signify equality.

---

1. This is the two character symbol `≤`.

What is the value of `new ManhattanPt(3,4).closerToO(new ManhattanPt(1,5))?`?

false.

What is the value of `new ManhattanPt(1,5).closerToO(new ManhattanPt(3,4))?`?

true, obviously.

What is the value of `new CartesianPt(12,5).closerToO(new CartesianPt(3,4))?`?

false, and true is the value of `new CartesianPt(3,4).closerToO(new CartesianPt(12,5))`.  

So finally, what is the value of `new CartesianPt(3,4).closerToO(new ManhattanPt(1,5))?`?

That's nonsense.
Why?  

130 The method closerToO can only consume CartesianPts, not ManhattanPts.

How can we fix that?  

131 We could replace 

        (CartesianPt p) 

by 

        (Point^D p) 

in the definition of closerToO for CartesianPt.

If we do that, can we still determine the value of 

        p.distanceToO()?  

132 Yes, because the definition of Point^D obligates every variant to provide a method named distanceToO.

Why does it help to replace (CartesianPt p) by (Point^D p)?  

133 Every CartesianPt is a Point^D, and every ManhattanPt is a Point^D, too.

Here is the improved CartesianPt.  

134 Easy.

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

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

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

Improve the definition of ManhattanPt.

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

135 Yes, they are identical.
Correct, and therefore we can add a copy to the abstract class \texttt{Point}\textsuperscript{D} and delete the definitions from the variants.

```java
abstract class Point\textsuperscript{D} {
    boolean closerToO(Point\textsuperscript{D} p) {
        return distanceToO() \leq p.distanceToO();
    }
    abstract int distanceToO();
}
```

Looks correct.

What else do the two point variants have in common? Their fields. Shouldn’t we lift them, too?

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

```java
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();
    }
    abstract int distanceToO();
}
```

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}.

```java
class CartesianPt extends Point\textsuperscript{D} {
    CartesianPt(int x, int y) {
        super(x, y);
    }
    int distanceToO() {
        return \sqrt{x^2 + y^2};
    }
}
```

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

```java
class ManhattanPt extends Point\textsuperscript{D} {
    ManhattanPt(int x, int y) {
        super(x, y);
    }
    int distanceToO() {
        return x + y;
    }
}
```
The expressions super(x, y) in the constructors CartesianPt and ManhattanPt create a Point with the appropriate fields, and the respective constructor guarantees that the point becomes a CartesianPt or a ManhattanPt.

Do we now have everything that characterizes Point'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?
Looks like good toppings. Let’s add Sausage.

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

What’s New?
Here is our favorite pizza:

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

How about removing the anchovies?  

Let's remove them. What is the value of

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

It should be a cheese and olive pizza, like this:

```java
new Olive(
    new Cheese(
        new Crust()))).
```

What is the value of

```java
new Sausage(
    new Olive(
        new Anchovy(
            new Sausage(
                new Cheese(
                    new Crust()))))))
```

It should be a cheese, sausage, and olive pizza, like this:

```java
new Sausage(
    new Olive(
        new Sausage(
            new Cheese(
                new Crust())))).
```

Does removeA belong to the datatype `Pizza`?  

Yes, and it produces them, too.
Define the methods that belong to the five variants. Here is a start.

```java
abstract Pizza\[^D\] remA();

Pizza

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

Crust
```

We didn’t expect you to know this one.

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

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

Olive

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

Sausage
```

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

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

Cheese

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

Olive

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

Sausage
```

The cheese, the olives, and the sausages on the pizzas must be put back on top of the pizza that \( p . r e m A() \) produces.

Explain why we use \( n e w \) Cheese . . . , \( n e w \) Olive . . . , and \( n e w \) Sausage . . . when we define these methods.

```java
new Cheese . . . ,
new Olive . . . , and
new Sausage . . .
```

The methods \( r e m A \) must produce a Pizza\[^D\], so let’s use \( n e w \) Crust(), the simplest Pizza\[^D\], for the method in Anchovy.

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

Anchovy
```

Yes, and now the methods of \( r e m A \) produce pizzas without any anchovies on them.

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

```
new Anchovy(
    new Crust())
  .remA().
```

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()))
  .remA().
```

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()))))
  .remA().
```

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(\(p\).remA())
```

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()))).
  .remA()
```

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

   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())?

   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())?

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

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

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

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

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

So what is the final answer?

   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!

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()))
```

And the answer is

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

Does that mean the final answer is

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

It should be a double-cheese pizza.

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()));
```

Now the object is an anchovy.

Yes, it puts the cheese on whatever we get for

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

Yes, though it's not the answer we want.

Yes, but we need to add cheese on top.
What do we want? A double-cheese pizza like
\[
\text{new Cheese(}
\text{new Cheese(}
\text{new Crust())),}
\]
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.

\[
\text{Pizza}\textsuperscript{D} \text{ remA()} \{ 
\text{ return } p.\text{remA}(); \} 
\]

Does this remA still belong to Pizza\textsuperscript{D}? 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 new Olive(
\text{new Anchovy(}
new Cheese(
\text{new Anchovy(}
new Crust())))
).top\textsuperscript{C1}()? Easy, it adds cheese on top of each anchovy:
\[
\text{new Olive(}
\text{new Cheese(}
\text{new Anchovy(}
\text{new Cheese(}
\text{new Anchovy(}
\text{new Crust())))))).
\]

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()?

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.

```java
abstract Pizza<topAwC> topAwC();
```

```java
Pizza<topAwC> topAwC() {
    return new Crust();
}
```

```java
Pizza<topAwC> topAwC() {
    return new Sausage(p.topAwC());
}
```

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.

```java
Pizza<topAwC> topAwC() {
    return
        new Cheese(
            new Anchovy(p.topAwC()));
}
```
How many layers of cheese are in the result of

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

37 One, because \text{remA} removes all the anchovies, so that \text{topAwC} doesn’t add any cheese.

How many occurrences of cheese are in the result of

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

38 Three, because \text{topAwC} first adds cheese for each anchovy. Then \text{remA} removes all the anchovies:

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

Perhaps we should replace every anchovy with cheese.

39 We just did something like that.

Is it true that for each anchovy in \(x.x.topAwC().remA()\) adds some cheese?

40 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.

41 Aha!

What’s New?
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 subAbC. Here is the abstract method.

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

Here is a skeleton.

\[
\begin{align*}
\text{Pizza}^{\text{subAbC}}() \{ \\
\quad \text{return new Crust(); } \\
\}
\end{align*}
\]

\[
\begin{align*}
\text{Pizza}^{\text{subAbC}}() \{ \\
\quad \text{return new Cheese(p.subAbC()); } \\
\}
\end{align*}
\]

\[
\begin{align*}
\text{Pizza}^{\text{subAbC}}() \{ \\
\quad \text{return new Olive(p.subAbC()); } \\
\}
\end{align*}
\]

\[
\begin{align*}
\text{Pizza}^{\text{subAbC}}() \{ \\
\quad \text{return Sausage(p.subAbC()); } \\
\}
\end{align*}
\]

1 A better name for these methods would be substituteAnchovybyCheese.

Does this skeleton look familiar?

Yes, this skeleton looks just like those of topAwC and remA.

Define the method that belongs in Anchovy.

Here it is.

\[
\begin{align*}
\text{Pizza}^{\text{subAbC}}() \{ \\
\quad \text{return new Cheese(p.subAbC()); } \\
\}
\end{align*}
\]
abstract class Pizza\(^D\) {
  abstract Pizza\(^D\) remA();
  abstract Pizza\(^D\) topAwC();
  abstract Pizza\(^D\) subAbC();
}

class Crust extends Pizza\(^D\) {
  Pizza\(^D\) remA() {
    return new Crust();
  }
  Pizza\(^D\) topAwC() {
    return new Crust();
  }
  Pizza\(^D\) subAbC() {
    return new Crust();
  }
}

class Cheese extends Pizza\(^D\) {
  Pizza\(^D\) p;
  Cheese(Pizza\(^D\) _p) {
    p = _p;
  }
  Pizza\(^D\) remA() {
    return new Cheese(p.remA());
  }
  Pizza\(^D\) topAwC() {
    return new Cheese(p.topAwC());
  }
  Pizza\(^D\) subAbC() {
    return new Cheese(p.subAbC());
  }
}

class Olive extends Pizza\(^D\) {
  Pizza\(^D\) p;
  Olive(Pizza\(^D\) _p) {
    p = _p;
  }
  Pizza\(^D\) remA() {
    return new Olive(p.remA());
  }
  Pizza\(^D\) topAwC() {
    return new Olive(p.topAwC());
  }
  Pizza\(^D\) subAbC() {
    return new Olive(p.subAbC());
  }
}

class Anchovy extends Pizza\(^D\) {
  Pizza\(^D\) p;
  Anchovy(Pizza\(^D\) _p) {
    p = _p;
  }
  Pizza\(^D\) remA() {
    return p.remA();
  }
  Pizza\(^D\) topAwC() {
    return new Anchovy(p.topAwC());
  }
  Pizza\(^D\) subAbC() {
    return new Anchovy(p.subAbC());
  }
}

class Sausage extends Pizza\(^D\) {
  Pizza\(^D\) p;
  Sausage(Pizza\(^D\) _p) {
    p = _p;
  }
  Pizza\(^D\) remA() {
    return new Sausage(p.remA());
  }
  Pizza\(^D\) topAwC() {
    return new Sausage(p.topAwC());
  }
  Pizza\(^D\) subAbC() {
    return new Sausage(p.subAbC());
  }

---

1 This is similar to the interpreter and composite patterns [4].
Let's add more Pizza foods.

Here is one addition: Spinach.

```java
class Spinach extends Pizza {
    Pizza p;
    Spinach(Pizza p) {
        p = -p;
    }
    Pizza remA() {
        return new Spinach(p.remA());
    }
    Pizza topAwC() {
        return new Spinach(p.topAwC());
    }
    Pizza subAbC() {
        return new Spinach(p.subAbC());
    }
}
```

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

Isn't that neat?

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

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

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

Good idea.

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

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

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

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.

Great.

We will stick with anchovies.
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 Pizza\textsuperscript{P}, 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.

```java
class OnlyOnions\textsuperscript{V} {  
    boolean forSkewer() {  
        return true;  
    }
    boolean forOnion(Shish\textsuperscript{P} s) {  
        return s.onlyOnions();  
    }
    boolean forLamb(Shish\textsuperscript{P} s) {  
        return false;  
    }
    boolean forTomato(Shish\textsuperscript{P} s) {  
        return false;  
    }
}
```

\textsuperscript{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: OnlyOnions\textsuperscript{V}.

Almost. Each of them corresponds to an onlyOnions method in one of the variants of Shish\textsuperscript{P}.

That’s right. The major difference is that they are all in one class, a visitor, whose name is OnlyOnions\textsuperscript{V}.

Is onlyOnions different from OnlyOnions\textsuperscript{V}?  
6 The former is used to name methods, the latter names a class.

*Come to Our Carousel*
What point?

And that's the whole point.

We want all the methods in one class.

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

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

What is the difference between the method onlyOnions in the Onion variant and the method forOnion in the visitor OnlyOnions\(^V\)?

Right. What is the difference?

Yes, that is the difference. Are the other for methods in OnlyOnions\(^V\) related to their counterparts in the same way?

It is time to discuss the boring part.

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

Chapter 4

7 What point?

" What methods?"

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

10 It's about time.

11 Everything following return is the same.

12 The difference is that onlyOnions in Onion is followed by ( ) and that forOnion in OnlyOnions\(^V\) is followed by (Shish\(^P\) s).

13 Indeed, they are.

15 True, we still don't know how to make Shish\(^P\) and its variants work with this visitor class, which contains all the action.
Now take a look at this.

abstract class Shish\^D \{ 
\text{\texttt{OnlyOnions}}^v ooFn = new OnlyOnions\^v(); 
abstract boolean onlyOnions(); 
\}

class Skewer extends Shish\^D \{ 
boolean onlyOnions() {
    return ooFn\_for\_Skewer(); 
}
\}

class Onion extends Shish\^D \{ 
Shish\^D s;
Onion(Shish\^D\_s) {
    s = \_s;
}
boolean onlyOnions() {
    return ooFn\_for\_Onion(s); 
}
\}

class Lamb extends Shish\^D \{ 
Shish\^D s;
Lamb(Shish\^D\_s) {
    s = \_s;
}
boolean onlyOnions() {
    return ooFn\_for\_Lamb(s); 
}
\}

class Tomato extends Shish\^D \{ 
Shish\^D s;
Tomato(Shish\^D\_s) {
    s = \_s;
}
boolean onlyOnions() {
    return ooFn\_for\_Tomato(s); 
}
\}

This is a strange set of definitions. All the \textit{onlyOnions} methods in the variants look alike. Each of them uses an instance of \texttt{OnlyOnions}^v, which is created in the datatype, to invoke a \texttt{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.

Explain `s.onlyOnions()`.

Explain `ooFn.forOnion(s)`.

So what is the value of `new Onion( new Onion( new Skewer()) ) .onlyOnions() ?`

And how do we determine that value with these new definitions?

If “consume” refers to what follows the name between parentheses, the method consumes \(s\), which is the rest of the shish.

Nothing, because a skewer is the basis of a shish and therefore has no fields.

It is always the rest of the shish, below the top layer, which is an onion. In other words, it is everything but the onion.

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.

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
We must add two lines to each variant, and they are almost the same as those for `ooFn`.

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

```java
class Onion extends Shish {
    Shish s;
    Onion(Shish _s) {
        s = _s;
    }

    boolean onlyOnions() {
        return ooFn.forOnion(s);
    }
    boolean isVegetarian() {
        return ivFn.forOnion(s);
    }
}
```

```java
class Lamb extends Shish {
    Shish s;
    Lamb(Shish _s) {
        s = _s;
    }

    boolean onlyOnions() {
        return ooFn.forLamb(s);
    }
    boolean isVegetarian() {
        return ivFn.forLamb(s);
    }
}
```

```java
class Tomato extends Shish {
    Shish s;
    Tomato(Shish _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 `OnlyOnionsV` except for the method `forTomato`.

```java
class IsVegetarianV {
    boolean forSkewer() {
        return true;
    }
    boolean forOnion(ShishP s) {
        return s.isVegetarian();
    }
    boolean forLamb(ShishP s) {
        return false;
    }
    boolean forTomato(ShishP 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
Is
new Anchovy(
    new Olive(
        new Anchovy(
            new Cheese(
                new Crust()))))))
a shish kebab?

No, it’s a pizza.

abstract class Pizza\textsuperscript{D} \{ \}

class Crust extends Pizza\textsuperscript{D} \{ \}

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

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

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

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

So what do we do next? We can define the protocol for the methods that belong to Pizza\textsuperscript{D} and its extensions: \textit{remA, topA\&C, and subA\&C}. 
Great! Here is the abstract portion of the protocol.

```java
abstract class PizzaD {
    RemAv remFn = new RemAv();
    TopAwCv topFn = new TopAwCv();
    SubAbCv subFn = new SubAbCv();
    abstract PizzaD remA();
    abstract PizzaD topAwC();
    abstract PizzaD subAbC();
}
```

And here are some variants.

```java
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);
    }
}
```

```java
class Crust extends PizzaD {
    PizzaD p;
    Crust(PizzaD _p) {
        p = -p;
    }
    PizzaD remA() {
        return remFn.forCrust(p);
    }
    PizzaD topAwC() {
        return topFn.forCrust(p);
    }
    PizzaD subAbC() {
        return subFn.forCrust(p);
    }
}
```

```java
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.

41 How innovative! The variants are totally mindless, now.

```java
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);
    }
}
```

```java
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. Is it time to define the visitors that correspond to the methods remA. topAwC, and subAbC? 

Okay, here is RemAV.

class RemAV
{
  class TopAwC
  
  class RemA
  
  Define TopAwC.

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

And we thought we were working with a pizza pie.

By now, even this is routine.
5.

Objects Are People, Too
Have we seen this kind of definition before?  

```java
abstract class PieD {
    Pie
}

class Bot extends PieD {
    Bot
}

class Top extends PieD {
    Object t;
    PieD r;
    Top(Object t,PieD r) {
        t = t;
        r = r;
    }
    Top
}
```

1 Better names for these classes would be PizzaPieD, 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.

```java
abstract class FishD {
}

class Anchovy extends FishD {
}

class Salmon extends FishD {
}

class Tuna extends FishD {
}
```
Nice datatype. Is

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

a pizza pie?

It is a pizza pie, and so is

\[
\text{new Top(new Tuna(),}
\ \text{new Top(new Integer(42),}
\ \text{new Top(new Anchovy(),}
\ \text{new Top(new Integer(5),}
\ \text{new Bot()))).}
\]

What is the value of

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

\text{.remA}?\]

It is this fishy pizza pie:

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

Is it true that the value of

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

is

\[
\text{new Top(new Salmon(),}
\ \text{new Tuna(),}
\ \text{new Bot())?}
\]

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

Does \text{remA} belong to \text{Pie_}\text{D}?

Yes, and it produces pizza pies.

Define the protocol for \text{RemA}_\text{V}. We provide the \text{abstract} part.

\[
\text{RemA}_\text{V} \text{.rF}n = \text{new RemA}_\text{V}();
\]

\text{abstract Pie}_\text{D} \text{.remA}();

This is easy by now.

\text{Pie}_\text{D} \text{.remA}() \{
\text{return rF}n.\text{forBot}();
\}

\text{Pie}_\text{D} \text{.remA}() \{
\text{return rF}n.\text{forTop}(1,x);\}

\text{Bot}
Great. Isn’t that easy?

What part of this exercise differs from datatype to datatype?

Anything else?

Why (t,r)?

Let’s define the visitor RemA\textsuperscript{V}.

class RemA\textsuperscript{V} {
  Pie\textsuperscript{D} forBot() {
    return _____ ; }
  Pie\textsuperscript{D} forTop(Object t,Pie\textsuperscript{D} r) {
    if (new Anchovy().equals(t))
      return _____ ;
    else
      return _____ ;
  }
}

Great guesses! What does

if (expr\textsubscript{1})
  return expr\textsubscript{2};
else
  return expr\textsubscript{3};

mean?

And what does

new Anchovy().equals(t)

mean?

Not yet. It depends on what equals means.

Easy and boring.

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

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

Because these are the fields of Top.

Here are some guesses.

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

We guess:

“This produces the value of either expr\textsubscript{2} or expr\textsubscript{3}, depending on whether or not expr\textsubscript{1} is determined to be true or false, respectively.”

We could guess:

“This expression determines whether \texttt{t} is equal to new Anchovy().”

What?
What is the value of
new Anchovy().equals(new Anchovy())?

Yes! And what is the value of
new Anchovy().equals(new Tuna())?

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

If we know that equals's answer is always false, why bother to use it?

We must define it anew¹ for all classes whose instances we wish to compare.

Okay. How?

For Fish¹ and its variants it works like this.

```
abstract class Fish¹ {
}

class Anchovy extends Fish¹ {
    public boolean equals(Object o) {
        return (o instanceof Anchovy);
    }
}

class Salmon extends Fish¹ {
    public boolean equals(Object o) {
        return (o instanceof Salmon);
    }
}

class Tuna extends Fish¹ {
    public boolean equals(Object o) {
        return (o instanceof Tuna);
    }
}
```

¹ Not always. We explain the correct answer in chapter 10.

Assuming that
(o instanceof Tuna)
is true when o is an instance of Tuna, these method definitions are obvious.
22. Aren’t they? Is every value constructed with \texttt{new} an instance of \texttt{Object}? Yes. Every such value is an \texttt{Object}, because every class \texttt{extends} \texttt{Object} directly or indirectly.

23. If \texttt{class A extends B}, is every value created by \texttt{new A(...) an instance of class B}? Yes, and of the class that \texttt{B} extends and so on.

24. Now, what is the value of \texttt{new Anchovy().equals(new Anchovy())}? \texttt{true}, because \texttt{new Anchovy()} is an instance of \texttt{Anchovy}.

25. Yet the value of \texttt{new Anchovy().equals(new Tuna())} is still \texttt{false}. Of course, because an anchovy is never a tuna.

26. Could we have written \texttt{RemA} without using \texttt{equals}?

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

Why haven’t we defined it this way?

27. Easy, because we want to generalize \texttt{RemA} so that it works for any kind of fish topping. 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.

28. What are good names for the more general methods and visitor? How about \texttt{remFish} and \texttt{RemFish}?
How do we use `remFish`?

Add the protocol for `RemFish`. We designed the abstract portion.

```java
RemFish remFish = new RemFish();
abstract Pie remFish(Fish f);
```

The rest is routine.

```java
Pie remFish(Fish f) {
    return rfFn.forBot(f);
}
```

Where do `(f)` and `(t,r,f)` come from?

The `f` stands for the `Fish` we want to remove in both cases. The `t` and the `r` are the fields of `Top`; `Bot` doesn't have any.

Let's define `RemFish` and its two methods.

```java
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));
    }
}
```

If we add another kind of fish to our datatype, what would happen to the definition of `remFish`?

Nothing, we just have to remember to add `equals` to the new variant.
Let's try it out with a short example:

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

The object is a topping, so we use `forTop` from `RemFish`.

Yes. What values does `forTop` consume?

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?

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?

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

```java
new Bot()
    .remFish(new Anchovy())?
```

It is the same as

```java
forBot(f);
```

where `f` is `new Anchovy()`.

What does `forBot` in `RemFish` produce?

It produces `new Bot()`, no matter what `f` is.

All clear?

Ready to move on, after snack time.
Yes, it looks like what we just evaluated.

What does \textit{remInt} do? It removes integers from pizza pies just as \textit{remFish} removes fish from pizza pies.

Who defined \textit{equals} for \textit{Integer}? The Machine decided
\begin{verbatim}
new Integer(0).equals(new Integer(0))
\end{verbatim}
to be true, and the rest was obvious.

Define the visitor \textit{RemInt}.

\begin{verbatim}
class RemInt {
    Pie\textsuperscript{D} forBot(Integer i) {
        return new Bot();
    }
    Pie\textsuperscript{D} forTop(Object t,Pie\textsuperscript{D} r,Integer i) {
        if (i.equals(t))
            return r.remInt(i);
        else
            return new Top(t,r.remInt(i));
    }
}
\end{verbatim}

Does it matter that this definition uses \textit{i} and not \textit{f}? No, \textit{i} is just a better name than \textit{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.
Can we remove Integers from Pie^D\text{'}s?  
Yes.

Can we remove Fish^D from Pie^D\text{'}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 \texttt{new} is an Object.

Just do it!  
It’s done.

```java
class Rem^V \{  
    Pie^D \text{ forBot}(Object o) \{  
        return new Bot();  
    }  
    Pie^D \text{ forTop}(Object t,Pie^D 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.

Why not RemFish^V and RemA^V and RemInt^V?  
They are unnecessary once we have Rem^V.

\textit{Objects Are People, Too}
And here are the pieces for Bot and Top.

```java
abstract class Pie {
    Rem remFn = new Rem();
    abstract Pie rem(Object o);
}
```

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

```java
class Top extends Pie {
    Object t;
    Pie r;
    Top(Object t, Pie r) {
        t = t;
        r = r;
    }
    Pie rem(Object o) {
        return remFn.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())))
```  

And here are the pieces for Bot and Top.

Works like a charm with the same result as before.

```java
class Top extends Pie {
    Object t:
    Pie r;
    Top(Object , Pie r) {
        t = t;
        r = r;
    }
    Pie rem(Object o) {
        return remFn.forTop(t, r, o);
    }
}
```

And how about

```java
new Top(new Anchovy(),
    new Bot())
```  

Ditto.

Next:

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

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()))).
\texttt{.rem(new Zero())}?

What's wrong with that?

We expected it to remove \texttt{new Zero()} from the pizza.

And why didn't it?

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

Always?

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

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

\begin{verbatim}
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;
        }
    }
}
\end{verbatim}

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

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

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

\footnote{In Java, this is called (downward) casting, because \texttt{OneMoreThan extends NumD}}
No. What is the type of o?

So what is o.predecessor?

Correct. What do we know after if has determined that

(o instanceof OneMoreThan)

is true?

Precisely. So what does ((OneMoreThan)o).predecessor do?

What is ((OneMoreThan) o)'s type?

Are o and ((OneMoreThan)o) interchangeable?

Is this complicated?

Did you also notice the

predecessor

.equals((OneMoreThan)o).predecessor

in equals for OneMoreThan?

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

So the second one, ((OneMoreThan) o).predecessor, refers to the predecessor field of the instance of OneMoreThan consumed by equals.
Yes. Are these two objects equal?  

If they are similar\(^1\) to the same int, they are equal. But most of the time, they are not.

\(^1\) Check chapter 1 for “similar.”

Time for lunch?  

That’s just in time.

Did you have a good lunch break?  

Yes, thank you.

Now what is the value of  

\[
\text{new Top(new Anchovy(),}
\text{new Top(new Integer(3),}
\text{new Top(new Zero(),}
\text{new Bot()))}
\text{.rem(new Zero())?}
\]

Now we get  

\[
\text{new Top(new Anchovy(),}
\text{new Top(new Integer(3),}
\text{new Bot())},
\]

which is precisely what we want.

And why?  

Because \textit{equals} now knows how to compare \textit{Num}^D\text{s}.

Do we always add \textit{equals} to a class?  

No, only if we need it.

Do we need \textit{equals} when we want to substitute one item for another on a pizza pie?  

Yes, we do.

What is the value of  

\[
\text{new Top(new Anchovy(),}
\text{new Top(new Tuna(),}
\text{new Top(new Anchovy(),}
\text{new Bot()))}
\text{.substFish(new Salmon(),}
\text{new Anchovy())?}
\]

It is the same pizza pie with all the anchovies replaced by salmon:  

\[
\text{new Top(new Salmon(),}
\text{new Top(new Tuna(),}
\text{new Top(new Salmon(),}
\text{new Bot()))}.
\]

What kind of values does \textit{substFish} consume?  

It consumes two \textit{fish} and works on \textit{Pie}^D\text{s}.

\textit{Objects Are People, Too}
And what does it produce?  

It always produces a Pie\(^2\).

What is the value of 
\[
\text{new Top(new Integer(3),}
\text{new Top(new Integer(2),}
\text{new Top(new Integer(3),}
\text{new Bot())))}
\]

It is the same pizza pie with all 3s replaced by 5s: 
\[
\text{new Top(new Integer(5),}
\text{new Top(new Integer(2),}
\text{new Top(new Integer(5),}
\text{new Bot())))}
\]

What kind of values does subst\(\text{Int}\) consume?  

It consumes two Integers and works on Pie\(^2\)s.

And what does it produce?  

It always produces a Pie\(^2\).

We can define Subst\(\text{Fish}^V\).

```java
class Subst\(\text{Fish}^V\) {
    Pie\(^2\) forBot(Fish\(^2\) n,Fish\(^2\) o) {
        return new Bot(); }
    Pie\(^2\) forTop(Object t,
        Pie\(^2\) r, 
        Fish\(^2\) n, 
        Fish\(^2\) o) {
            if (o.equals(t))
                return new Top(n,r.subst\(\text{Fish}^V(n,o)));
            else
                return new Top(t,r.subst\(\text{Fish}^V(n,o)));
        }
}
```

Define Subst\(\text{Int}^V\).

```java
class Subst\(\text{Int}^V\) {
    Pie\(^2\) forBot(Integer n,Integer o) {
        return new Bot(); }
    Pie\(^2\) forTop(Object t,
        Pie\(^2\) r, 
        Integer n, 
        Integer o) {
            if (o.equals(t))
                return new Top(n,r.subst\(\text{Int}^V(n,o)));
            else
                return new Top(t,r.subst\(\text{Int}^V(n,o)));
        }
}
```

Did we forget the boring parts?  

Yes, because there is obviously a more general version like Rem\(^V\).
We substitute \( \text{Object} \) for \( \text{Fish} \) and \( \text{Integer} \). Define it.

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

```
abstract class \( \text{Pie}^\mathcal{P} \) {
    \text{Pie}^\mathcal{P} \text{ rem}(\text{Object } o) {
        \text{return } \text{remFn}.\text{forBot}(o); }
    \text{Pie}^\mathcal{P} \text{ subst}(\text{Object } n,\text{Object } o) {
        \text{return } \text{substFn}.\text{forBot}(n,o); }
    }
}
```

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

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

So?  

The abstract part is obvious.

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

*That was some heavy lifting.*
6. Boring Protocols
Are protocols truly boring? We acted as if they were.

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.

```java
abstract class PieD {
    RemV remFn = new RemV();
    SubstV substFn = new SubstV();
    abstract PieD rem(Object o);
    abstract PieD subst(Object n, Object 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.

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

```java
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);
    }
}
```
And that is all there is to the methods in the variants of a protocol.

Let’s not create `remFn` and `substFn` in the datatype:

```java
abstract class Pie
{
    abstract Pie rem(Rem v, Object o);
    abstract Pie subst(Subst v, Object n, Object o);
}
```

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 `Rem v` and a `Subst v`, respectively. And every `Rem v` defines `forBot` and `forTop`, and every `Subst v` does.

Here is how it changes `Top`.

```java
class Top extends Pie
{
    Object t;
    Pie r;
    Top(Object t, Pie r) {
        t = t;
        r = r;
    }
    Pie rem(Rem v, Object o) {
        return remFn.forTop(t, r, o);
    }
    Pie subst(Subst v, Object n, Object o) {
        return substFn.forTop(t, n, o);
    }
}
```

How does it affect `Bot`?

```java
class Bot extends Pie
{
    Pie rem(Rem v, Object o) {
        return remFn.forBot(o);
    }
    Pie subst(Subst v, Object n, Object o) {
        return substFn.forBot(n, 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?

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

Chapter 6
We still have some work to do. We still have some work to do. Consuming an extra value here also affects how the methods rem and subst are used. In Rem$^v$ and Subst$^v$, the interesting parts, for example.

Like what?

Where are they used?

Yes. Here is Rem$^v$.

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

That takes all the fun out of it.

```
class Subst$^v$ {
    Pie$^D$ forBot(Object n,
        Object o) {
        return new Bot();
    }
    Pie$^D$ forTop(Object t,
        Pie$^D$ 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));
    }
}
```

Modify Subst$^v$ accordingly.

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 rem in Bot and Top asks for the forBot and forTop methods of remFn, respectively.

Boring Protocols
How does that happen?  

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}.  

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

That’s it. Tricky?  

Not really, just self-referential.

Why?  

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

What is the value of  
\[
\text{new Top(new Anchovy(),}
\text{new Top(new Integer(3),}
\text{new Top(new Zero(),}
\text{new Bot())))}
\]  
\[ .rem(new Rem\textsuperscript{V}(),
\text{new Zero())?} \]

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?  

It creates a \textit{Rem} \textsuperscript{V} object, which corresponds to the \textit{remFn} in the old \textit{Pie} \textsuperscript{P}.

What is the value of  
\[
\text{new Top(new Integer(3),}
\text{new Top(new Integer(2),}
\text{new Top(new Integer(3),}
\text{new Bot())))}
\]  
\[ .subst(new Subst\textsuperscript{V}(),
\text{new Integer(5),}
\text{new Integer(3))?} \]

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?  

It creates a Subst\textsuperscript{V} object, which corresponds to the \textit{remFn} in the old \textit{Pie}\textsuperscript{D}.

So what is the underlined part about?  

We changed the methods in \textit{Pie}\textsuperscript{D}, which means that we must also change how it is used.

Ready for the next protocol?  

Let’s grab a quick snack.

How about some ice cream?  

Cappuccino crunch sounds great. The more coffee, the better.

Take a look at \textit{subst} in \textit{Top} and at \textit{forTop} in Subst\textsuperscript{V}. What happens to the values that they consume?  

Nothing really. They get handed back and forth, though \textit{forTop} compares \textit{o} to \textit{t}.

Is the handing back and forth necessary?  

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));
      }
   }
```

Wow. This visitor has two fields.\textsuperscript{1}

\textsuperscript{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 \textit{subst}.
We use
\[
\text{new Subst}^V (\text{new Integer(5)},
\text{new Integer(3))}.
\]

It creates a Subst\textsuperscript{V} whose methods know how
to substitute \text{new Integer(5)} for all
occurrences of \text{new Integer(3)} in Pie\textsuperscript{7}.

The values have now become fields of the
Subst\textsuperscript{V} object to which the methods belong.
They no longer need to be consumed.

We write
\[
\text{new Top(new Integer(3)},
\text{new Top(new Integer(3)}),
\text{new Top(new Integer(3))},
\text{new Bot(())})
\]
\text{.subst(new Subst}^V (\text{new Integer(5)},
\text{new Integer(3)}).

We write
\[
\text{new Top(new Integer(3)},
\text{new Top(new Integer(2)},
\text{new Top(new Integer(3)},
\text{new Bot(())})
\]
\text{.subst(new Subst}^V (\text{new Integer(7)},
\text{new Integer(2)}).

Of course, because the methods subst in the
Bot and Top variants consume only one value
now.
In the Top variant, we still need to hand over both \( t \) and \( r \).

```java
abstract class Pie\(D\) {
    abstract Pie\(D\) rem(Rem\(V\) remFn);
    abstract Pie\(D\) subst(Subst\(V\) substFn);
}

class Top extends Pie\(D\) {
    Object \( t \);
    Pie\(D\) \( r \);
    Top(Object .\( t \),Pie\(D\) .\( r \)) {
        \( t = .\( t \); \)
        \( r = .\( r \); \)
    }
}
```

Is there anything else missing?

We haven’t defined Rem\(V\) for this new protocol. But it is simple and hardly worth our attention.

What is the difference between \( \text{rem} \) and \( \text{subst} \) in Bot?

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

What is the difference between \( \text{rem} \) and \( \text{subst} \) 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 \( \text{rem} \) is defined as it is or as

```java
Pie\(D\) rem(Rem\(V\) substFn) {
    return substFn.forTop(\( t \),r); }
```

True, because \( \text{substFn} \) is just a name for a value we don’t know yet. But how can we make the types the same?

Both Rem\(V\) and Subst\(V\) 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
Great job, except that we will use interface for specifying visitors like these.

Here it is.

```java
abstract class PieVisitor<T> {
    abstract Pie<T> forBot();
    abstract Pie<T> forTop(Object t, Pie<T> r);
}
```

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

```java
interface PieVisitor<T> {
    Pie<T> forBot();
    Pie<T> forTop(Object t, Pie<T> r);
}
```

This superscript is a reminder that the name refers to an interface. Lower superscripts when you enter this kind of definition in a file: PieVisitor1.

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

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

```java
Pie<T> rem(PieVisitor<T> pvFn) {
    return pvFn.forTop(t, r);
}
```

and

```java
Pie<T> subst(PieVisitor<T> 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 pvFn? Easy: ask, because we ask for services.

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

```java
abstract class Pie {
    abstract Pie accept(PieVisitor ask);
}
```

class Bot extends Pie {
    Pie accept(PieVisitor ask) {
        return ask.forBot();
    }
}

class Top extends Pie {
    Object t;
    Pie r;
    Top(Object t,Pie r) {
        t = t;
        r = r;
    }
    Pie accept(PieVisitor ask) {
        return ask.forTop(t,r);
    }
}
```

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.
Now define the new Subst$^V$.

Here it is.

```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));
    }
}
```

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

Here is our picture.

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

The Subst$^V$ visitor implements PieVisitor$^T$. It doesn’t extend it. Arrows mean “extends,” lines mean “implements.”

And the dashed line?

It tells us the name of the method that connects the datatype to the visitors.
What is the value of
new Top(new Anchovy(),
new Top(new Tuna(),
new Top(new Anchovy(),
new Top(new Tuna(),
new Top(new Anchovy(),
new Bot())())))
   .accept(new LtdSubstV(2,
                        new Salmon(),
                        new Anchovy()))?

Explain what LtdSubstV produces.\(^1\)

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

Good. Define LtdSubstV.

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

```java
class LtdSubstV implements PieVisitor<
    int c;
    Object n;
    Object o;
    LtdSubstV(int _c, Object _n, Object _o) {
        c = _c;
        n = _n;
        o = _o;
    }

    public PieD forBot() {
        return new Bot();
    }
    public PieD forTop(Object t, PieD 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 Tuna(),
        new Top(new Anchovy(),
          new Bot())))))

  .accept(new LtdSubst\^v(2,
    new Salmon(),
    new Anchovy()))?

How come?

Why doesn't \( c \) ever change?

Can we fix this?

If \( c \) stands for the current count, how do we create a \( \text{LtdSubst}^v \) that shows that we have just substituted one fish by another.

Ooops, there are too few anchovies on this pizza pie:

new Top(new Salmon(),
  new Top(new Tuna(),
    new Top(new Salmon(),
      new Top(new Tuna(),
        new Top(new Salmon(),
          new Bot()))))).

Because \( c \), the counting field, never changes.

Because this, the \( \text{LtdSubst}^v \) that performs the substitutions, never changes.

We can't change this, but we can replace this with a new \( \text{LtdSubst}^v \) that reflects the change.

Simple, we use

\( \text{new LtdSubst}^v(c - 1, n, o) \)

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 LtdSubst$^v$.

```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$^P$ forBot() {
        return new Bot();
    }
    public Pie$^P$ forTop(Object t, Pie$^P$ 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));
    }
}
```

Voilà.

How does this differ from new LtdSubst$^v$(c - 1, n, o)?

They are two different LtdSubst$^v$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.
7. Oh My!
Is
  new Flat(new Apple(),
  new Flat(new Peach(),
  new Bud()))
a flat Tree^D?

  Yes.   *

Is
  new Flat(new Pear(),
  new Bud())
a flat Tree^D?

  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())))?

  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()))).

  No, it isn't flat either.  

Is it flat?

Is the difference between flat trees and split trees obvious now?

  Unless there is anything else to Tree^D, it's totally clear.  

Good. Then let's move on.

  Okay, let's.

Oh My!
Here are some fruits.

```java
abstract class Fruit
{
    public boolean equals(Object o)
    {
        return (o instanceof Peach);
    }
}

class Peach extends Fruit
{
    public boolean equals(Object o)
    {
        return (o instanceof Peach);
    }
}

class Apple extends Fruit
{
    public boolean equals(Object o)
    {
        return (o instanceof Apple);
    }
}

class Pear extends Fruit
{
    public boolean equals(Object o)
    {
        return (o instanceof Pear);
    }
}

class Lemon extends Fruit
{
    public boolean equals(Object o)
    {
        return (o instanceof Lemon);
    }
}

class Fig extends Fruit
{
    public boolean equals(Object o)
    {
        return (o instanceof Fig);
    }
}
```

It does not differ too much from what we have seen before.

```java
abstract class Tree
{
    class Bud extends Tree
    {
    }

class Flat extends Tree
{
    Fruit f;
    Tree t;
    Flat(Fruit .f,Tree .t) {
        f = .f;
        t = .t;
    }
}

class Split extends Tree
{
    Tree l;
    Tree r;
    Split(Tree .l,Tree .r) {
        l = .l;
        r = .r;
    }
}
```

Let's say all Tree's are either flat, split, or bud. Formulate a rigorous description for Tree's.

Did you notice that we have redefined the method equals in the variants of Fruit?

That probably means that we will need to compare fruits and other things.

Do Tree's variants contain equals?

No, which means we won't compare them, but we could.
The name of the new datatype occurs twice in its Split variant.

Let's add a visitor interface whose methods produce booleans.

```java
interface bTreeVisitor<T> {
    boolean forBud();
    boolean forFlat(Fruit<T> f, Tree<T> t);
    boolean forSplit(Tree<T> l, Tree<T> r);
}
```

That just means extending what we have with one method each.

```java
class Bud extends Tree<T> {
    boolean accept(bTreeVisitor<T> ask) {
        return ask.forBud();
    }
}
```

```java
class Flat extends Tree<T> {
    Fruit<T> f;
    Tree<T> t;
    Flat(Fruit<T> f, Tree<T> t) {
        f = f;
        t = t;
    }

    boolean accept(bTreeVisitor<T> ask) {
        return ask.forFlat(f,t);
    }
}
```

```java
class Split extends Tree<T> {
    Tree<T> l;
    Tree<T> r;
    Split(Tree<T> l, Tree<T> r) {
        l = l;
        r = r;
    }

    boolean accept(bTreeVisitor<T> ask) {
        return ask.forSplit(l,r);
    }
}
```

But isn't 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!
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>How many methods does the definition of blsFlat\textsuperscript{v} contain, assuming it implements bTreeVisitor\textsuperscript{2}?</td>
<td>Three, because it works with Tree\textsuperscript{2}’s, and the datatype definition for Tree\textsuperscript{2}’s has three variants.</td>
</tr>
<tr>
<td>What type of values do the methods of blsFlat\textsuperscript{v} produce?</td>
<td>booleans.</td>
</tr>
<tr>
<td>What visitor does blsFlat\textsuperscript{v} remind us of?</td>
<td>OnlyOnions\textsuperscript{v}.</td>
</tr>
</tbody>
</table>

Here is a skeleton for blsFlat\textsuperscript{v}.

```java
class blsFlat\textsuperscript{v} implements bTreeVisitor\textsuperscript{2} {
    public boolean forBud() {
        return true;
    }
    public boolean forFlat(Fruit\textsuperscript{2} f, Tree\textsuperscript{2} t) {
        return true;
    }
    public boolean forSplit(Tree\textsuperscript{2} l, Tree\textsuperscript{2} r) {
        return false;
    }
}
```

Fill in the blanks.

Define the blsSplit\textsuperscript{v} visitor, whose methods check whether a Tree\textsuperscript{2} is constructed with Split and Bud only.

```java
class blsSplit\textsuperscript{v} implements bTreeVisitor\textsuperscript{2} {
    public boolean forBud() {
        return true;
    }
    public boolean forFlat(Fruit\textsuperscript{2} f, Tree\textsuperscript{2} t) {
        return false;
    }
    public boolean forSplit(Tree\textsuperscript{2} l, Tree\textsuperscript{2} r) {
        return false;
    }
}
```

Here is the easy part.

```java
class blsSplit\textsuperscript{v} implements bTreeVisitor\textsuperscript{2} {
    public boolean forBud() {
        return true;
    }
    public boolean forFlat(Fruit\textsuperscript{2} f, Tree\textsuperscript{2} t) {
        return false;
    }
    public boolean forSplit(Tree\textsuperscript{2} l, Tree\textsuperscript{2} r) {
        return false;
    }
}
```
What is difficult about the last line?

Isn't that easy?

And then?

If \( l.\text{accept}(\text{this}) \) is true, do we need to know whether \( r.\text{accept}(\text{this}) \) is true?

If \( \neg l.\text{accept}(\text{this}) \) is false, do we need to know whether \( r.\text{accept}(\text{this}) \) is true?

Finish the definition of \( \text{blsSplit}^{\vee} \) using

```java
class \text{blsSplit}^{\vee} \text{ implements } \text{bTreeVisitor}^{7} \{
    \text{public boolean } \text{forBud}() \{
        \text{return true;} \}
    \text{public boolean } \text{forFlat}(\text{Fruit}^{D} f, \text{Tree}^{D} t) \{
        \text{return false;}
    \text{public boolean } \text{forSplit}(\text{Tree}^{D} l, \text{Tree}^{D} r) \{
        \text{if}(L.\text{accept}(\text{this}))
            \text{return } r.\text{accept}(\text{this});
        \text{else}
            \text{return false;}
    \}
\}
```

1 We could have written the if...as \( \text{return } l.\text{accept}(\text{this}) \land \neg r.\text{accept}(\text{this}) \).

Oh My!
Give an example of a $\text{Tree}^D$ for which the methods of $\text{blsSplit}^D$ respond with true.

There is a trivial one:

```java
new Bud();
```

How about one with five uses of $\text{Split}$?

Here is one:

```java
new Split(
    new Split(
        new Bud(),
        new Split(
            new Bud(),
            new Bud())),
    new Split(
        new Bud(),
        new Split(
            new Bud(),
            new Bud()))).
```

Does this $\text{Tree}^D$ have any fruit?

No.

Define the $\text{bHasFruit}^D$ visitor.

Here it is.

```java
class bHasFruit^D implements bTreeVisitor^D {
    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);
        }
    }
}
```

---

1 We could have written the if … as

```java
return l.accept(this) || r.accept(this);
```
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())))?

29
3.

What is the height of
new Split(
    new Bud(),
    new Flat(new Lemon(),
        new Bud()))?

28
2.

What is the height of
new Flat(new Lemon(),
    new Bud())?

30
1.

What is the height of
new Bud()?

31
0.

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?
What is the value of
\[
\text{new Split(}
\text{new Split(}
\text{new Bud(),}
\text{new Bud()),}
\text{new Flat(new Fig(),}
\text{new Flat(new Lemon(),}
\text{new Flat(new Apple(),}
\text{new Bud()))))}
\].accept(new iHeightv())?

Why is the height 4?  
Because the value of
\[
\text{new Split(}
\text{new Bud(),}
\text{new Bud())}
\].accept(new iHeightv())
is 1; the value of
\[
\text{new Flat(new Fig(),}
\text{new Flat(new Lemon(),}
\text{new Flat(new Apple(),}
\text{new Bud()))})
\].accept(new iHeightv())
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 TreeD was constructed with Split and those two TreeD’s.

\( \square \) picks the larger of two numbers, \( x \) and \( y \).  
Oh, that’s nice. What kind of methods does iHeightv define?

\(^1\) When you enter this in a file, use
\[\text{Math.max(x,y)}.\]
Math is a class that contains \text{max} as a (static) method.

iHeightv’s methods measure the heights of the TreeD’s to which they correspond.  
Now that’s a problem.
We defined only interfaces that produce booleans in this chapter.

So what?

The methods of iHeight produce ints, which are not booleans.

Okay, so let's define a visitor interface that produces ints.

It's almost the same as bTreeVisitor.

```java
interface iTreeVisitor2 {  
    int forBud();  
    int forFlat(Fruit f, Tree t);  
    int forSplit(Tree l, Tree r);  
}
```

Yes, and once we have that we can add another accept method to Tree.

Does that mean we can have two methods with the same name in one class?

In Java, defining multiple methods with the same name and different input types is called "overloading."

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 is indeed different from iTreeVisitor, so we can have two versions of accept in Tree.

Add the new accept methods to Tree's variants. Start with the easy one.

It is easy.

```java
class Bud extends Tree {  
    boolean accept(bTreeVisitor ask) {  
        return ask.forBud();  
    }  
    int accept(iTreeVisitor ask) {  
        return ask.forBud();  
    }  
}
```
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);
    }
}
```

Here is iHeightv.

```java
class iHeightv implements iTreeVisitor {
    public int forBud() {
        return 0;
    }
    public int forFlat(FruitD f, TreeD t) {
        return t.accept(this) + 1;
    }
    public int forSplit(TreeD l, TreeD r) {
        return (l.accept(this) + r.accept(this)) + 1;
    }
}
```

Complete these methods.

What is the value of

new Split(
    new Bud(),
    new Bud())
    .accept(new iHeightv())?

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.
If the visitor subst substitutes apples for figs, here is what we get:

```
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()))))
```

It's like substFish and substInt from the end of chapter 5, but we can't do it just yet.

Correct. Define the subst\textsuperscript{v} visitor.

What's the problem?

Its methods produce Tree\textsuperscript{s}, neither ints nor boolean\textsuperscript{s}, which means that we need to add yet another interface.

```
interface tTreeVisitor\textsuperscript{v} {
  Tree\textsuperscript{D} forBud();
  Tree\textsuperscript{D} forFlat(Fruit\textsuperscript{D} f, Tree\textsuperscript{D} t);
  Tree\textsuperscript{D} forSplit(Tree\textsuperscript{D} l, Tree\textsuperscript{D} r);
}
```

Good job. How about the datatype Tree\textsuperscript{D}.

Easy. Here is the abstract one.

```
abstract class Tree\textsuperscript{D} {
  abstract boolean accept(bTreeVisitor\textsuperscript{v} ask);
  abstract int accept(iTreeVisitor\textsuperscript{v} ask);
  abstract Tree\textsuperscript{D} accept(tTreeVisitor\textsuperscript{v} ask);
}
```

Oh My!
Define the variants of $\text{Tree}^D$. No problem.

```java
class Bud extends TreeD {
    boolean accept(bTreeVisitor^ ask) {
        return ask.forBud();
    }
    int accept(iTreeVisitor^ ask) {
        return ask.forBud();
    }
    TreeD accept(tTreeVisitor^ ask) {
        return ask.forBud();
    }
}

class Flat extends TreeD {
    Fruit f;
    TreeD t;
    Flat(Fruit f, TreeD t) {
        f = f;
        t = t;
    }
    boolean accept(bTreeVisitor^ ask) {
        return ask.forFlat(f, t);
    }
    int accept(iTreeVisitor^ ask) {
        return ask.forFlat(f, t);
    }
    TreeD accept(tTreeVisitor^ ask) {
        return ask.forFlat(f, t);
    }
}

class Split extends TreeD {
    TreeD l;
    TreeD r;
    Split(TreeD l, TreeD r) {
        l = l;
        r = r;
    }
    boolean accept(bTreeVisitor^ ask) {
        return ask.forSplit(l, r);
    }
    int accept(iTreeVisitor^ ask) {
        return ask.forSplit(l, r);
    }
    TreeD accept(tTreeVisitor^ ask) {
        return ask.forSplit(l, r);
    }
}
```
Then define $t_{\text{Subst}}^v$.

That's easy, too. It has two fields, one for the new Fruit$^D$ and one for the old one, and the rest is straightforward.

```java
class $t_{\text{Subst}}^v$ implements $t_{\text{TreeVisitor}}^D$ {
    Fruit$^D$ n;
    Fruit$^D$ o;
    $t_{\text{Subst}}^v$(Fruit$^D$ n,Fruit$^D$ o) {
        n = .n;
        o = .o;
    }
    public Tree$^D$ forBud() {
        return new Bud();
    }
    public Tree$^D$ forFlat(Fruit$^D$ f,Tree$^D$ t) {
        if (o.equals(f))
            return new Flat(n,t.accept(this));
        else
            return new Flat(f,t.accept(this));
    }
    public Tree$^D$ forSplit(Tree$^D$ l,Tree$^D$ r) {
        return new Split(l.accept(this),r.accept(this));
    }
}
```

Here is a 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 Fruit$^D$ occurs in a tree.

Even the visitors are no longer interesting.

```java
class $i_{\text{Occurs}}^v$ implements $i_{\text{TreeVisitor}}^D$ {
    Fruit$^D$ a;
    $i_{\text{Occurs}}^v$(Fruit$^D$ a) {
        a = .a;
    }
    public int forBud() {
        return 0;
    }
    public int forFlat(Fruit$^D$ f,Tree$^D$ t) {
        if (f.equals(a))
            return t.accept(this) + 1;
        else
            return t.accept(this);
    }
    public int forSplit(Tree$^D$ l,Tree$^D$ 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 Tree^D’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 Tree^D. 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 Tree^D.

```java
interface TreeVisitor^D {
    Object forBud();
    Object forFlat(Fruit^D f, Tree^D t);
    Object forSplit(Tree^D l, Tree^D r);
}
```

Here they are.

```java
class Flat extends Tree^D {
    Fruit^D f;
    Tree^D t;
    Flat(Fruit^D f, Tree^D t) {
        f = f;
        t = t;
    }

    Object accept(TreeVisitor^D 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;
    }

    Object accept(TreeVisitor^D ask) {
        return ask.forSplit(l, r);
    }
}
```

Define the remaining variants of Tree^D.
Good. Now define IsFlat\(^v\), an Object producing version of blsFlat\(^v\).

```java
class lsFlat\(^v\) implements TreeVisitor\(^2\) {
    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); }
}
```

That's no big deal.

And how about IsSplit\(^v\)?

```java
class lsSplit\(^v\) implements TreeVisitor\(^2\) {
    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); }
}
```

Now that's different. Here we need a way to determine the underlying boolean of the Boolean that is produced by l.accept(this) in the original definition.

Okay, here it is.

```java
class lsSplit\(^v\) implements TreeVisitor\(^2\) {
    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 l.accept(this) produces an Object, we must first convert\(^1\) it to a Boolean. Then we can determine the underlying boolean with the booleanValue method. We have seen this in chapter 5 when we converted an Object to a OneMoreThan.

Will the conversion always work?

```java
class lsSplit\(^v\) implements TreeVisitor\(^2\) {
    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); }
    }
```

Yes, because the Object produced by l.accept(this) is always a Boolean.

**The Seventh Bit of Advice**

*When designing visitor protocols for many different types, create a unifying protocol using Object.*

---

\(^1\) If Java had parametric polymorphism for methods, no downward cast would be necessary for our visitors (Martin Odersky and Philip Wadler, *Pizza into Java: Translating Theory into Practice*, Conference Record on Principles of Programming Languages, 146-159. Paris, 1997).
Did you think that was bad? Then study this definition during your next break.

```
class Occurs\textsuperscript{V} implements TreeVisitor\textsuperscript{J} {
    Fruit\textsuperscript{D} a;
    Occurs\textsuperscript{V}(Fruit\textsuperscript{D}.a) {
        a = .a;
    }

    public Object forBud() {
        return new Integer(0);
    }

    public Object forFlat(Fruit\textsuperscript{D} f, Tree\textsuperscript{D} t) {
        if (f.equals(a))
            return new Integer(((Integer) t.accept(this)).intValue() + 1);
        else
            return t.accept(this);
    }

    public Object forSplit(Tree\textsuperscript{D} l, Tree\textsuperscript{D} r) {
        return new Integer(((Integer) (l.accept(this)).intValue() + ((Integer) (r.accept(this)).intValue()));
    }
}
```

Oh my!
8.
Like Father, Like Son
What is the value of 
\[(7 + (4 - 3) \times 5)\]?

What is the value of 
\[(+ 7 (\times (- 4 3) 5))\]?

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)))})\)?

Where do the constructors come from?

Did you like that?

What is the value of 
\[({7,5} \cup (\{(4 \setminus 3) \cap \{5\})}\)?

What is the value of 
\[({7,5} \cup (\{(4 \setminus 3) \cap \{5\})}\)?

What is the value of 
\[({7,5} \times (\{4 \setminus \{3\} \{5\})}\)?

Like Father, Like Son
What is the value of
\[
\text{new Plus(}
\text{new Const(new Empty())}
\text{.add(new Integer(7)))}
\text{.add(new Integer(5))},
\text{new Prod(}
\text{new Diff(}
\text{new Const(new Empty())}
\text{.add(new Integer(4)))}
\text{.add(new Integer(3)))},
\text{new Const(new Empty())}
\text{.add(new Integer(5))))?}
\]

because we have just rewritten the previous expression using the constructors.

Where do the constructors come from?

A datatype and its variants that represent set expressions.

Do you still like it?

Sure, why not.

Does the arithmetic expression look like the set expression?

Yes, they look the same except for the constants:

\[
\text{new Plus(}
\text{new Const(\ast)},
\text{new Prod(}
\text{new Diff(}
\text{new Const(\ast)},
\text{new Const(\ast)}),
\text{new Const(\ast))))}.
\]

Let’s say that an expression is either

- a Plus(expr₁,expr₂),
- a Diff(expr₁,expr₂),
- a Prod(expr₁,expr₂), or
- a constant,

where expr₁ and expr₂ stand for arbitrary expressions. What should be the visitor interface?

That’s a tricky question.

```java
interface ExprVisitor {
    Object forPlus(ExprD l,ExprD r);
    Object forDiff(ExprD l,ExprD r);
    Object forProd(ExprD l,ExprD r);
    Object forConst(Object c);
}
```
Good answer. Here is the datatype now.

```java
abstract class ExprD {
    abstract
    Object accept(ExprVisitor<Expr> ask);
}
```

Define the variants of the datatype and equip them with an `accept` method that produces Objects.

```java
class Prod extends ExprD {
    ExprD l;
    ExprD r;
    Prod(ExprD l, ExprD r) {
        l = l;
        r = r;
    }
    Object accept(ExprVisitor<Expr> ask) {
        return ask.forProd(l, r);
    }
}
```

```java
class Plus extends ExprD {
    ExprD l;
    ExprD r;
    Plus(ExprD l, ExprD r) {
        l = l;
        r = r;
    }
    Object accept(ExprVisitor<Expr> ask) {
        return ask.forPlus(l, r);
    }
}
```

```java
class Diff extends ExprD {
    ExprD l;
    ExprD r;
    Diff(ExprD l, ExprD r) {
        l = l;
        r = r;
    }
    Object accept(ExprVisitor<Expr> ask) {
        return ask.forDiff(l, r);
    }
}
```

```java
class Const extends ExprD {
    Object c;
    Const(Object c) {
        c = c;
    }
    Object accept(ExprVisitor<Expr> ask) {
        return ask.forConst(c);
    }
}
```

*Like Father, Like Son*
Can we now define a visitor whose methods determine the value of an arithmetic expression?

Yes, we can. It must have four methods, one per variant, and it is like Occurs\(^V\) from the previous chapter.

How do we add

```
new Integer(3)
```

and

```
new Integer(2)
```

We have done this before. We use the method \(intValue\) to determine the \(\text{ints}\) that correspond to the \text{Integers}, and then add them together.

But what is the result of

```
new Integer(3).intValue() +
new Integer(2).intValue()
```

An int. what else?

How do we turn that into an \text{Integer}?

We use \text{new Integer(...)}.

Okay, so here is a skeleton of \text{IntEval}\(^V\).

```java
class IntEval\(^V\) implements ExprVisitor\(^I\) {
    public Object forPlus(Expr \(l\), Expr \(r\)) {
        return plus(l.accept(this),
                   r.accept(this)); }
    public Object forDiff(Expr \(l\), Expr \(r\)) {
        return diff(l.accept(this),
                   r.accept(this)); }
    public Object forProd(Expr \(l\), Expr \(r\)) {
        return prod(l.accept(this),
                   r.accept(this)); }
    public Object forConst(Object \(c\)) {
        return \(c\);
    }
    Object plus(Expr \(l\), Expr \(r\)) {
        return \(-1\);
    }
    Object diff(Expr \(l\), Expr \(r\)) {
        return \(-1\);
    }
    Object prod(Expr \(l\), Expr \(r\)) {
        return \(-1\);
    }
}
```

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 `Expr`s, determines their respective values, and `pluses` 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`, because that’s what

```latex
l.accept(this)
```

and

```latex
r.accept(this)
```

produce.

What must we put in the first and second blanks? `Object`.

Can we add `Objects`? No, we must convert them to `Integers` first and extract their underlying `ints`.

Can we convert all `Objects` to `Integers`? 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`

```latex
new Const(new Empty()),
new Const(new Integer(5))
```

`.accept(new IntEvalv())`?

`Wow. At some level, this is nonsense.`

Correct, so sometimes the conversion may fail, because we use an instance of `IntEvalv` on nonsensical arithmetic expressions.

`Like Father, Like Son` 121
We agree to avoid such arithmetic expressions.¹

¹ 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.

If we want to add \( l \) and \( r \), we write

\[
\text{new Integer}
\begin{align*}
&((\text{Integer})l).\text{intValue}() \\
+&((\text{Integer})r).\text{intValue}() 
\end{align*}
\]

Complete the definition now.

And their set expressions, too.

Now it's easy. Here we go.

```
class IntEval implements ExprVisitor { 
   public Object forPlus(Expr l,Expr r) { 
      return plus(l.accept(this),
             r.accept(this)); }
   public Object forDiff(Expr l,Expr r) { 
      return diff(l.accept(this),
             r.accept(this)); }
   public Object forProd(Expr l,Expr r) { 
      return prod(l.accept(this),
             r.accept(this)); }
   public Object forConst(Object c) { 
      return c; }
   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()); }
}```
We certainly need methods for `plusing`, `diffing`, and `proding` sets.

That’s correct, and here is everything.

```java
abstract class Set{
    Set add(Integer i) {
        if (mem(i))
            return this;
        else
            return new Add(i, this); }
    abstract boolean mem(Integer i);
    abstract Set plus(Set s);
    abstract Set diff(Set s);
    abstract Set prod(Set s);
}
```

Explain the method in the nested box in your own words.

```
We use our words:
“As its name says, `add` adds an element to a set. If the element is a `member` of the set, the set remains the same; otherwise, a `new` set is constructed with `Add`.”
```

Why is this so tricky?

```
Constructors always construct, and `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 i) {
        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 (t.mem(i))
            return s.diff(t);
        else
            return s.diff(t).add(i);
    }
    Set$^D$ prod(Set$^D$ t) {
        if (t.mem(i))
            return s.prod(t).add(i);
        else
            return s.prod(t);
    }
```

Chapter 8
Do we need to understand these definitions?  

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 $\text{IntEval}^V$ to obtain $\text{SetEval}^V$, an evaluator for set expressions?

Not much, just $\text{plus}$, $\text{diff}$, and $\text{prod}$.

How should we do that?

Oh, that’s a piece of pie. We just copy the definition of $\text{IntEval}^V$ and replace its $\text{plus}$, $\text{diff}$, and $\text{prod}$ methods.

That’s the worst way of doing that.

What?

Why should we throw away more than half of what we have?

That’s true. If we copied the definition and changed it, we would have identical copies of $\text{forPlus}$, $\text{forDiff}$, $\text{forProd}$, and $\text{forConst}$. We should reuse this definition.

Yes, and we are about to show you better ways. How do we have to change $\text{plus}$, $\text{diff}$, and $\text{prod}$?

That part is easy:

Object $\text{plus}(\text{Object}\ l,\text{Object}\ r)$ {
    return $((\text{Set}^D)l).\text{plus}((\text{Set}^D)r)$; }

and

Object $\text{diff}(\text{Object}\ l,\text{Object}\ r)$ {
    return $((\text{Set}^D)l).\text{diff}((\text{Set}^D)r)$; }

and

Object $\text{prod}(\text{Object}\ l,\text{Object}\ r)$ {
    return $((\text{Set}^D)l).\text{prod}((\text{Set}^D)r)$; }

\[\text{Like Father, Like Son}\]
Very good, and if we define $\text{SetEval}^v$ as an extension of $\text{IntEval}^v$, that's all we have to put inside of $\text{SetEval}^v$.

Now that's much easier than copying and modifying.

```
class SetEval$^v$ extends IntEval$^v$ {
    Object plus(Object l, Object r) {
        return ((Set$^D$)l).plus((Set$^D$)r); }
    Object diff(Object l, Object r) {
        return ((Set$^D$)l).diff((Set$^D$)r); }
    Object prod(Object l, Object r) {
        return ((Set$^D$)l).prod((Set$^D$)r); }
}
```

Is it like $\text{equals}$?

Yes, when we include $\text{equals}$ in our class definitions, we override the one in $\text{Object}$. Here, we override the methods $\text{plus}$, $\text{diff}$, and $\text{prod}$ as we extend $\text{IntEval}^v$.

How many methods from $\text{IntEval}^v$ are overridden in $\text{SetEval}^v$?

Three.

How many methods from $\text{IntEval}^v$ are not overridden in $\text{SetEval}^v$?

Four: $\text{forPlus}$, $\text{forDiff}$, $\text{forProd}$, and $\text{forConst}$.

Does $\text{SetEval}^v$ implement $\text{ExprVisitor}^T$?

It doesn't say so.

Does $\text{SetEval}^v$ extend $\text{IntEval}^v$?

It says so.

Does $\text{IntEval}^v$ implement $\text{ExprVisitor}^T$?

It says so.

Does $\text{SetEval}^v$ implement $\text{ExprVisitor}^T$?

By implication.
That’s correct. What is the value of
\[
\text{new Prod(}
  \text{new Const(new Empty())}
  \text{.add(new Integer(7))},
  \text{new Const(new Empty())}
  \text{.add(new Integer(3)))}
\text{.accept(new SetEval\textsuperscript{v}())?}
\]

Interesting question. How does this work now?

What type of value is
\[
\text{new Prod(}
  \text{new Const(new Empty())}
  \text{.add(new Integer(7))},
  \text{new Const(new Empty())}
  \text{.add(new Integer(3)))?}
\]

It is a Prod and therefore an Expr\textsuperscript{D}.

And what does \text{accept} consume?

An instance of SetEval\textsuperscript{v}, but its type is ExprVisitor\textsuperscript{T}.

What is
\[
\text{new SetEval\textsuperscript{v}().forProd(}
  \text{new Const(new Empty())}
  \text{.add(new Integer(7))},
  \text{new Const(new Empty())}
  \text{.add(new Integer(3)))?}
\]

That’s what we need to determine the value of next, because it is \textit{ask.forProd(l,r)}, with \textit{ask}, \textit{l}, and \textit{r} replaced by what they stand for.

Where is the definition of SetEval\textsuperscript{v}’s method \textit{forProd}?

It is in IntEval\textsuperscript{V}.

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?

If their values were \textit{A} and \textit{B}, we would have to determine the value of \textit{prod(A,B)}.
Isn’t that strange? 57 Why?

So far, we have always used a method on a particular object. 58 That’s true. What is the object with which we use \( prod(A,B) \)?

It is this object. 59 Oh, does that mean we should evaluate \( \text{new SetEval}^V().prod(A,B) \)?

Absolutely. If the use of a method omits the object, we take the one that we were working with before. 60 That clarifies things.

Good. And now what? 61 Now we still need to determine the values of

\[
\begin{align*}
\text{new Const(new Empty())} & \quad .add(\text{new Integer(7)}) \\
& \quad .accept(\text{this}) \\
\text{and} \\
\text{new Const(new Empty())} & \quad .add(\text{new Integer(3)}) \\
& \quad .accept(\text{this}).
\end{align*}
\]

The values are obviously

\[
\begin{align*}
\text{new Empty()} & \quad .add(\text{new Integer(7)}) \\
\text{and} \\
\text{new Empty()} & \quad .add(\text{new Integer(3)}).
\end{align*}
\]

Where is the definition of forConst that determines these values? 62 It, too, is in IntEval^V.

Here is the next expression in our sequence: 63 The object is an instance of SetEval^V, which overrides the \( prod \) method in IntEval^V with its own.

\[
\begin{align*}
\text{new SetEval}^V() & \quad .prod(\text{new Empty()} \\
& \quad .add(\text{new Integer(7)}), \\
& \quad \text{new Empty()} \\
& \quad .add(\text{new Integer(3)})).
\end{align*}
\]

Where does \( prod \) come from?
What next?

Next we need to determine the value of

\[
((S^p)(\text{new Empty()})
  .add(\text{new Integer(7)})))
  .prod((S^p)(\text{new Empty()})
  .add(\text{new Integer(3)})),
\]

because it is

\[
((S^p)l.\text{accept(this)})
  .prod((S^p)r.\text{accept(this)})
\]

with \( l.\text{accept(this)} \) and \( r.\text{accept(this)} \) replaced by their respective values.

Is \text{new Empty().add(new Integer(7))} an instance of \( S^p \)?

Of course it is, but the type of \( l.\text{accept(this)} \), which is where it comes from, is \text{Object}.

And how about \text{new Empty().add(new Integer(3))}? It's the same.

And that is why the method must contain a conversion from \text{Object} to \( S^p \).s.

This example makes the need for conversions obvious again.

Time for the last question. Where does this \text{prod} come from now?

This one belongs to \( S^p \) or more precisely its \text{Empty} and \text{Add} variants.

And what does \text{prod} do?

It determines the intersection of one \( S^p \) with another \( S^p \), but didn’t we agree that the previous question was the last question on that topic?

We overrode that, too.

Thanks, guys.

Is it natural that \( S^v \text{Eval} \) extends \( I^n \text{Eval} \)?

No, not at all.

Like Father, Like Son
Why did we do that? 

Because we defined IntEval\textsuperscript{V} 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 IntEval\textsuperscript{V} from SetEval\textsuperscript{V}? 

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.

Isn't this abstract class like Point\textsuperscript{D}? 

abstract class Eval\textsuperscript{D} 

implements ExprVisitor\textsuperscript{D} 

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; 

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 \( \text{Eval}^D \) 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 \( \text{IntEval}^V \) extending \( \text{Eval}^D \).

```java
class IntEval^V extends Eval^D {
    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 \( \text{SetEval}^V \).

```java
class SetEval^V extends Eval^D {
    Object plus(Object l, Object r) {
        return ((Set^D)l).plus((Set^D)r);}
    Object diff(Object l, Object r) {
        return ((Set^D)l).diff((Set^D)r);}
    Object prod(Object l, Object r) {
        return ((Set^D)l).prod((Set^D)r);}
}
```

Is it natural for two evaluators to be on the same footing?

Time for supper.

If you are neither hungry nor tired, you may continue.

*Like Father, Like Son*
Remember $\text{Subst}^V$ 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 $\text{LtdSubst}^V$, 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 $\text{forBot}$.

Where do they differ?  
They differ in $\text{forTop}$, but $\text{LtdSubst}^V$ 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.
Define the **abstract class** Subst, which contains all the common pieces and specifies what a concrete pie substituter must contain in addition.

```java
abstract class Subst

    implements PieVisitor{

    Object n;
    Object o;

    public Pie forBot() {
        return new Bot();
    }

    public abstract Pie forTop(Object t, Pie r);

}
```

It's not a big deal, except for the fields.

```java
abstract class Subst

    implements PieVisitor

    {

    Object n;
    Object o;

    public Pie forBot()

    {

        return new Bot();

    }

    public abstract Pie forTop(Object t, Pie r);

}
```

We can define Subst by extending Subst.

```java
class SubstV extends Subst {

    SubstV(Object n, Object o) {

        n = .n;
        o = .o;
    }

    public Pie forTop(Object t, Pie r) {

        if (o.equals(t))
            return new Top(n, r.accept(this));
        else
            return new Top(t, r.accept(this));
    }
}
```

It also extends Subst.

```java
class SubstV extends Subst {

    int c;

    SubstV(int c, Object n, Object o) {

        n = .n;
        o = .o;
        c = .c;
    }

    public Pie forTop(Object t, Pie r) {

        if (c == 0)
            return new Top(t, r);
        else
            if (o.equals(t))
                return new Top(n, r.accept(new SubstV(c - 1, n, o)));
            else
                return new Top(t, r.accept(this));
    }
}
```

Define LtdSubstV.

Do the two remaining classes still have things in common? No, but the constructors have some overlap. Shouldn't we lift the SubstV constructor into Subst, because it holds the common elements?

---

*Like Father, Like Son*
That’s a great idea. Here is the new version of Subst$^n$.

```java
abstract class Subst$^n$
    implements PieVisitor$^n$
{
    Object n;
    Object o;
    Subst$^n$(Object _n,Object _o) {
        n = _n;
        o = _o;
    }

    public Pie$^n$ forBot() {
        return new Bot();
    }

    public abstract Pie$^n$ forTop(Object t,Pie$^n$ r);
}
```

We must use super in the constructors.

```java
class Subst$^n$ extends Subst$^n$
{
    Subst$^n$(Object _n,Object _o) {
        super(_n,_o);
    }

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

Revise Subst$^n$ and LtdSubst$^n$.

```java
class LtdSubst$^n$ extends Subst$^n$
{
    int c;
    LtdSubst$^n$(int _c,Object _n,Object _o) {
        super(_n,_o);
        c = _c;
    }

    public Pie$^n$ forTop(Object t,Pie$^n$ r) {
        if (c == 0)
            return new Top(t,r);
        else
            if (o.equals(t))
                return new Top(n,
                               r.accept(
                               new LtdSubst$^n$(c-1,n,o)));
            else
                return new Top(t,r.accept(this));
    }
}
```

Was that first part easy? 

90 As pie.
That's neat. How about some art work? Is this called a pie chart?

No, but the picture captures the important relationships.

Is it also possible to define $\text{LtdSubst}^\gamma$ as an extension of $\text{Subst}^\gamma$?

It may even be better. In some sense, $\text{LtdSubst}^\gamma$ just adds a service to $\text{Subst}^\gamma$: It counts as it substitutes.

If $\text{LtdSubst}^\gamma$ is defined as an extension of $\text{Subst}^\gamma$, what has to be added and what has to be changed?

As we just said, $c$ is an addition and $\text{forTop}$ 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 $\text{Subst}^V$ from chapter 6 one more time.

```java
class $\text{Subst}^V$ implements PieVisitor$^T$ {
    Object $n$;
    Object $o$;
    $\text{Subst}^V$(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 ($o.\text{equals}(t)$)
            return new Top($n$, $r.\text{accept}(this)$);
        else
            return new Top($t$, $r.\text{accept}(this)$);
    }
}
```

Define $\text{LtdSubst}^V$ as an extension of $\text{Subst}^V$.

```java
class $\text{LtdSubst}^V$ extends $\text{Subst}^V$ {
    int $c$;
    $\text{LtdSubst}^V$(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 ($o.\text{equals}(t)$)
                return new Top($n$,
                    $r.\text{accept}(new \text{LtdSubst}^V(c - 1, n, o))$);
            else
                return new Top($t$, $r.\text{accept}(this)$);
    }
}
```

Let's draw a picture.

You deserve a super-deluxe pizza now.
9. Be a Good Visitor
Remember Point\textsuperscript{D}? If not, here is the
datatype with one additional method, `minus`.
We will talk about `minus` when we need it,
but for now, just recall Point\textsuperscript{D}’s variants.

```java
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() <= p.distanceToO();
  }

  Point\textsuperscript{D} minus(Point\textsuperscript{D} p) {
    return new CartesianPt(x - p.x,y - p.y);
  }

  abstract int distanceToO();
}
```

Good. Take a look at this extension of
ManhattanPt.

```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;
  }
}
```

What is unusual about the constructor?

---

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.

   ```java
   class CartesianPt extends Point\textsuperscript{D} {
     CartesianPt(int _x,int _y) {
       super(_x,_y);
     }

     int distanceToO() {
       return \sqrt{x^2 + y^2};
     }
   }
   ```

   class ManhattanPt extends Point\textsuperscript{D} {
     ManhattanPt(int _x,int _y) {
       super(_x,_y);
     }

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

2. It uses
   \[ Δx = Δ\textsubscript{x}; \]
   \[ Δy = Δ\textsubscript{y}; \]
   in addition to super(_x,_y).
And what does that mean? 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? It is a ManhattanPt with two additional fields: Δx and Δy. 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)? Yes.

What is unusual about distanceToO? 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 distanceToO that is relevant in the class that ShadowedManhattanPt extends.

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 distanceToO? 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()? 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.

```java
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?

10 Nothing. We just discussed this kind of constructor for ShadowedManhattanPt.

Is this a ShadowedCartesianPt:

new ShadowedCartesianPt(12,5,3,4)?

Yes.

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
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.

```
class ShadowedCartesianPt
    extends CartesianPt {
    int \( \Delta_x \);
    int \( \Delta_y \);
    ShadowedCartesianPt(int \_x,\_y,\_\( \Delta_x \),\_\( \Delta_y \)) {
        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.\(^1\)

\(^1\) 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))?

19 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? 20 That’s obvious.

Is the rest of this chapter obvious, too? 21 What?

That was a hint that now is a good time to take a break.

Come back fully rested. You will more than need it.

Are sandwiches square meals for you? 23 Fine.

They can be well-rounded.

Here are circles and squares.

```java
class Circle extends ShapeD {
  int r;
  Circle(int .r) {
    r = .r;
  }

  boolean accept(ShapeVisitor ask) {
    return ask.forCircle(r);
  }
}

class Square extends ShapeD {
  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 ShapeD {
  abstract boolean accept(ShapeVisitor ask);
}
```
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(Point<Shape> s);
}
```

Yes and we will need this third variant.

```java
class Trans1 extends ShapeD {
    Point<Shape> q;
    ShapeD s;
    Trans(Point<Shape> q, ShapeD s) {
        q = -q;
        s = -s;
    }
    boolean accept(ShapeVisitor<T ask) {
        return ask.forTrans(q,s);
    }
}
```

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?

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?

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?

No, we also need to do that for the $y$ coordinate.

Aren’t we on a roll?

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 \texttt{new CartesianPt(5,6)} again. Can we think of this point as the origin?

We can if we translate all other points by an appropriate amount. By how much?

By 5 in the $x$ direction and 6 in the $y$ direction, respectively.

How could we translate the points by an appropriate amount?

We could subtract the appropriate amount from each point.

Is there a method in Point$^D$ that accomplishes that?

Yes. Is that why we included \texttt{minus} in the new definition of Point$^D$?
Indeed. And now we can define the visitor HasPt\(^v\), whose methods determine whether some Shape\(^p\) has a Point\(^p\) inside of it.

```java
class HasPt\(^v\) implements ShapeVisitor\(^p\) {
    Point\(^p\) p;
    HasPt\(^v\)(Point\(^p\) _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\(^p\) q, Shape\(^p\) s) {
        return s.accept(
            new HasPt\(^v\)(p.minus(q)));
    }
}
```

1 We could have written the if... as return \(p.x <= s\) \&\& \(p.y <= s\).

We could have written the if... as return \(p.x <= s\) \&\& \(p.y <= s\).

What is the value of

```java
new Circle(10)
    .accept(
        new HasPt\(^v\)(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

```java
new Square(10)
    .accept(
        new HasPt\(^v\)(new CartesianPt(10,10)));
```

true.

Let’s consider something a bit more interesting. What is the value of

```java
new Trans(  
    new CartesianPt(5,6),
    new Circle(10))
    .accept(
    new HasPt\(^v\)(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:
\[
\text{new Trans(}
\text{new CartesianPt}(5,4),
\text{new Trans(}
\text{new CartesianPt}(5,6),
\text{new Circle}(10))))
\text{.accept(}
\text{new HasPt}(\text{new CartesianPt}(10,10)))?
\]

But what is the value?  

First, we have to find out whether
\[
\text{new Trans(}
\text{new CartesianPt}(5,6),
\text{new Circle}(10))\text{.accept(}
\text{new HasPt}(\text{new CartesianPt}(5,6)))
\]
is true or false.

Second, we need to look at
\[
\text{new Circle}(10)\text{.accept(}
\text{new HasPt}(\text{new CartesianPt}(0,0))),
\]
but the value of this is obviously true.

Very good. Can we nest Trans three times?  

Ten times, if we wish, because a Trans contains a Shape, and that allows us to nest things as often as needed.

Ready to begin?  

What? Wasn’t that it?

No. The exciting part is about to start.  

We are all eyes.

How can we project a cube of cheese to a piece of paper?  

It becomes a square, obviously.

And the orange on top?  

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.

```
class Union extends ShapeD {
    ShapeD s;
    ShapeD t;
    Union(ShapeD _s,ShapeD _t) {
        s = _s;
        t = _t;
    }
    boolean accept(ShapeVisitorD ask) {
        return _________;
    }
}
```

What do we know from Circle, Square, and Trans about `accept`?

That looks obvious after the fact. But why is there a blank in `accept`?

We know that a `ShapeVisitorD` 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 `ShapeVisitorD` 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 `ShapeVisitorD`.

Why can’t we change it?

In that case, we’re stuck.

Just to make the problem more interesting.

Be a Good Visitor

149
We would be stuck, but fortunately we can extend interfaces. Take a look at this.

```
interface UnionVisitor^2
    extends ShapeVisitor^2 {
        boolean forUnion(Shape^D s, Shape^D 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 ShapeVisitor^2 and the additional one named forUnion.

```
class Union extends Shape^D {
    Shape^D s;
    Shape^D t;
    Union(Shape^D s, Shape^D t) {
        s = s;
        t = t;
    }

    boolean accept(ShapeVisitor^2 ask) {
        return ((UnionVisitor^2)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? No. It does not provide the method forUnion.

Define UnionHasPt, which extends HasPt with an appropriate method forUnion.

```java
class UnionHasPt extends HasPt {
    UnionHasPt(Point p) {
        super(p); }
    boolean forUnion(Shape s, Shape 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 UnionHasPt contain forUnion? Of course, we just put it in.
Is UnionHasPt a UnionVisitor?  

Correct, but unfortunately we have to add three more words to make this explicit.

```java
class UnionHasPt extends HasPt
    implements UnionVisitor {
    UnionHasPt(Point p) {
        super(p); }

    public boolean forUnion(Shape s, Shape t) {
        if (s.accept(this))
            return true;
        else
            return t.accept(this); }
```

It provides the required methods: forCircle, forSquare, forTrans, and forUnion.

The first two additional words have an obvious meaning. They explicitly say that this visitor provides the services of UnionVisitor. 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(
            new CartesianPt(13,17)))?
```

We know how forTrans works, so we're really asking whether

```
    new CartesianPt(10,10)
```

is inside the Union shape.

So?

```
Which means that we're asking whether
    new CartesianPt(10,10)
```

is inside of

```
    new Square(10)
```

or inside of

```
    new Circle(10).
```
Okay. And what should be the answer?  

Let's see whether the value of  
\[ \text{new Trans(} \text{new CartesianPt(3,7),} \]  
\[ \text{new Union(} \text{new Square(10),} \]  
\[ \text{new Circle(10))Accept(} \]  
\[ \text{new UnionHasPt}(\text{new CartesianPt(13,17)))} \]  
is true?  

And?  

How did we construct this shape?  

Which method should we use on it?  

Where is \( \text{forTrans} \) defined?  

So what should we do now?  

What type of object is  
\[ \text{new Union(} \text{new Square(10),} \]  
\[ \text{new Circle(10))} \]?
How did we construct this Shape? "With Union.

So which method should we use on it? "forUnion, of course.

How do we find the appropriate forUnion method? In accept, which is defined in Union, we confirm that

```
new HasPtV(
    new CartesianPt(10,10))
```

is a UnionVisitor and then invoke its forUnion.

Is an instance of HasPtV a UnionVisitor? No!

Does it contain a method forUnion? No!

Then what is the value of

```
new Union(
    new Square(10),
    new Circle(10))
.accept(
    new HasPtV(
        new CartesianPt(10,10)))
```

It doesn't have a value. We are stuck.1

1 A Java program raises a RuntimeException, indicating that the attempt to confirm the UnionVisitorIness of the object failed. More specifically, we would see the following when running the program:

```
java.lang.ClassCastException: UnionsHasPtV
    at Unions.accept(...java:...)
    at UnionsHasPtV.forTrans(...java:...)
    at Trans.accept(...java:...)
```


Which of those is best? You guessed it: whatever you did is best.

We should have prepared this extension in a better way. How could we have done that?
In two ways. First, it contains a new method: \texttt{newHasPt}. Second, it uses the new method in place of \texttt{new HasPt\textasciitilde v} in \texttt{forTrans}.

Here is the definition of \texttt{HasPt\textasciitilde v} that we should have provided if we wanted to extend it without making changes.

```java
class HasPt\textasciitilde v implements ShapeVisitor\xspace {
    Point\textsuperscript{D} p;
    HasPt\textasciitilde v(Point\textsuperscript{D} \_p) {
        p = \_p;
    }
    ShapeVisitor\xspace newHasPt(Point\textsuperscript{D} p) {
        return new HasPt\textasciitilde v(p);
    }
    public boolean forCircle(int r) {
        return p.distanceToOQ < r;
    }
    public boolean forSquare(int s) {
        if (p.x < 5)
            return (p.y < 5);
        else
            return false;
    }
    public boolean forTrans(Point\textsuperscript{D} q, Shape\textasciitilde p) {
        return s.accept(newHasPt(p.minus(q))); }
}
```

How does this definition differ from the previous one?

---

Good. What does \texttt{newHasPt} produce? A new \texttt{ShapeVisitor\xspace}, as its interface implies.

And how does it produce that? By constructing a \texttt{new} instance of \texttt{HasPt\textasciitilde v}.

Is \texttt{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 $\text{HasPt}_v$ and the previous one are really the same? They are mostly indistinguishable. Both $\text{forTranses}$, the one in the previous and the one in the new definition of $\text{HasPt}_v$, produce the same values when they consume the same values.

A functional programmer would say that $\text{newHasPt}$ and $\text{HasPt}_v$ are $\eta$-equivalent.

Very well. But how does that help us with our problem? That's not obvious.

Can we override $\text{newHasPt}$ when we extend $\text{HasPt}_v$? Yes, we can override any method that we wish to override.

Let's override $\text{newHasPt}$ in $\text{UnionHasPt}_v$. When we override it, we need to make sure it produces a $\text{ShapeVisitor}$. The latter. Then $\text{forTrans}$ in $\text{HasPt}_v$ keeps producing a $\text{UnionHasPt}_v$, if we start with a $\text{UnionHasPt}_v$.

That's true. Should it produce a $\text{HasPt}_v$ or a $\text{UnionHasPt}_v$? Let's just reread it.

The Ninth Bit of Advice

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`. Here it is.

```java
class UnionHasPtV
    extends HasPtV
    implements UnionVisitorZ {
    UnionHasPtV(PointD p) {
        super(p);
    }

    ShapeVisitorZ newHasPt(PointD p) {
        return new UnionHasPtV(p);
    }

    public boolean forUnion(Shape s, Shape t) {
        if (s.accept(this))
            return true;
        else
            return t.accept(this);
    }
}
```

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.

---

1. The is an instance of the factory method pattern [4].

---

*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\textsuperscript{\textsc{v}} and how it is overridden in UnionHasPt\textsuperscript{\textsc{v}}.

Is anything missing?

Square, but that's okay.

Let's see whether this definition works.

What is the value of

\begin{verbatim}
new Trans(
    new CartesianPt(3,7),
    new Union(
        new Square(10),
        new Circle(10))
    .accept(
        new UnionHasPt\textsuperscript{\textsc{v}}(
            new CartesianPt(13,17)))?
\end{verbatim}

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\textsuperscript{\textsc{v}}.

So what should we do now?

We should determine the value of

\begin{verbatim}
new Union(
    new Square(10),
    new Circle(10))
    .accept(
        this.newHasPt(
            new CartesianPt(10,10))).
\end{verbatim}

What is this?

The current visitor, of course.

And how does that work?

We determine the value of

\begin{verbatim}
this.newHasPt(
    new CartesianPt(10,10))
\end{verbatim}

and then use accept for the rest.
The new UnionVisitor:

\[
\text{new UnionHasPt}^V(\text{new CartesianPt}(10,10)).
\]

And what do we create?

\[
\text{new UnionHasPt}^V \text{ also satisfies the interface ShapeVisitor}, \text{ so now we can invoke the } \text{forUnion method}.
\]

What is the value of

\[
\text{new Union(}
\text{new Square}(10),
\text{new Circle}(10))
.\text{accept(}
\text{new UnionHasPt}^V(\text{new CartesianPt}(10,10))))?
\]

How do we do that?

\[
\text{We first determine the value of}
\text{new Square}(10)
.\text{accept(}
\text{new UnionHasPt}^V(\text{new CartesianPt}(10,10))).
\]

If it is true, we’re done.

Is it true?

\[
\text{It is. So we’re done and we got the value we expected.}
\]

Are we happy now?

\[
\text{Ecstatic.}
\]

Is it good to have extensible definitions?

\[
\text{Yes. People should use extensible definitions if they want their code to be used more than once.}
\]

Very well. Does this mean we can put together flexible and extensible definitions if we use visitor protocols with these constructor-like methods?

\[
\text{Yes, we can and should always do so.}
\]

And why is that?

\[
\text{Because no program is ever finished.}
\]

Are you hungry yet?

\[
\text{Are our meals ever finished?}
\]

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. This is beyond anything we have seen before.

```java
class Pieman^M implements Pieman^T {
    Pie^D p = new Bot();
    public int addTop(Object t) {
        p = new Top(t, p);
        return occTop(t);
    }
    public int remTop(Object t) {
        p = (Pie^D)p.accept(new Rem^V(t))
        return occTop(t);
    }
    public int substTop(Object n, Object o) {
        p = (Pie^D)p.accept(new Subst^V(n, o))
        return occTop(n);
    }
    public int occTop(Object o) {
        return ((Integer)p.accept(new Occurs^V(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^M.

How so? Haven't we seen Pie^D, Top, and Bot before? We have seen them.

And haven't we seen visitors like Rem^V, Subst^V, and Occurs^V 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 PiemanM.

```java
interface PiemanM {  
    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^1 and Pie^1.

```java
interface PieVisitor^1 {  
    Object forBot();  
    Object forTop(Object t,Pie^1 r);  
}
```

"They are very familiar."

```java
abstract class Pie^1 {  
    abstract Object accept(PieVisitor^1 ask);  
}
```

Define Bot and Top.

```java
class Bot extends Pie^1 {  
    Object accept(PieVisitor^1 ask) {  
        return ask.forBot();  
    }
}
```

```java
class Top extends Pie^1 {  
    Object t: Pie^1 Top(Object _t,Pie^1 _r) {  
        t = _t;
        r = _r;  
    }
    Object accept(PieVisitor^1 ask) {  
        return ask.forTop(t,r);  
    }
}
```

"Isn't it missing p'"
Here is Occurs\textsuperscript{V}. It counts how often some topping occurs on a pie.

\begin{verbatim}
class Occurs\textsuperscript{V} implements PieVisitor\textsuperscript{T} {
    Object a;
    Occurs\textsuperscript{V}(Object _a) {
        a = _a;
    }

    public Object forBot() {
        return new Integer(0); }
    public Object forTop(Object t, Pie\textsuperscript{D} r) {
        if (t.equals(a))
            return new Integer(((Integer)
                (r.accept(this)))
                .intValue() + 1);
        else
            return r.accept(this); }
}
\end{verbatim}

Great! Now we have almost all the visitors for our pieman. Define Rem\textsuperscript{V}, which removes a topping from a pie.

\begin{verbatim}
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 new Top(n,(Pie\textsuperscript{D})r.accept(this));
        else
            return new Top(t,(Pie\textsuperscript{D})r.accept(this)); }
}
\end{verbatim}

We remember that one, too.

\begin{verbatim}
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)); }
}
\end{verbatim}
Now we are ready to talk. What is the value of

\texttt{new Pieman^M().occTop(new Anchovy())}?  

We first create a \texttt{Pieman^M} and then ask how many anchovies occur on the pie.

Which pie?  

The pie named \texttt{p} in the new \texttt{Pieman^M}.

And how many anchovies are on that pie?  

None.

And what is the value of \texttt{new Pieman^M().addTop(new Anchovy())}?  

That's where those stand-alone semicolons come in again. They were never explained.

True. If we wish to determine the value of \texttt{new Pieman^M().addTop(new Anchovy())}, we must understand what

\[ p = \texttt{new Top(new Anchovy(),p)} ; \]

\texttt{return occTop(new Anchovy())}  

means?

Yes, we must understand that. There is no number \( x \) in the world for which

\[ x = x + 1, \]

so why should we expect there to be a Java \texttt{p} such that

\[ p = \texttt{new Top(new Anchovy(),p)}? \]

That's right. But that's what happens when you have one too many double espressos.

So what does it mean?

Here it means that \texttt{p} changes and that future references to \texttt{p} reflect the change.

And the change is that \texttt{p} has a new topping, right?

When does the future begin?  

Does it begin below the stand-alone semicolon?

That's precisely what a stand-alone semicolon means. Now do we know what

\texttt{return occTop(new Anchovy())}  

produces?

It produces the number of anchovies on \texttt{p}.  

Chapter 10
And how many are there?  

And now what is the value of  
\[ \text{new Pieman}^M().\text{addTop(new Anchovy())?} \]

It's 2, isn't it?

No, it's not. Take a close look. We created a new pieman, and that pieman added only one anchovy to his p.

Oh, isn't there a way to place several requests with the same pieman?

Yes, there is. Take a look at this:
\[ \text{Pieman}^y \text{ } y = \text{new Pieman}^M(). \]

Okay, y stands for some pieman.

What is the value of  
\[ y.\text{addTop(new Anchovy())?} \]

1. We know that.

And now what is the value of  
\[ y.\text{substTop(new Tuna(), new Anchovy())?} \]

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

0, because y's pie no longer contains any anchovies.

Very good. And now take a look at this:
\[ \text{Pieman}^y \text{ } y = \text{new Pieman}^M(). \]

What is the value of  
\[ y.y.\text{addTop(new Anchovy())} ; \]
\[ y.y.\text{addTop(new Anchovy())} ; \]
\[ y.y.\text{addTop(new Anchovy())} ; \]
\[ y.y.\text{addTop(new Salmon())} ; \]

What are the doing at the end?
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())
; yysubstTop(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?

It’s 0, because remTop first removes all tunas and then counts how many there are left.

Does that mean remTop always produces 0? Yes, it always does.

Now what is the value of

```java
yy.occTop(new Salmon())
```

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? No, it changed.

So is it the same one? 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?

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?

No, because yy’s type is PiemanT, p isn’t available. Only addTop, remTop, substTop, and occTop are visible.
Isn't it good that we didn't include \( p \) in \( \text{Pieman}^2 \)?

37 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?

38 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?

39 We can't do that yet.

And that's what we discuss next. Do you need a break?

40 No, a cup of coffee will do.

Compare this new \( \text{PieVisitor}^2 \) with the first one in this chapter.

41 It isn't all that different. A \( \text{PieVisitor}^2 \) must still provide two methods: \( \text{forBot} \) and \( \text{forTop} \), except that the former now consumes a \( \text{Bot} \) and the latter a \( \text{Top} \).

interface \( \text{PieVisitor}^2 \) {
  Object \( \text{forBot}(\text{Bot} \, \text{that}); \)
  Object \( \text{forTop}(\text{Top} \, \text{that}); \)
}

42 The definition is straightforward.

abstract class \( \text{Pie}^D \) {
  abstract
  Object \( \text{accept}(\text{PieVisitor}^2 \, \text{ask}); \)
}

class \( \text{Bot} \) extends \( \text{Pie}^D \) {
  Object \( \text{accept}(\text{PieVisitor}^2 \, \text{ask}) \) {
    return \( \text{ask} \, \text{forBot}(\text{this}); \) }
}

Define the Bot variant.

43 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*
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(PieVisitorZ ask) {
        return ask.forTop(this);
    }
}
```

Modify this version of Occurs\textsuperscript{v} so that it implements the new PieVisitor\textsuperscript{T}.

```java
class Occurs\textsuperscript{v} implements PieVisitor\textsuperscript{T} {
    Object a;
    Occurs\textsuperscript{v}(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 Occurs\textsuperscript{v} implements PieVisitor\textsuperscript{T} {
    Object a;
    Occurs\textsuperscript{v}(Object _a) {
        a = _a;
    }

    public Object forBot(Bot that) {
        return new Integer(0);
    }

    public Object forTop(Top 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? It now consumes a Bot, which is why we had to add (Bot that) behind its name.
How does `forTop` change?

It no longer receives the field values of the corresponding `Top`. Instead it consumes the entire object, which makes the two fields available as `that.t` and `that.r`.

And?

With that, we can replace the fields `t` and `r` with `that.t` and `that.r`.

Isn't that easy?

This modification of `Occurs^v` certainly is.

Then try `Rem^v`.

It's easy; we use the same trick.

```java
class Rem^v implements PieVisitor^T {
    Object o;
    Rem^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^v)that.r.accept(this)); }
}
```

Do we need to do `Subst^v`?

Not really. It should be just like `Rem^v`.

And indeed, it is. Happy now?

So far, so good. But what's the point of this exercise?

Oh, Point^2^s? They will show up later.

Seriously.

\textit{The State of Things to Come}
Here is the point. What is new about this version of Subst\(^v\)?

```java
class Subst\(^v\) implements PieVisitor\(^x\) {
    Object n;
    Object o;
    Subst\(^v\) (Object \_n, Object \_o) {
        n = \_n;
        o = \_o;
    }

    public Object forBot(Bot that) {
        return that;
    }
    public Object forTop(Top that) {
        if (o.equals(that)) {
            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?”

Yes, because we want to define a version of Subst\(^v\) that modifies toppings without constructing a new pie. [55] Does this saying apply here, too?

That’s a way of putting it.

What do the methods of Subst\(^v\) 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.

Chapter 10
Correct. And from here on, that.t holds the new topping. What is that.r.accept(this) about?

In the previous Subst, 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?

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 PiemanM?

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 PiemanM manages the toppings of p, and nobody else sees p.

Can we do LtdSubst now without creating new instances of LtdSubst 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();
}
```

```java
class CartesianPt extends PointD {
    super(_x, _y);
    int distanceToO() {
        return \[x^2 + y^2\];
    }
}
```

```java
class ManhattanPt extends PointD {
    super(_x, _y);
    int distanceToO() {
        return x + y;
    }
}
```

```java
class ShadowedManhattanPt extends ManhattanPt {
    int \Delta_x;
    int \Delta_y;
    ShadowedManhattanPt(int _x, int \_x, int \_y) {
        super(_x, _y);
        \Delta_x = \_x;
        \Delta_y = \_y;
    }
    int distanceToO() {
        return super.distanceToO() + \Delta_x + \Delta_y;
    }
}
```

Aren't we missing a variant?  Yes, we are missing ShadowedCartesianPt.
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.

First we must know what the method is supposed to produce.

The method should return the new distance to the origin.

```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();
}
```

Let `ptChild` stand for

```java
new ManhattanPt(1,4).
```

What is the value of

```java
ptChild.distanceToO();
```
What is the value of $ptChild.moveBy(2,8)$?

Good. Now let’s watch a child with a helium-filled balloon that casts a shadow. Let $ptChildBalloon$ be $new ShadowedManhattanPt(1,4,1,1)$. What is the value of $ptChildBalloon.distanceToO()$?

What is the value of $ptChildBalloon.moveBy(2,8)$?

Did the balloon move, too?

Isn’t that powerful?

The more things change, the cheaper our desserts get.

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.

No, we wouldn’t forget factorial.

```java
interface T^2 { 
  o->o^2 apply(T^2 x); 
}

interface o->o^2 { 
  Object apply(Object x); 
}

interface oo->oo^2 { 
  o->o^2 apply(oo->oo^2 x); 
}

interface oo->oo^2 { 
  o->o^2 apply(oo->oo^2 x); 
}

class Y implements oo->oo^2 { 
  public o->o^2 apply(oo->oo^2 f) { 
    return new H(f).apply(new H(f)); 
  }
}

class H implements T^2 { 
  oo->oo^2 f; 
  H(oo->oo^2 f) { 
    f = f; 
  } 
  public o->o^2 apply(T^2 x) { 
    return f.apply(new G(x)); 
  }
}

class G implements o->o^2 { 
  T^2 x; 
  G(T^2 x) { 
    x = x; 
  } 
  public Object apply(Object y) { 
    return (x.apply(x)).apply(y); 
  }
}

class MkFact implements oo->oo^2 { 
  public o->o^2 apply(o->o^2 fact) { 
    return new Fact(fact); 
  }
}

class Fact implements o->o^2 { 
  o->o^2 fact; 
  Fact(o->o^2 fact) { 
    fact = fact; 
  } 
  public Object apply(Object i) { 
    int inti = ((Integer)i).intValue(); 
    if (inti == 0) 
      return new Integer(1); 
    else 
      return 
        new Integer( 
          inti 
        * 
          ((Integer) 
            fact.apply(new Integer(inti - 1)) 
          .intValue()); 
        )
  }
```
Index

Add, 124
addTop, 161
Anchovy, 43, 53, 64, 65, 69, 72
Apple, 100
Base, 10
bHasFruit$, 104
bIsFlat$, 102
bIsSplit$, 102, 103
Bot, 69, 78, 83, 85, 86, 91, 93, 162, 167
Brass, 29
Bud, 100, 101, 107, 110, 112
bTreeVisitor$, 101
CartesianPt, 5, 13, 37, 38–40, 139, 172
Cheese, 43, 53, 64, 65
Circle, 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$, 130
Expr$, 119
ExprVisitor$, 118
Fig, 100
Fish$, 69, 72
Flat, 100, 101, 108, 110, 112
Fruit$, 100
Gold, 29
HasPt$, 147, 155
Holder, 28, 36
iHasHeight$, 104
IntEval$, 120, 122, 131
iOccurs$, 111
IsFlat$, 113
IsSplit$, 113
isVegetarian, 25, 27
isVegetarian, 63
isVeggie, 30, 31, 36
TreeVisitor$, 107
Kebab$, 28, 36
Lamb, 16, 27, 28, 36, 59, 62
Layer$, 10
Lemon, 100
LdtSubPt$, 95, 97, 132–136
ManhattanPt, 5, 13, 37, 38, 39, 139, 172
mem, 124
minus, 139
moveBy, 173
newHasPt, 155, 157
Num$, 7, 79
occTop, 161
Occurs$, 114, 163, 168
Olive, 43, 53, 64, 65
OneMoreThan, 7, 79
Onion, 16, 27, 28, 36, 59, 62
onlyOnions, 18, 27
OnlyOnions$, 57, 59
Peach, 100
Pear, 100
Pepper, 4, 37
Pie$, 69, 78, 83, 85, 86, 91, 93, 162, 167
PieVisitor$, 92, 162, 167
PieMan$, 162
PieMan$, 161
Pizza$, 43, 53, 63–65
Plate$, 29
plus, 124
Plus, 119
Point$, 5, 13, 40, 130, 172, 173
proof, 124
Prod, 119
Radish, 28, 36
remA, 45, 49, 70
RemA\textsuperscript{V}, 66, 71, 73
RemFish\textsuperscript{V}, 74
Rem\textsuperscript{V}, 77, 87, 93, 163, 169
RemInt\textsuperscript{V}, 76
rem\textsuperscript{Top}, 161
Rod\textsuperscript{P}, 29
Sabre, 29
Sage, 5
Salmon, 69, 72
Salt, 4
Sausage, 43, 53, 63-65
Seasoning\textsuperscript{P}, 4
Set\textsuperscript{P}, 123
SetEval\textsuperscript{V}, 126, 131
ShadowedCartesianPt, 141, 142
ShadowedManhattanPt, 139, 172
Shallot, 28, 36
Shape\textsuperscript{P}, 143
ShapeVisitor\textsuperscript{2}, 144
Shisk\textsuperscript{P}, 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

subAAC, 52, 53
SubAbC\textsuperscript{V}, 66
Subst\textsuperscript{P}, 133, 134
SubstFish\textsuperscript{V}, 82
Subst\textsuperscript{V}, 83, 87, 89, 94, 132-136, 163, 170
SubstInt\textsuperscript{V}, 82
subst\textsuperscript{Top}, 161
Sword, 29
Thyme, 5
Tomato, 16, 27, 28, 36, 59, 62
Top, 69, 78, 83, 85, 86, 91, 93, 162, 168
top\textsuperscript{AvC}, 50, 53
top\textsuperscript{AvC}\textsuperscript{V}, 66
Trans, 144
Tree\textsuperscript{P}, 100, 101, 107, 109, 112
TreeVisitor\textsuperscript{2}, 112
tSubst\textsuperscript{V}, 111
tTreeVisitor\textsuperscript{2}, 109
Tuna, 69, 72
Union, 149, 150
UnionHasPt\textsuperscript{V}, 151, 152, 157
UnionVisitor\textsuperscript{2}, 150
what\textsuperscript{Holder}, 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.

“\nThis 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