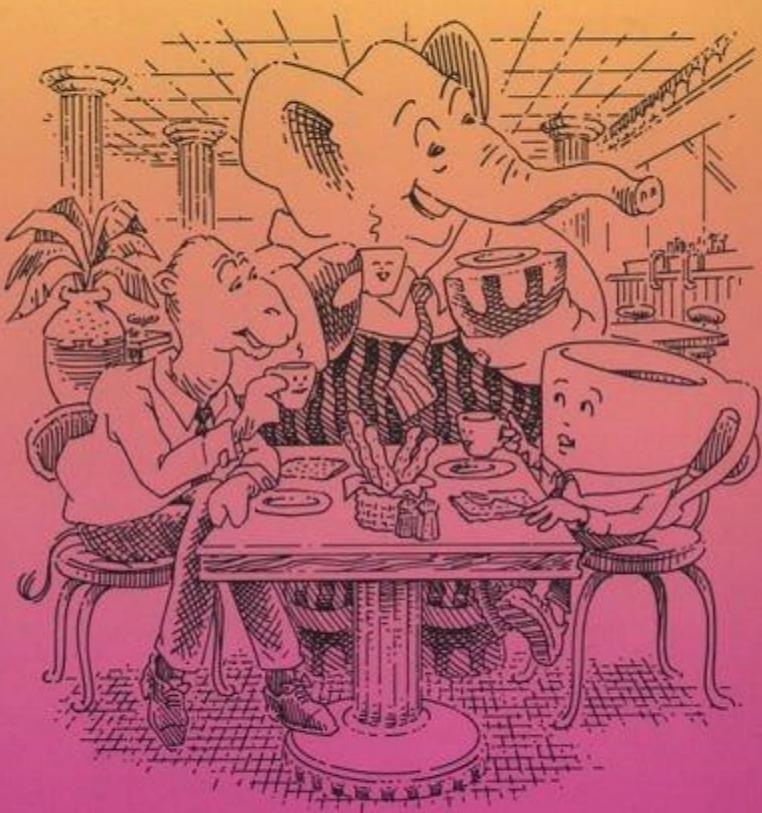


# A Little Java, A Few Patterns



Matthias Felleisen and Daniel P. Friedman

Foreword by Ralph E. Johnson

## **Contents**

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

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 a 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 **SETEX**. 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

## EXPERIMENTING WITH JAVA

Here are some hints on how to experiment with Java:<sup>1</sup>

1. Create a file that contains a complete hierarchy of classes.
2. To each class whose name does *not* end with a superscript  $\mathcal{D}$ ,  $\mathcal{V}$ ,  $\mathcal{I}$ , or  $\mathcal{M}$ , add a `toString` method according to these rules:

- a) if the class does not contain any fields, use

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

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

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

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

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

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

```
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:

```
class Main {  
    public static void main(String args[ ]) {  
        PointD y = new ManhattanPt(2,8);  
        System.out.println( y.distanceTo0() ); } }
```

---

<sup>1</sup>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.

If you wish to experiment with a sequence of expressions that modify *y*, as in chapter 10, *e.g.*,

```
y...- - - - - ;  
y...- - - - - ;  
y...- - - - -
```

replace *.....* with

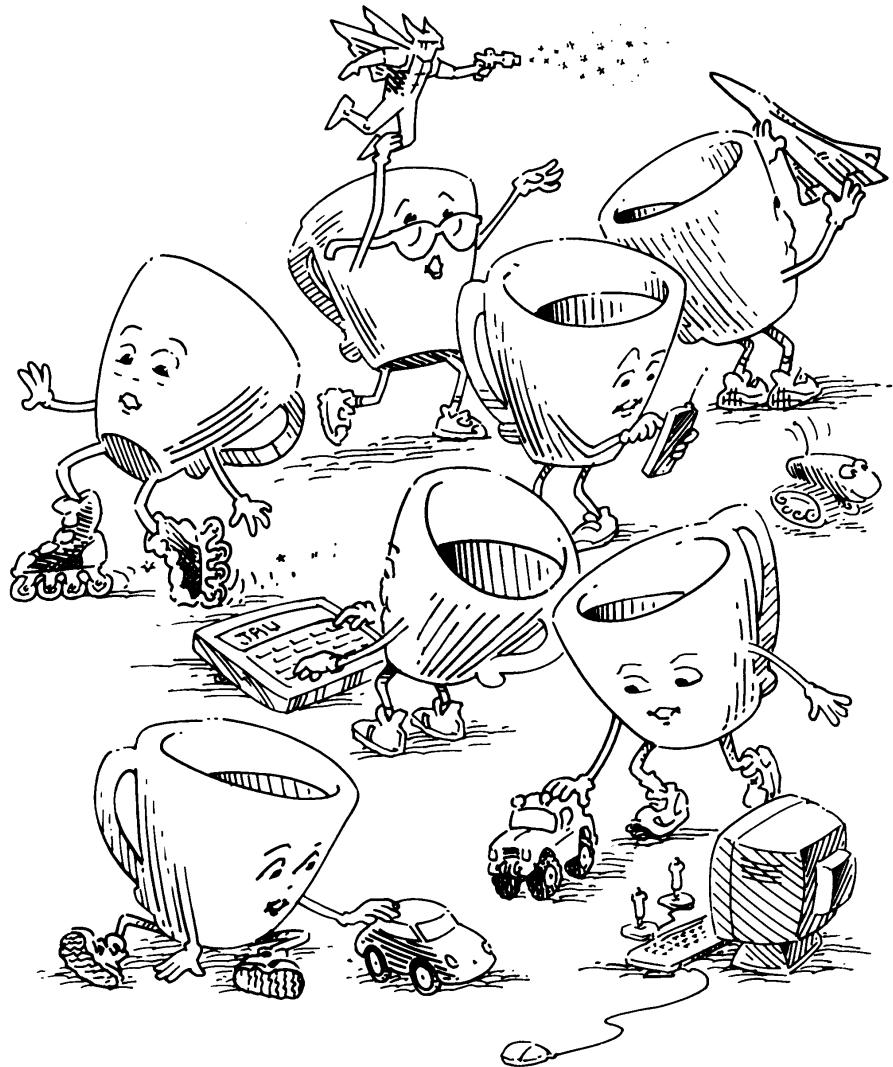
```
y...- - - - - + "\n" +  
y...- - - - - + "\n" +  
y...- - - - -
```

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

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

4. Finally, compile the file and interpret the class **Main**.

# I. Modern Toys



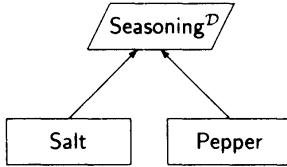
- 
- Is 5 an integer? <sup>1</sup> Yes, it is.
- 
- Is this a number: -23? <sup>2</sup> Yes, but we don't use negative integers.
- 
- Is this an integer: 5.32? <sup>3</sup> No, and we don't use this type of number.
- 
- What type of number is 5? <sup>4</sup> **int**.<sup>1</sup>
- <sup>1</sup> In Java, **int** stands for "integer."
- 
- Quick, think of another integer! <sup>5</sup> How about 19?
- 
- What type of value is true? <sup>6</sup> **boolean**.
- 
- What type of value is false? <sup>7</sup> **boolean**.
- 
- Can you think of another **boolean**? <sup>8</sup> No, that's all there is to **boolean**.
- 
- What is **int**? <sup>9</sup> A type.
- 
- What is **boolean**? <sup>10</sup> Another type.
- 
- What is a type? <sup>11</sup> A type is a name for a collection of values.
- 
- What is a type? <sup>12</sup> Sometimes we use it as if it were the collection.
- 
- Can we make new types? <sup>13</sup> We don't know how yet.
-

---

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

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

<sup>14</sup> Is this it?



<sup>D</sup> 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.

<sup>15</sup> Okay. But aren't all three classes introducing new types?

---

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

<sup>16</sup> Yes, it is, because `new Salt()` creates an instance of `Salt`, and every instance of `Salt` is also a `SeasoningD`.

---

And  
`new Pepper()`?

<sup>17</sup> 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**?

<sup>18</sup> Easy:  
**abstract class** introduces a datatype,  
**class** introduces a variant, and  
**extends** connects a variant to a datatype.

---

Is there any other `SeasoningD`?

<sup>19</sup> No, because only Salt and Pepper extend `SeasoningD`.<sup>1</sup>

<sup>1</sup> 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?

<sup>20</sup> No, but **boolean** is a type that also has just two values.

---

Let's define more `SeasoningD`s.

<sup>21</sup> We can have lots of `SeasoningD`s.

`class Thyme extends SeasoningD {}`

`class Sage extends SeasoningD {}`

---

And then there were four.

<sup>22</sup> Yes.

---

What is a Cartesian point?

<sup>23</sup> It is basically a pair of numbers.

---

What is a point in Manhattan?

<sup>24</sup> An intersection where two city streets meet.

---

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

<sup>25</sup> 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?<sup>1</sup>

`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; }  
}`

<sup>1</sup> This bar indicates the end of the constructor definition. It is used as an eye-catching separator. We recommend that you use  
// -----  
when you enter it in a file.

---

The underlined occurrences of `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:

`new CartesianPt(2,3)`,

we use the constructor in the definition of `CartesianPt`.

<sup>26</sup> How do we use these constructors?

<sup>27</sup> Obvious!

Correct. Is this a `ManhattanPt`:

`new ManhattanPt(2,3)?`

<sup>28</sup> So now we have created a `CartesianPt` whose *x* field is 2 and whose *y* field is 3. And because `CartesianPt` extends `PointD`, it is also a `PointD`.

Isn't all this obvious?

<sup>29</sup> Yes, and its *x* field is 2 and its *y* field is 3.

<sup>30</sup> 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.

<sup>31</sup> 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.

<sup>32</sup> Good. But what is `new PointD()`?

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

<sup>33</sup> That makes sense. Let's move on.

---

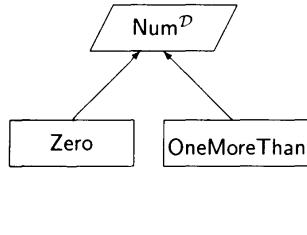
Do the following classes define another datatype with variants?

```
abstract class NumD {}
```

```
class Zero extends NumD {}
```

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

<sup>34</sup> Yes, they define a datatype and two variants.



Draw the picture, too.

---

Is this a Num<sup>D</sup>:  
`new Zero()`?

<sup>35</sup> Obviously, just like `new Salt()` is a Seasoning<sup>D</sup>.

---

Is this a Num<sup>D</sup>:  
`new OneMoreThan(  
 new Zero())`?

<sup>36</sup> Yes, because OneMoreThan constructs a Num<sup>D</sup> from a Num<sup>D</sup>, and every instance of OneMoreThan is also a Num<sup>D</sup>.

---

How does OneMoreThan do that?

<sup>37</sup> We give it `new Zero()`, which is a Num<sup>D</sup>, and it constructs a new Num<sup>D</sup>.

---

And what does it mean to construct this new instance?

<sup>38</sup> 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?

<sup>39</sup> No, its type says that it stands for a Num<sup>D</sup>, which, at the moment, may be either a Zero or a OneMoreThan.

---

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

<sup>40</sup> A  $\text{Num}^{\mathcal{D}}$ , because `OneMoreThan` constructs an instance from a  $\text{Num}^{\mathcal{D}}$  and we agreed that  
`new OneMoreThan(  
 new Zero())`

is a  $\text{Num}^{\mathcal{D}}$ .

---

What is  
`new OneMoreThan(  
 0)?`

<sup>41</sup> That is nonsense,<sup>1</sup> because 0 is not a  $\text{Num}^{\mathcal{D}}$ .

<sup>1</sup> We use the word “nonsense” to refer to expressions for which Java cannot determine a type.

---

Is `new Zero()` the same as 0?

<sup>42</sup> No, 0 is similar to, but not the same as,  
`new Zero()`.

---

Is  
`new OneMoreThan(  
 new Zero())`  
like  
1?

<sup>43</sup> 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()))))`

similar to?

<sup>44</sup> 4.

---

Are there more  $\text{Num}^{\mathcal{D}}$ s than `booleans`?

<sup>45</sup> Lots.

---

Are there more  $\text{Num}^{\mathcal{D}}$ s than `ints`?

<sup>46</sup> No.<sup>1</sup>

<sup>1</sup> This answer is only conceptually correct. Java limits the number of `ints` to approximately  $2^{32}$ .

---

---

What is the difference between `new Zero()` and `0`?

<sup>47</sup> Easy: `new Zero()` is an instance of `Zero` and, by implication, is a  $\text{Num}^D$ , 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.

<sup>48</sup> So are types just names for different collections with no common instances?

---

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

<sup>49</sup> 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  $\text{Num}^D$ , which is extended by `Zero`. Indeed, it is of any type that  $\text{Num}^D$  extends, too.

<sup>50</sup> And what is that?

---

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

<sup>51</sup> This must mean that everything is an `Object`.

---

Almost. We will soon see what that means.

<sup>52</sup> 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?

```
abstract class LayerD {}
```

<sup>53</sup> They define a new datatype and its two variants. The first variant contains a field of type Object.

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

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

What is

```
new Base(  
    new Zero())?
```

<sup>54</sup> It looks like an instance of Base, which means it is also a Layer<sup>D</sup> and an Object.

And what is

```
new Base(  
    new Salt())?
```

<sup>55</sup> It also looks like an instance of Base. But how come both

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

and

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

are instances of the same variant?

---

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

<sup>56</sup> Hence, we can use both

```
new Zero()
```

and

```
new Salt()
```

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

---

Is anything else an Object?

<sup>57</sup> We said that only things created with **new** are Objects.<sup>1</sup>

<sup>1</sup> Arrays and strings are objects, too. We don't discuss them.

---

Correct. Is this a Layer<sup>D</sup>:  
**new Base(**  
    5**)?**

<sup>58</sup> 5 is not created with **new**, so this must be nonsense.

Is this a Layer<sup>D</sup>:  
**new Base(**  
    false**)?**

<sup>59</sup> false is not created with **new**, so this must be nonsense, too.

---

Correct again! How about this Layer<sup>D</sup>:  
**new Base(**  
    **new Integer(5))?**

<sup>60</sup> This must mean that Integer creates an object from an int.

Guess how we create a Layer<sup>D</sup> from false?

<sup>61</sup> Easy now:

**new Base(**  
    **new Boolean(false)).**

---

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

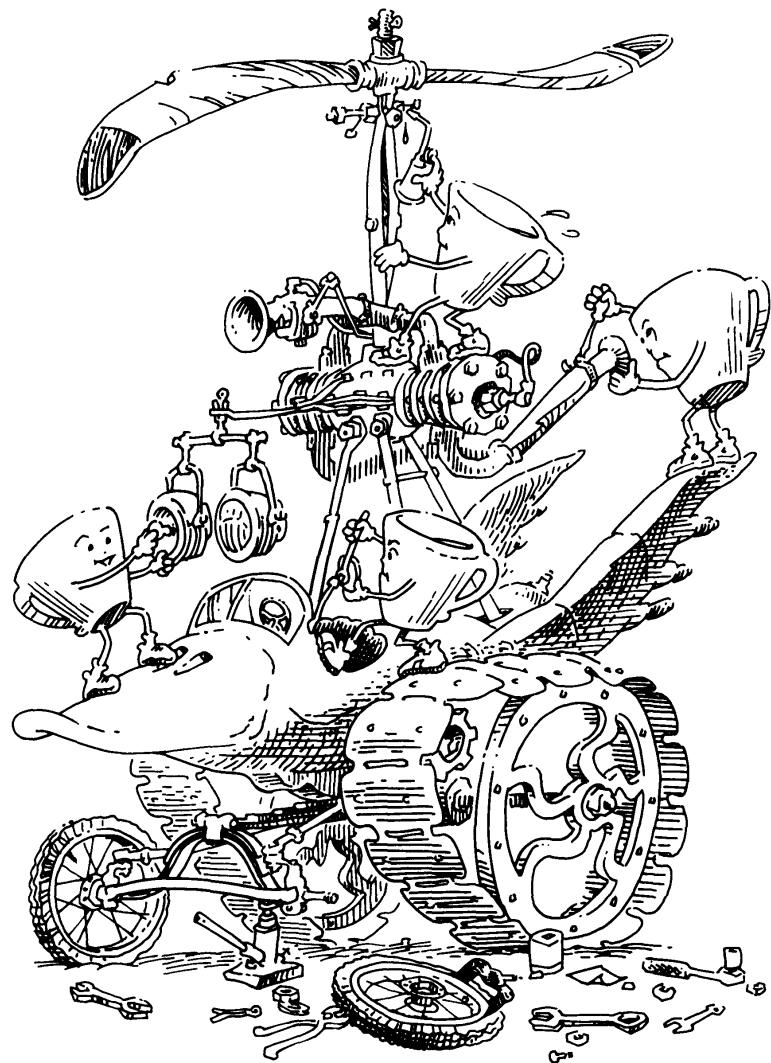
<sup>62</sup> Too much coffee does that.

Ready for more?

<sup>63</sup> Can't wait.

---

2.  
Methods to  
Our Madness



---

Remember points?

<sup>1</sup> Sure, we just talked about them. But what are these labeled ovals about?

```
abstract class PointD {  
    ——————  
    Point  
    }—————
```

```
class CartesianPt extends PointD {  
    int x;  
    int y;  
    CartesianPt(int _x,int _y) {  
        x = _x;  
        y = _y; }  
    ——————  
    }—————  
    CartesianPt
```

```
class ManhattanPt extends PointD {  
    int x;  
    int y;  
    ManhattanPt(int _x,int _y) {  
        x = _x;  
        y = _y; }  
    ——————  
    }—————  
    ManhattanPt
```

---

We will find out soon. Did you notice the big <sup>2</sup> It must be for drawing the picture of the white space on the right? classes.

---

How far is  
**new ManhattanPt(3,4)**  
from the Empire State Building?

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

---

And how far is

<sup>4</sup> 5, which is  $\sqrt{3^2 + 4^2}$ .

**new CartesianPt(3,4)**  
from the origin?

---

Write the methods *distanceToO* using `{`, `}`, `(`, `)`, `.`, `..`, `return`, `int`, `+`, `[ $\sqrt{\cdot}$ ]`, and `.2`, which determine how far a point is from the origin.<sup>5</sup> Of course, you can't write these methods, yet. Okay, you deserve something sweet for enduring this last question.

---

What do the methods produce?

<sup>6</sup> ints, which represent the distances to the origin.

---

Here they are.

`| abstract int distanceToO();`

Point

`| int distanceToO() {`  
`return [ $\sqrt{x^2 + y^2}$ ]1; }`

CartesianPt

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

ManhattanPt

<sup>7</sup> They correspond to the unexplained labels in the definition of the datatype and its variants.

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

<sup>1</sup> 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 <sup>8</sup> That's simple enough. these methods into Point<sup>D</sup>, CartesianPt, and ManhattanPt.

---

How many times have we defined the method <sup>9</sup> 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**?

---

<sup>10</sup> Yes, they always do.

An **abstract** method in an **abstract** class      <sup>11</sup> Okay.  
introduces an obligation, which says that all  
concrete classes that extend this abstract  
class<sup>1</sup> must contain a matching method  
definition.

<sup>1</sup> 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()?

---

<sup>12</sup> 7.

How do we arrive at that value?

<sup>13</sup> 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()?

---

<sup>14</sup> 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?

---

<sup>15</sup> The largest **int** that does not exceed the  
square root of  $x$ .

---

Time for a short break?

---

<sup>16</sup> An apple a day keeps the dentist away. A  
cup of coffee does not.

---

Here is another datatype with its variants. <sup>17</sup> It is like  $\text{Num}^D$  but has more 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; }  
    ——————  
    Onion  
}
```

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

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

---

Did you notice the big space on the right? <sup>18</sup> Yes, isn't it for drawing the picture of the classes?

---

---

Construct a Shish<sup>D</sup>.

<sup>19</sup> How about  
new Skewer()?

---

Yes, every Skewer is also a Shish<sup>D</sup>. How about another one?

<sup>20</sup> Here's one:  
new Onion(  
new Skewer()).

---

And a third?

<sup>21</sup> Here's one more:  
new Onion(  
new Lamb(  
new Onion(  
new Skewer()))).

---

Are there only Onions on this Shish<sup>D</sup>:  
new Skewer()?

<sup>22</sup> true, because there is neither Lamb nor Tomato on new Skewer ()�.

---

Are there only Onions on this Shish<sup>D</sup>:  
new Onion(  
new Skewer())?

<sup>23</sup> true.

---

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

<sup>24</sup> false.

---

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

<sup>25</sup> true.

---

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

<sup>26</sup> false.

---

---

Write the methods *onlyOnions*<sup>1</sup> using {}, (), , ;, true, false, return, and boolean.

<sup>27</sup> Of course, you can't write these methods, yet. Okay, you deserve a lollipop for enduring this kind of question again.

<sup>1</sup> A better name for these methods would be nothingButOnions.

---

And what do they produce?

<sup>28</sup> **booleans.**

---

Here are the methods.

<sup>29</sup> Yes. We said above that the labeled ovals in the center of the blank lines in the above class definitions tell us where to put the boxes with the corresponding labels.

```
abstract boolean onlyOnions();
```

Shish

```
boolean onlyOnions() {
```

return true; }

Skewer

```
boolean onlyOnions() {
```

return s.onlyOnions(); }

Onion

```
boolean onlyOnions() {
```

return false; }

Lamb

```
boolean onlyOnions() {
```

return false; }

Tomato

Did you notice the labels in the boxes?

---

Good. How many methods have we defined?

<sup>30</sup> Five, but the first one is **abstract**; the others are concrete.

---

---

Do **abstract** methods belong to the **abstract class**?

<sup>31</sup> Yes, we said so.

---

Does each variant of `ShishD` contain a method called `onlyOnions`?

<sup>32</sup> Yes, because `ShishD` contains an **abstract** method called `onlyOnions` that obligates each variant to define a matching, concrete method.

---

Is this always the case?

<sup>33</sup> Always.

---

What do these concrete methods consume?

<sup>34</sup> Nothing, just as the **abstract** method says.

---

What do these concrete methods produce?

<sup>35</sup> `booleans`, just as the **abstract** method says.

---

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

<sup>36</sup> `true`.

---

And how do we determine the value of  
`new Onion(`  
    `new Onion(`  
        `new Skewer()`)  
. `onlyOnions()`?

<sup>37</sup> 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()`?

<sup>38</sup> 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**?

<sup>39</sup> *s.onlyOnions()*.

---

What is the field *s* of the object

**new Onion(**  
    **new Onion(**  
        **new Skewer())?)**

<sup>40</sup> It is

**new Onion(**  
    **new Skewer()),**  
isn't it?

---

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

<sup>41</sup> No, it has type **Shish<sup>D</sup>**, and it can stand for  
any variant of **Shish<sup>D</sup>**: **Skewer**, **Onion**, **Lamb**,  
or **Tomato**.

---

Then what is *s.onlyOnions()*?

<sup>42</sup> It should be  
**new Onion(**  
    **new Skewer()**)  
    *.onlyOnions()*,  
right?

---

Why do we need to know the meaning of

**new Onion(**  
    **new Skewer()**)  
.onlyOnions()?

<sup>43</sup> 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()?

<sup>44</sup> Let's see.

---

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

**new Onion(**  
    **new Skewer()**)  
**.onlyOnions()**?

<sup>45</sup> 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*?

<sup>46</sup> *s.onlyOnions()*.

---

What is the field *s* of the object

**new Onion(**  
    **new Skewer()**)?

<sup>47</sup> **new Skewer()**.

---

Then what is *s.onlyOnions()*?

<sup>48</sup> It is  
**new Skewer()**  
**.onlyOnions()**,  
just as we would have expected.

---

Why do we need to know the meaning of

**new Skewer()**  
**.onlyOnions()**?

<sup>49</sup> Because the answer for

**new Skewer()**  
**.onlyOnions()**

is also the answer for  
**new Onion(**  
    **new Skewer()**)  
**.onlyOnions()**,

which in turn is the answer for  
**new Onion(**  
    **new Onion(**  
        **new Skewer()**)  
**.onlyOnions()**).

---

How do we determine the answer for

**new Skewer()**  
**.onlyOnions()**?

<sup>50</sup> We need to determine one more time which version of *onlyOnions* we must use.

---

Is

<sup>51</sup> Obviously.

**new Skewer()**  
a  
Skewer?

---

Then what is the answer?

<sup>52</sup> true.

---

Why?

<sup>53</sup> Because true is what the *onlyOnions* method in Skewer always returns.

---

Are we done?

<sup>54</sup> Yes! The answer for  
**new Onion(**  
**new Onion(**  
**new Skewer()))**  
.onlyOnions()

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

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

which is  
true.

---

What is the value of

**new Onion(**  
**new Lamb(**  
**new Skewer()))**  
.onlyOnions()?

---

<sup>55</sup> false, isn't it?

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

**new Onion(**  
**new Lamb(**  
**new Skewer()))**  
.onlyOnions()?

---

<sup>56</sup> 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**?

<sup>57</sup> *s.onlyOnions()*.

---

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

<sup>58</sup> It is the object built from  
**new Lamb(**  
    **new Skewer()**).

---

Then what is *s.onlyOnions()*?

<sup>59</sup> It is  
**new Lamb(**  
    **new Skewer()**)  
.onlyOnions(),  
of course.

---

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

<sup>60</sup> 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()?

<sup>61</sup> We determine which version of *onlyOnions* to use.

---

And?

<sup>62</sup> We use the one that belongs to **Lamb**.

---

And now what is the answer?

<sup>63</sup> **false**, because **false** follows the word **return** in the corresponding method definition in **Lamb**.

---

Are we done?

<sup>64</sup> Yes! The answer for  
`new Onion(  
 new Lamb(  
 new Skewer())))  
.onlyOnions()`

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

---

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

<sup>65</sup> Here are our words:  
“The methods determine for a `ShishD`  
whether its contents are edible by an onion  
lover.”

---

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

<sup>66</sup> Here are our words again:  
“For each layer of the `ShishD`, except for  
Onion, the corresponding method knows  
whether it is good or bad. The method for  
Onion needs to determine whether the  
remaining layers are only Onions sitting on  
a `Skewer`.”

---

Is

`new Tomato(  
 new Skewer())`

a `ShishD`?

<sup>67</sup> Yes.

Is

`new Onion(  
 new Tomato(  
 new Skewer())))`

a `ShishD`?

<sup>68</sup> The object

`new Tomato(  
 new Skewer())`

is an instance of `ShishD`, so we can also wrap  
an Onion around it.

---

And how about another Tomato?

<sup>69</sup> Sure.

---

---

Is

```
new Tomato(  
    new Onion(  
        new Tomato(  
            new Skewer()))))
```

a vegetarian shish kebab?

<sup>70</sup> Of course, there is no Lamb on it.

---

And

```
new Onion(  
    new Onion(  
        new Onion(  
            new Skewer()))))?
```

<sup>71</sup> 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.

<sup>72</sup> 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? <sup>73</sup> Five: one **abstract**, the others concrete.

---

Do **abstract** methods belong to the **abstract class**? <sup>74</sup> Yes, they always do.

---

Does each variant of  $\text{Shish}^D$  contain a method called *isVegetarian*? <sup>75</sup> Yes, because  $\text{Shish}^D$  contains an **abstract** method called *isVegetarian*.

---

Is this always the case? <sup>76</sup> Always.

---

What do these concrete methods consume? <sup>77</sup> Nothing, just as the **abstract** method says.

---

What do these concrete methods produce? <sup>78</sup> **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  $\text{Shish}^D$ . Here is the datatype.

```
abstract class ShishD {  
    abstract boolean onlyOnions();  
    abstract boolean isVegetarian();  
}
```

<sup>79</sup> There are two methods per variant.

```
class Skewer extends ShishD {  
    boolean onlyOnions() {  
        return true; }  
    boolean isVegetarian() {  
        return true; }  
}
```

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

---

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

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

---

```
boolean onlyOnions() {  
    return false; }  
boolean isVegetarian() {  
    return false; }  
}
```

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

---

```
boolean onlyOnions() {  
    return false; }  
boolean isVegetarian() {  
    return s.isVegetarian(); }  
}
```

---

What do the following define?

```
abstract class KebabD {  
    —————— Kebab ——————  
}
```

<sup>80</sup> They define a datatype and four variants  
that are similar in shape to Shish<sup>D</sup>.

```
class Holder extends KebabD {  
    Object o;  
    Holder(Object _o) {  
        o = _o; }  
    —————— Holder ——————  
}
```

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

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

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

Don't forget the picture.

---

---

What is different about them?

<sup>81</sup> They are placed onto different Holders.

---

Here are some holders.

```
abstract class RodD {}
```

<sup>82</sup> 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?

```
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?

<sup>83</sup> We could move all of the food to various forms of plates.

```
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

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

<sup>84</sup> It's a Kebab<sup>D</sup>.

---

```

Is
new Shallot(
  new Radish(
    new Holder(
      new Dagger()))))
a vegetarian KebabD?
```

<sup>85</sup> Sure it is. It only contains radishes and shallots.

Is  
**new Shallot(**  
**new Radish(**  
**new Holder(**  
**new Gold()))))**  
a Kebab<sup>D?</sup>

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

```
Is  
  new Shallot()  
  new Radish()  
  new Holder()  
  new Gold()))))  
a vegetarian kebab?
```

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

Let's define the methods (*i.e.*, the function)

<sup>8</sup> If you can, you may rest now.

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

Write the abstract method *isVeggie*.

<sup>89</sup> That's possible now.

**abstract boolean** *isVeggie()*

## Kebab

Of course, *isVeggie* belongs to **Kebab<sup>D</sup>** and *isVegetarian* to **Shish<sup>D</sup>**.

---

The concrete methods are similar to those called *isVegetarian*. Here are two more; define the remaining two.

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

Holder

<sup>90</sup> Except for the names of the methods and fields, the definitions are the same as they were for Shish<sup>D</sup>.

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

Shrimp

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

Shallot

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

Radish

---

What is the value of

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

<sup>91</sup> true.

---

What is

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

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

---

What is the value of

```
new Shallot(  
    new Radish(  
        new Holder(  
            new Gold()))))  
.isVeggie()?
```

<sup>93</sup> It is true, too.

---

And what is

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

<sup>94</sup> It is also a Kebab<sup>D</sup>, because any kind of Holder will do.

---

What type of value is

```
new Shallot(  
    new Radish(  
        new Holder(  
            new Integer(52))))  
.isVeggie()?
```

<sup>95</sup> **boolean.**

What type of value is

```
new Shallot(  
    new Radish(  
        new Holder(  
            new OneMoreThan(  
                new Zero()))))  
.isVeggie()?
```

<sup>96</sup> **boolean.**

What type of value is

```
new Shallot(  
    new Radish(  
        new Holder(  
            new Boolean(false))))  
.isVeggie()?
```

<sup>97</sup> **boolean.**

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

<sup>98</sup> 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())))?
```

<sup>99</sup> All the food is on a Dagger.

What is the holder of

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

<sup>100</sup> All the food is now on a Gold plate.

---

---

What is the holder of  
new Shallot(  
new Radish(  
new Holder(  
new Integer(52))))?

---

<sup>101</sup> All the food is on an **Integer**.

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

---

<sup>102</sup> The dagger.

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

---

<sup>103</sup> The gold plate.

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

---

<sup>104</sup> An **Integer**, whose underlying **int** is 52.

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

---

<sup>105</sup> They produce rods, plates, and integers. And it looks like they can produce a lot more.

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

---

<sup>106</sup> They always produce an **Object**, which is also the type of the field of **Holder**.

Here is the abstract method *whatHolder*.

**abstract Object** *whatHolder()*

Kebab

---

<sup>107</sup> If we add this method to **Kebab**<sup>D</sup>, then we must add a method definition to each of the four variants.

---

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

---

<sup>108</sup>   **new Integer(52).**

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

---

<sup>109</sup>   **new Sword().**

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

---

<sup>110</sup>   It is *b*.

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

---

<sup>111</sup>   With these kinds of hints, it's easy.

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

Holder

---

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

---

<sup>112</sup>   **new Integer(52).**

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

---

<sup>113</sup>   **new Integer(52).**

---

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

<sup>114</sup> **new Integer(52).**

Does that mean that the value of  
**new Radish(**  
  **new Shallot(**  
    **new Shrimp(**  
      **new Holder(**  
        **new Integer(52))))**  
  **.whatHolder()**)

<sup>115</sup> Yes, all four have the same answer:  
**new Integer(52).**

is the same as  
**new Shallot(**  
  **new Shrimp(**  
    **new Holder(**  
      **new Integer(52))))**  
  **.whatHolder()**,

which is the same as  
**new Shrimp(**  
  **new Holder(**  
    **new Integer(52))**  
  **.whatHolder()**,

which is the same as  
**new Holder(**  
  **new Integer(52))**  
  **.whatHolder()**?

---

Here is the method for Shallot.

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

Shallot

<sup>116</sup> They are all the same.

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

Shrimp

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

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

Radish

---

Here is the datatype and one of its variants.

<sup>117</sup> There are only three left.

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

```
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(); }  
}
```

Collect the remaining variants.

---

Are there any other  $\text{Kebab}^D$  foods besides Shallot, Shrimp, and Radish?

<sup>118</sup> No, these are the only kinds of foods on a  $\text{Kebab}^D$ .

---

---

Can we add more foods?

<sup>119</sup> Sure. We did something like that when we added Thyme and Sage to Seasoning<sup>D</sup>.

---

Let's define another Kebab<sup>D</sup>.

```
class Pepper extends KebabD {  
    KebabD k;  
    Pepper(KebabD _k) {  
        k = _k; }  
  
    boolean isVeggie() {  
        return k.isVeggie(); }  
    Object whatHolder() {  
        return k.whatHolder(); }  
}
```

Why does it include *isVeggie* and *whatHolder* methods?

<sup>120</sup> A concrete class that extends Kebab<sup>D</sup> must define these two methods. That's what the **abstract** specifications say. We can define as many Kebab<sup>D</sup>s as we wish.

```
class Zucchini extends KebabD {  
    KebabD k;  
    Zucchini(KebabD _k) {  
        k = _k; }  
  
    boolean isVeggie() {  
        return k.isVeggie(); }  
    Object whatHolder() {  
        return k.whatHolder(); }  
}
```

Is it obvious how the new methods work?

<sup>121</sup> Totally. In both cases *isVeggie* just checks the rest of the Kebab<sup>D</sup>, because green peppers and zucchini are vegetables. Similarly, *whatHolder* returns whatever holder belongs to the rest of the Kebab<sup>D</sup>.

---

And then there were six.

<sup>122</sup> Yes, now Kebab<sup>D</sup> has six variants.

---

Which of these points is closer to the origin:

**new ManhattanPt(3,4)**  
and  
**new ManhattanPt(1,5)?**

<sup>123</sup> 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 CartesianPt(3,4)**  
or  
**new CartesianPt(12,5)?**

<sup>124</sup> 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.

```
class CartesianPt extends PointD {  
    int x;  
    int y;  
    CartesianPt(int _x,int _y) {  
        x = _x;  
        y = _y; }  
  
    int distanceToO() {  
        return ⌊√(x2 + y2)⌋; }  
    boolean closerToO(CartesianPt p) {  
        return  
            distanceToO() ≤ p.distanceToO(); }  
}
```

Add the corresponding method to `ManhattanPt`.

<sup>125</sup> The definitions are nearly identical. The method for `ManhattanPt` consumes a `ManhattanPt` and determines which of those two points is closer to the origin.

```
class ManhattanPt extends PointD {  
    int x;  
    int y;  
    ManhattanPt(int _x,int _y) {  
        x = _x;  
        y = _y; }  
  
    int distanceToO() {  
        return x + y; }  
    boolean closerToO(ManhattanPt p) {  
        return  
            distanceToO() ≤1 p.distanceToO(); }  
}
```

<sup>1</sup> This is the two character symbol `≤`.

---

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

<sup>126</sup> `false`.

---

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

<sup>127</sup> `true`, obviously.

---

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

<sup>128</sup> `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))`?

<sup>129</sup> That's nonsense.

---

Why?

<sup>130</sup> The method *closerToO* can only consume *CartesianPts*, not *ManhattanPts*.

---

How can we fix that?

<sup>131</sup> We could replace  
(*CartesianPt p*)  
by  
(*Point<sup>D</sup> p*)  
in the definition of *closerToO* for *CartesianPt*.

---

If we do that, can we still determine the value of  
*p.distanceToO()*?

<sup>132</sup> Yes, because the definition of *Point<sup>D</sup>* obligates every variant to provide a method named *distanceToO*.

---

Why does it help to replace (*CartesianPt p*) by (*Point<sup>D</sup> p*)?

<sup>133</sup> Every *CartesianPt* is a *Point<sup>D</sup>*, and every *ManhattanPt* is a *Point<sup>D</sup>*, too.

---

Here is the improved *CartesianPt*.

```
class CartesianPt extends PointD {  
    int x;  
    int y;  
    CartesianPt(int _x,int _y) {  
        x = _x;  
        y = _y; }  
  
    int distanceToO() {  
        return [sqrt(x2 + y2]; }  
    boolean closerToO(PointD p) {  
        return  
            distanceToO() ≤ p.distanceToO(); }  
}
```

<sup>134</sup> Easy.

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

Improve the definition of *ManhattanPt*.

---

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

<sup>135</sup> Yes, they are identical.

---

---

Correct, and therefore we can add a copy to the abstract class `PointD` and delete the definitions from the variants.

```
abstract class PointD {  
    boolean closerToO1(PointD p) {  
        return  
            distanceToO() ≤ p.distanceToO(); }  
    abstract int distanceToO();  
}
```

<sup>136</sup> Looks correct.

<sup>1</sup> The method `closerToO` is a *template* and the method `distanceToO` is a *hook* in the *template method pattern instance* [4].

---

What else do the two point variants have in common?

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

```
abstract class PointD {  
    int x;  
    int y;  
    PointD(int _x,int _y) {  
        x = _x;  
        y = _y; }  
  
    boolean closerToO(PointD p) {  
        return  
            distanceToO() ≤ p.distanceToO(); }  
    abstract int distanceToO();  
}
```

<sup>137</sup> Their fields. Shouldn't we lift them, too?  
<sup>138</sup> This not only lifts `x` and `y`, it also introduces a new constructor.

---

Absolutely. And we need to change how a `CartesianPt` is built. Define `ManhattanPt`.

```
class CartesianPt extends PointD {  
    CartesianPt(int _x,int _y) {  
        super(_x,_y); }  
  
    int distanceToO() {  
        return ⌊√(x2 + y2)⌋; }  
}
```

<sup>139</sup> Mimicking this change is easy. But what does `super(_x,_y)` mean?

```
class ManhattanPt extends PointD {  
    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 `PointD` with the appropriate fields, and the respective constructor guarantees that the point becomes a `CartesianPt` or a `ManhattanPt`.

---

<sup>140</sup> That's simple.

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

<sup>141</sup> Yes, and those things that distinguish the two variants from each other reside in the corresponding variants.

---

Is this a long chapter?

<sup>142</sup> Yes, let's have a short snack break.

---

3.  
What's New?



---

Do you like to eat pizza?

```
abstract class PizzaD {  
    ——————  
    }  
    ——————  
    Pizza  
    ——————
```

```
class Crust extends PizzaD {  
    ——————  
    }  
    ——————  
    Crust  
    ——————
```

```
class Cheese extends PizzaD {  
    PizzaD p;  
    Cheese(PizzaD _p) {  
        p = _p; }  
    ——————  
    }  
    ——————  
    Cheese  
    ——————
```

```
class Olive extends PizzaD {  
    PizzaD p;  
    Olive(PizzaD _p) {  
        p = _p; }  
    ——————  
    }  
    ——————  
    Olive  
    ——————
```

```
class Anchovy extends PizzaD {  
    PizzaD p;  
    Anchovy(PizzaD _p) {  
        p = _p; }  
    ——————  
    }  
    ——————  
    Anchovy  
    ——————
```

<sup>1</sup> Looks like good toppings. Let's add Sausage.

```
class Sausage extends PizzaD {  
    PizzaD p;  
    Sausage(PizzaD _p) {  
        p = _p; }  
    ——————  
    }  
    ——————  
    Sausage  
    ——————
```

---

Here is our favorite pizza:

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

<sup>2</sup> This looks too salty.

---

How about removing the anchovies?

<sup>3</sup> That would make it less salty.

---

Let's remove them. What is the value of

```
new Anchovy(  
  new Olive(  
    new Anchovy(  
      new Anchovy(  
        new Cheese(  
          new Crust())))))  
.remA1()?
```

<sup>4</sup> It should be a cheese and olive pizza, like this:

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

---

What is the value of

```
new Sausage(  
  new Olive(  
    new Anchovy(  
      new Sausage(  
        new Cheese(  
          new Crust())))))  
.remA()?
```

<sup>5</sup> It should be a cheese, sausage, and olive pizza, like this:

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

---

Does *remA* belong to the datatype  $\text{Pizza}^D$  and its variants?

<sup>6</sup> Yes, and it produces them, too.

---

Define the methods that belong to the five variants. Here is a start.

**abstract** Pizza<sup>D</sup> remA();

Pizza

Pizza<sup>D</sup> remA() {  
    **return new** Crust(); }

Crust

<sup>7</sup> We didn't expect you to know this one.

---

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

Pizza<sup>D</sup> remA() {  
    **return new** Cheese(p.remA()); }

Cheese

Pizza<sup>D</sup> remA() {  
    **return new** Olive(p.remA()); }

Olive

Pizza<sup>D</sup> remA() {  
    **return new** Sausage(p.remA()); }

Sausage

<sup>8</sup> The Olive and Sausage methods are similar to the Cheese method.

---

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

<sup>9</sup> The cheese, the olives, and the sausages on the pizzas must be put back on top of the pizza that p.remA() produces.

---

The methods remA must produce a Pizza<sup>D</sup>, so let's use **new** Crust(), the simplest Pizza<sup>D</sup>, for the method in Anchovy.

Pizza<sup>D</sup> remA() {  
    **return new** Crust(); }

Anchovy

<sup>10</sup> Yes, and now the methods of remA produce pizzas without any anchovies on them.

---

Let's try it out on a small pizza:

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

<sup>11</sup> That's easy. The object is an Anchovy. So the answer is **new Crust()**.

---

Is  
    **new Crust()**  
like  
    **new Anchovy(**  
        **new Crust()**)  
without anchovy?

<sup>12</sup> Absolutely, but what if we had more anchovies?

No problem. Here is an example:

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

<sup>13</sup> 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()?
```

<sup>14</sup> 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())))?
```

<sup>15</sup> 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())?

<sup>16</sup> 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())?

<sup>17</sup> 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())?

<sup>18</sup> Yes, we know that it produces **new Crust()**.

---

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

---

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

<sup>19</sup> No, we still have to add cheese and an olive.

<sup>20</sup> Yes, we must first add cheese, producing

**new Cheese(**  
  **new Crust())**

and then we add the olive.

---

So what is the final answer?

<sup>21</sup> It is

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

---

---

Let's try one more example:

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

What kind of pizza should this make?

---

Check it out!

<sup>22</sup> It should be a double-cheese pizza.

<sup>23</sup> The object is an instance of `Cheese` so the value is

```
new Cheese(p.remA())  
where p stands for  
new Anchovy(  
    new Cheese(  
        new Crust()))).
```

---

Doesn't that mean that the result is

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

<sup>24</sup> Yes, it puts the cheese on whatever we get for

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

---

What about

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

<sup>25</sup> Now the object is an anchovy.

---

And the answer is

```
new Crust()?
```

<sup>26</sup> Yes, but we need to add cheese on top.

---

Does that mean the final answer is

```
new Cheese(  
    new Crust())?)
```

<sup>27</sup> Yes, though it's not the answer we want.

---

What do we want?

<sup>28</sup> A double-cheese pizza like

```
new Cheese(  
    new Cheese(  
        new Crust()))),
```

because that's what it means to remove anchovies and nothing else.

---

Which *remA* method do we need to change to get the cheese back?

<sup>29</sup> The one in Anchovy.

```
PizzaD remA() {  
    return p.remA(); }
```

Anchovy

---

Does this *remA* still belong to *Pizza<sup>D</sup>*?

<sup>30</sup> 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. <sup>31</sup> Yes, that would hide their taste, too.

---

What kind of pizza is

```
new Olive(  
    new Anchovy(  
        new Cheese(  
            new Anchovy(  
                new Cheese(  
                    new Crust())))))  
.topAwC1()?
```

<sup>32</sup> Easy, it adds cheese on top of each anchovy:

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

<sup>1</sup> A better name for these methods would be *topAnchovywithCheese*.

---

Did you notice the underlines?

<sup>33</sup> Yes, they show where we added cheese.

---

---

And what is

```
new Olive(  
    new Cheese(  
        new Sausage(  
            new Crust()))))  
.topAwC()?
```

<sup>34</sup> 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.

```
| abstract PizzaD topAwC();
```

Pizza

```
| PizzaD topAwC() {  
    return new Crust(); }
```

Crust

<sup>35</sup> We expect you to know some of the answers.

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

Cheese

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

Olive

```
PizzaD topAwC() {  
    return new Sausage(p.topAwC()); }
```

Sausage

---

Take a look at this method.

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

Anchovy

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

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

Anchovy

ers of cheese are in the result

<sup>37</sup> One, because *remA* removes all the anchovies, so that *topAwC* doesn't add any cheese.

```
ovy(  
ese(  
ichovy(  
rust()))))
```

?

urrences of cheese are in the

<sup>38</sup> Three, because *topAwC* first adds cheese for each anchovy. Then *remA* removes all the anchovies:

```
ovy(  
ese(  
chovy(  
.rust()))))
```

new Olive()  
new Cheese()  
new Cheese()  
new Cheese()  
new Fruit( )) ) )

How many layers  
of  
(new Olive()  
    new Anch()  
        new Che()  
            new An()  
                new C()  
                    .remA()()  
                        .topAwC()()

How many occ  
result of  
(new Olive  
new Anch  
new Che  
new An  
new C  
n... AmCVA

Perhaps we should replace every anchovy with cheese.

<sup>39</sup> We just did something like that.

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

<sup>40</sup> 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!

---

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*.<sup>1</sup> Here is the abstract method.

```
abstract PizzaD subAbC();
```

Pizza

<sup>42</sup> Here is a skeleton.

```
PizzaD subAbC() {  
    return new Crust(); }
```

Crust

```
PizzaD subAbC() {  
    return new Cheese(p.subAbC()); }
```

Cheese

```
PizzaD subAbC() {  
    return new Olive(p.subAbC()); }
```

Olive

```
PizzaD subAbC() {  
    return _____; }
```

Anchovy

```
PizzaD subAbC() {  
    return new Sausage(p.subAbC()); }
```

Sausage

<sup>1</sup> A better name for these methods would be `substituteAnchovybyCheese`.

---

Does this skeleton look familiar?

<sup>43</sup> Yes, this skeleton looks just like those of *topAwC* and *remA*.

---

Define the method that belongs in Anchovy.

<sup>44</sup> Here it is.

```
PizzaD subAbC() {  
    return new Cheese(p.subAbC()); }
```

Anchovy

Collection time.<sup>1</sup>

<sup>45</sup> The classes are getting larger.

```
abstract class PizzaD {  
    abstract PizzaD remA();  
    abstract PizzaD topAwC();  
    abstract PizzaD subAbC();  
}
```

```
class Crust extends PizzaD {  
    PizzaD remA() {  
        return new Crust(); }  
    PizzaD topAwC() {  
        return new Crust(); }  
    PizzaD subAbC() {  
        return new Crust(); }  
}
```

```
class Cheese extends PizzaD {  
    PizzaD p;  
    Cheese(PizzaD -p) {  
        p = -p; }  
  
    PizzaD remA() {  
        return new Cheese(p.remA()); }  
    PizzaD topAwC() {  
        return new Cheese(p.topAwC()); }  
    PizzaD subAbC() {  
        return new Cheese(p.subAbC()); }  
}
```

```
class Olive extends PizzaD {  
    PizzaD p;  
    Olive(PizzaD -p) {  
        p = -p; }  
}
```

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

```
class Anchovy extends PizzaD {  
    PizzaD p;  
    Anchovy(PizzaD -p) {  
        p = -p; }  
  
    PizzaD remA() {  
        return p.remA(); }  
    PizzaD topAwC() {  
        return  
            new Cheese(  
                new Anchovy(p.topAwC())); }  
    PizzaD subAbC() {  
        return new Cheese(p.subAbC()); }  
}
```

```
class Sausage extends PizzaD {  
    PizzaD p;  
    Sausage(PizzaD -p) {  
        p = -p; }  
  
    PizzaD remA() {  
        return new Sausage(p.remA()); }  
    PizzaD topAwC() {  
        return new Sausage(p.topAwC()); }  
    PizzaD subAbC() {  
        return new Sausage(p.subAbC()); }  
}
```

<sup>1</sup> This is similar to the *interpreter* and *composite* patterns [4].

---

Let's add more `PizzaD` foods.

<sup>46</sup> Good idea.

---

Here is one addition: `Spinach`.

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

---

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

<sup>47</sup> Yes, we must define three concrete methods for each variant of `PizzaD`.

Isn't that neat?

<sup>48</sup> No. When we add variants to the datatypes we have defined, we don't need to change what we have.

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

<sup>49</sup> Yes, this is a really flexible way of defining classes and methods. Unfortunately, the more things we want to do with `PizzaD`s, the more methods we must add.

---

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

<sup>50</sup> 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.

---

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

<sup>51</sup> Great.

<sup>52</sup> We will stick with anchovies.

450  
Come To Our  
Carousel



---

Wasn't this last collection overwhelming?

<sup>1</sup> It sure was. We defined seven classes and each contained three method definitions.

---

Could it get worse?

<sup>2</sup> It sure could. For everything we want to do with **Pizza<sup>D</sup>**, we must add a method definition to each class.

---

Why does that become overwhelming?

<sup>3</sup> 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.

<sup>4</sup> These methods look familiar. Have we seen them before?

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

<sup>V</sup> This superscript is a reminder that the class is a visitor class. Lower superscripts when you enter this kind of definition in a file: **OnlyOnionsV**.

---

Almost. Each of them corresponds to an *onlyOnions* method in one of the variants of **Shish<sup>D</sup>**.

<sup>5</sup> That's right. The major difference is that they are all in one class, a visitor, whose name is **OnlyOnions<sup>V</sup>**.

---

Is *onlyOnions* different from **OnlyOnions<sup>V</sup>**?

<sup>6</sup> The former is used to name methods, the latter names a class.

---

---

And that's the whole point.

<sup>7</sup> What point?

---

We want all the methods in one class.

<sup>8</sup> What methods?

---

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

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

---

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

<sup>10</sup> It's about time.

---

What is the difference between the method *onlyOnions* in the **Onion** variant and the method *forOnion* in the visitor **OnlyOnions<sup>V</sup>**?

<sup>11</sup> Everything following **return** is the same.

Right. What is the difference?

<sup>12</sup> The difference is that *onlyOnions* in **Onion** is followed by () and that *forOnion* in **OnlyOnions<sup>V</sup>** is followed by (**Shish<sup>D</sup> s**).

---

Yes, that is the difference. Are the other *for* methods in **OnlyOnions<sup>V</sup>** related to their counterparts in the same way?

<sup>13</sup> Indeed, they are.

---

It is time to discuss the boring part.

<sup>14</sup> What boring part?

---

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

<sup>15</sup> True, we still don't know how to make **Shish<sup>D</sup>** and its variants work with this visitor class, which contains all the action.

---

---

Now take a look at this.

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

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

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

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

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

---

<sup>16</sup> This is a strange set of definitions. All the *onlyOnions* methods in the variants look alike. Each of them uses an instance of *OnlyOnions<sup>V</sup>*, which is created in the datatype, to invoke a *for* method with a matching name.

---

What does the *forOnion* method in **Onion** consume?

<sup>17</sup> If “consume” refers to what follows the name between parentheses, the method consumes *s*, which is the rest of the shish.

---

That’s what “consumption” is all about. And what does the *forSkewer* method in **Skewer** consume?

<sup>18</sup> Nothing, because a skewer is the basis of a shish and therefore has no fields.

---

So what does the  $(\text{Shish}^D s)$  mean in the definition of *forOnion*?

<sup>19</sup> 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.

---

Very good. The notation  $(\text{Shish}^D s)$  means that *forOnion* consumes a **Shish**<sup>D</sup> and that between { and }, *s* stands for that shish.

<sup>20</sup> That makes sense and explains *s.onlyOnions()*.

---

Explain *s.onlyOnions()*.

<sup>21</sup> Here are our words:  
“*s* is a **Shish**<sup>D</sup>, and therefore *s.onlyOnions()* determines whether what is below the onion is also edible by an onion lover.”

---

Explain *ooFn.forOnion(s)*.

<sup>22</sup> 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**<sup>D</sup>, and *s* is the **Shish**<sup>D</sup> that represents what is below the onion.”

---

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

<sup>23</sup> It is still true.

---

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

<sup>24</sup> 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? <sup>25</sup> It checks whether the rest of this shish is edible by onion lovers.

---

How does it do that? <sup>26</sup> It uses the method *onlyOnions* on *s*.

---

Isn't that where we started from? <sup>27</sup> Yes, we're going round and round.

---

Welcome to the carousel. <sup>28</sup> 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 <sup>29</sup> That's exactly it.  
for this example the answer is **true**.

---

Do we need to remember that we are on a carousel? <sup>30</sup> 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? <sup>31</sup> 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.<sup>1</sup> <sup>32</sup> What about it?

```
abstract class ShishD {  
    OnlyOnionsV ooFn = new OnlyOnionsV();  
    IsVegetarianV ivFn = new IsVegetarianV();  
    abstract boolean onlyOnions();  
    abstract boolean isVegetarian();  
}
```

<sup>1</sup> *The American Heritage Dictionary* defines protocol as “[t]he 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.

---

---

Write the rest!

<sup>33</sup> We must add two lines to each variant, and they are almost the same as those for *ooFn*.

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

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

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

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

---

That's why we call this part boring.

<sup>34</sup> True, there's very little to think about. It could be done automatically.

---

How do we define the visitor?

<sup>35</sup> Does that refer to the class that contains the actions?

---

Yes, that one. Define the visitor.

<sup>36</sup> It is like `OnlyOnionsV` except for the method `forTomato`.

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

---

Are we moving fast?

<sup>37</sup> 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?

<sup>38</sup> 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.*

---

Is

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

a shish kebab?

<sup>39</sup> No, it's a pizza.

```
abstract class PizzaD {}
```

```
class Crust extends PizzaD {}
```

```
class Cheese extends PizzaD {  
    PizzaD p;  
    Cheese(PizzaD ~p) {  
        p = ~p; }  
}
```

```
class Olive extends PizzaD {  
    PizzaD p;  
    Olive(PizzaD ~p) {  
        p = ~p; }  
}
```

```
class Anchovy extends PizzaD {  
    PizzaD p;  
    Anchovy(PizzaD ~p) {  
        p = ~p; }  
}
```

```
class Sausage extends PizzaD {  
    PizzaD p;  
    Sausage(PizzaD ~p) {  
        p = ~p; }  
}
```

---

So what do we do next?

<sup>40</sup> We can define the protocol for the methods that belong to `PizzaD` and its extensions: `remA`, `topAwC`, and `subAbC`.

---

---

Great! Here is the abstract portion of the protocol.

```
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.

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

```
class Cheese extends PizzaD {
    PizzaD p;
    Cheese(PizzaD _p) {
        p = _p; }

    PizzaD remA() {
        return remFn.forCheese(p); }
    PizzaD topAwC() {
        return topFn.forCheese(p); }
    PizzaD subAbC() {
        return subFn.forCheese(p); }
}
```

Define the rest.

<sup>41</sup> How innovative! The variants are totally mindless, now.

```
class Olive extends PizzaD {
    PizzaD p;
    Olive(PizzaD _p) {
        p = _p; }

    PizzaD remA() {
        return remFn.forOlive(p); }
    PizzaD topAwC() {
        return topFn.forOlive(p); }
    PizzaD subAbC() {
        return subFn.forOlive(p); }
}
```

```
class Anchovy extends PizzaD {
    PizzaD p;
    Anchovy(PizzaD _p) {
        p = _p; }

    PizzaD remA() {
        return remFn.forAnchovy(p); }
    PizzaD topAwC() {
        return topFn.forAnchovy(p); }
    PizzaD subAbC() {
        return subFn.forAnchovy(p); }
}
```

```
class Sausage extends PizzaD {
    PizzaD p;
    Sausage(PizzaD _p) {
        p = _p; }

    PizzaD remA() {
        return remFn.forSausage(p); }
    PizzaD topAwC() {
        return topFn.forSausage(p); }
    PizzaD subAbC() {
        return subFn.forSausage(p); }
}
```

---

We are all set.

<sup>42</sup> Is it time to define the visitors that correspond to the methods *remA*, *topAwC*, and *subAbC*?

---

Okay, here is *RemA<sup>V</sup>*.

```
class RemAV {
    PizzaD forCrust() {
        return new Crust();
    }
    PizzaD forCheese(PizzaD p) {
        return new Cheese(p.remA());
    }
    PizzaD forOlive(PizzaD p) {
        return new Olive(p.remA());
    }
    PizzaD forAnchovy(PizzaD p) {
        return p.remA();
    }
    PizzaD forSausage(PizzaD p) {
        return new Sausage(p.remA());
    }
}
```

Define *TopAwC<sup>V</sup>*.

<sup>43</sup> By now, even this is routine.

```
class TopAwCV {
    PizzaD forCrust() {
        return new Crust();
    }
    PizzaD forCheese(PizzaD p) {
        return new Cheese(p.topAwC());
    }
    PizzaD forOlive(PizzaD p) {
        return new Olive(p.topAwC());
    }
    PizzaD forAnchovy(PizzaD p) {
        return
            new Cheese(
                new Anchovy(p.topAwC()));
    }
    PizzaD forSausage(PizzaD p) {
        return new Sausage(p.topAwC());
    }
}
```

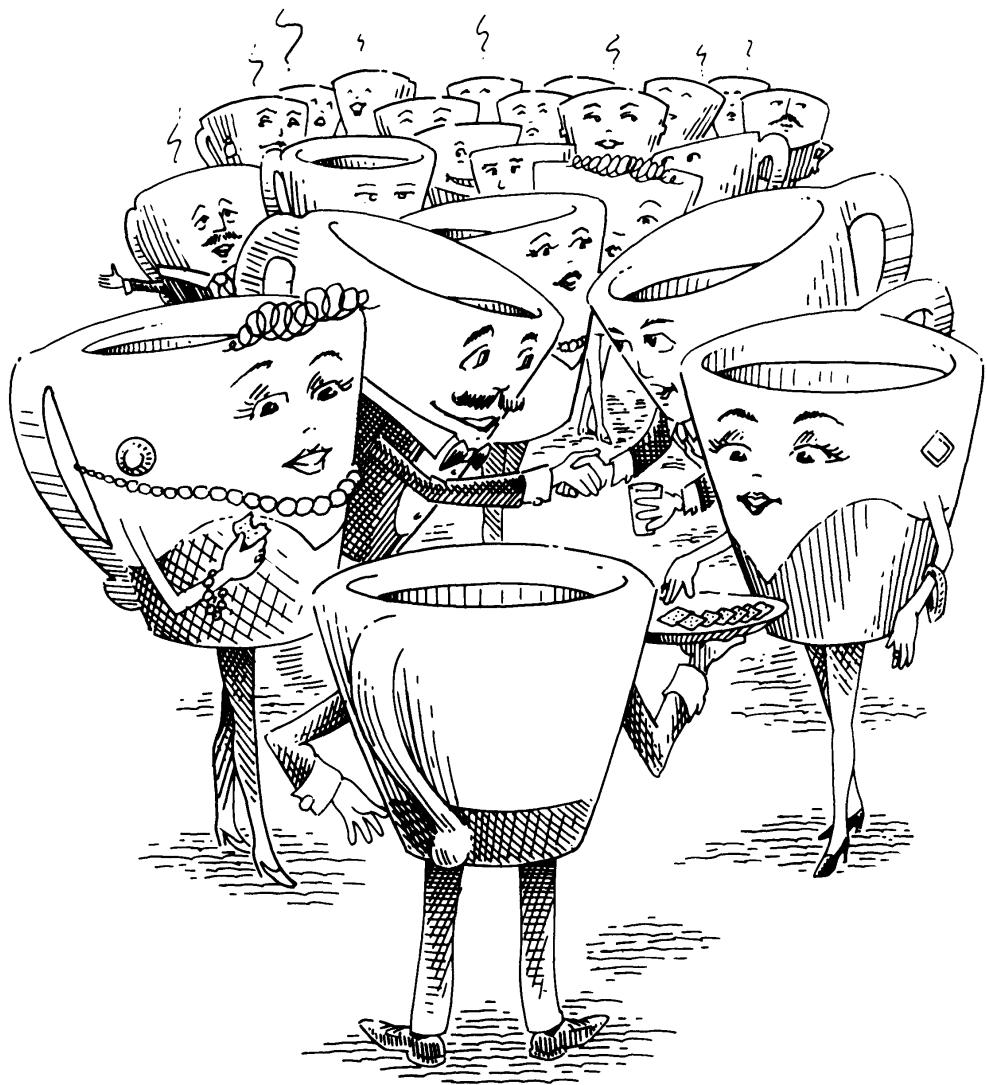
---

The last one, *SubAbC<sup>V</sup>*, is a piece of cake.

<sup>44</sup> And we thought we were working with a pizza pie.

```
class SubAbCV {
    PizzaD forCrust() {
        return new Crust();
    }
    PizzaD forCheese(PizzaD p) {
        return new Cheese(p.subAbC());
    }
    PizzaD forOlive(PizzaD p) {
        return new Olive(p.subAbC());
    }
    PizzaD forAnchovy(PizzaD p) {
        return new Cheese(p.subAbC());
    }
    PizzaD forSausage(PizzaD p) {
        return new Sausage(p.subAbC());
    }
}
```

5.  
Objects Are  
People, Too



---

Have we seen this kind of definition before?<sup>1</sup> <sup>1</sup> What? More pizza!

```
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
```

<sup>1</sup> Better names for these classes would be `PizzaPieD`, `Bottom` and `Topping`, respectively.

---

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

<sup>2</sup> 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?

<sup>3</sup> Any kind, because `Object` is the class of all objects. Here are some fish toppings.

```
abstract class FishD {}
```

```
class Anchovy extends FishD {}
```

```
class Salmon extends FishD {}
```

```
class Tuna extends FishD {}
```

---

Nice datatype. Is

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

a pizza pie?

<sup>4</sup> It is a pizza pie, and so is

```
new Top(new Tuna(),  
new Top(new Integer(42),  
new Top(new Anchovy(),  
new Top(new Integer(5),  
new Bot()))))).
```

What is the value of

```
new Top(new Salmon(),  
new Top(new Anchovy(),  
new Top(new Tuna(),  
new Top(new Anchovy(),  
new Bot()))))  
.remA()?
```

<sup>5</sup> It is this fishy pizza pie:

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

Is it true that the value of

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

is

```
new Top(new Salmon(),  
new Top(new Tuna(),  
new Bot()))?
```

<sup>6</sup> Yes. The pizza that comes out is the same as the one that goes in, because there are no anchovies on that pizza.

---

Does *remA* belong to  $\text{Pie}^D$ ?

<sup>7</sup> Yes, and it produces pizza pies.

Define the protocol for  $\text{RemA}^V$ . We provide the **abstract** part.

```
| RemAV raFn = new RemAV();  
| abstract PieD remA();
```

<sup>8</sup> This is easy by now.

```
| Pie | PieD remA() {  
| | return raFn.forBot(); }  
| | Bot  
| | Pie | PieD remA() {  
| | | return raFn.forTop(t,r); }  
| | | Top  
| | |
```

---

Great. Isn't that easy?

<sup>9</sup> Easy and boring.

---

What part of this exercise differs from datatype to datatype?

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

---

Anything else?

<sup>11</sup> No, from that we know that *raFn.forBot* is followed by () and *raFn.forTop* by (t,r).

---

Why (t,r)?

<sup>12</sup> Because these are the fields of Top.

---

Let's define the visitor RemA<sup>V</sup>.

<sup>13</sup> Here are some guesses.

```
class RemAV {  
    PieD forBot() {  
        return _____; }  
    PieD forTop(Object t,PieD r) {  
        if (new Anchovy().equals(t))  
            return _____;  
        else  
            return _____; }  
}
```

```
class RemAV {  
    PieD forBot() {  
        return new Bot(); }  
    PieD forTop(Object t,PieD r) {  
        if (new Anchovy().equals(t))  
            return r.remA();  
        else  
            return new Top(t,r.remA()); }  
}
```

---

Great guesses! What does

```
if (expr1)  
    return expr2;  
else  
    return expr3;
```

mean?

<sup>14</sup> We guess:

"This produces the value of either *expr*<sub>2</sub> or *expr*<sub>3</sub>, depending on whether or not *expr*<sub>1</sub> is determined to be true or false." respectively."

---

And what does

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

<sup>15</sup> We could guess:

"This expression determines whether *t* is equal to **new Anchovy()**."

---

Not yet. It depends on what *equals* means.

<sup>16</sup> What?

---

---

What is the value of

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

<sup>17</sup> The “Not yet.” implies that the value is `false`.

---

Yes! And what is the value of

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

<sup>18</sup> `false`,

because no anchovy is a tuna.

---

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

<sup>19</sup> If we know that `equals`’s answer is always `false`, why bother to use it?

<sup>1</sup> Not always. We explain the correct answer in chapter 10.

---

We must define it anew<sup>1</sup> for all classes whose instances we wish to compare.

<sup>20</sup> Okay. How?

<sup>1</sup> In Java, redefining a method is called “overriding.”

---

For `FishD` and its variants it works like this.

```
abstract class FishD {}
```

<sup>21</sup> Assuming that

`(o instanceof Tuna)`

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

```
class Anchovy extends FishD {
    public1 boolean equals(Object o) {
        return (o instanceof Anchovy);
    }
}
```

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

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

<sup>1</sup> The class `Object` is defined in a separate package, called `java.lang.Object`. Overriding methods that reside in other packages requires the word `public`.

---

---

Aren't they? Is every value constructed with **new** an instance of **Object**?<sup>22</sup>

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

---

If class **A extends B**, is every value created by **new A(...)** an instance of class **B**?<sup>23</sup>

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

---

Now, what is the value of

**new Anchovy().equals(new Anchovy())?**

<sup>24</sup> true,

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

---

Yet the value of

**new Anchovy().equals(new Tuna())**

is still false.

<sup>25</sup> Of course, because an anchovy is never a tuna.

---

Could we have written **RemA<sup>V</sup>** without using **equals**?<sup>26</sup>

Absolutely, **instanceof** is enough.

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

Why haven't we defined it this way?

---

Easy, because we want to generalize **RemA<sup>V</sup>** so that it works for any kind of fish topping.

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

---

What are good names for the more general methods and visitor?<sup>28</sup>

How about **remFish** and **RemFish<sup>V</sup>**?

---

How do we use *remFish*?

<sup>29</sup> We give it a  $\text{Fish}^{\mathcal{D}}$ .

---

Add the protocol for  $\text{RemFish}^{\mathcal{V}}$ . We designed the abstract portion.

```
RemFishV rfFn = new RemFishV();
abstract PieD remFish(FishD f);
```

Pie

<sup>30</sup> The rest is routine.

```
PieD remFish(FishD f) {
    return rfFn.forBot(f); }
```

Bot

```
PieD remFish(FishD f) {
    return rfFn.forTop(t,r,f); }
```

Top

---

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

<sup>31</sup> The  $f$  stands for the  $\text{Fish}^{\mathcal{D}}$  we want to remove in both cases. The  $t$  and the  $r$  are the fields of  $\text{Top}$ ;  $\text{Bot}$  doesn't have any.

---

Let's define  $\text{RemFish}^{\mathcal{V}}$  and its two methods.

<sup>32</sup> Instead of comparing the top layer  $t$  of the pizza to  $\text{Anchovy}$ , we now determine whether it equals the  $\text{Fish}^{\mathcal{D}} f$ , which is the additional value consumed by the method.

```
class RemFishV {
    PieD forBot(FishD f) {
        return new Bot(); }
    PieD forTop(Object t,PieD r,FishD 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  $\text{RemFish}^{\mathcal{V}}$ ?

<sup>33</sup> Nothing, we just have to remember to add *equals* to the new variant.

---

---

Let's try it out with a short example:

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

---

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

Yes. What values does *forTop* consume?

<sup>35</sup> 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<sup>D</sup> to be removed.

---

And now?

<sup>36</sup> Now we need to determine the value of

```
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?

<sup>37</sup> Given what *f* and *t* stand for, *f.equals(t)* is true. Hence, we must determine the value of *r.remFish(f)*.

---

What is the value of

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

<sup>38</sup> It is the same as

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

---

What does *forBot* in RemFish<sup>V</sup> produce?

<sup>39</sup> It produces **new Bot()**, no matter what *f* is.

---

All clear?

<sup>40</sup> Ready to move on, after snack time.

---

---

Does

```
new Top(new Integer(2),
        new Top(new Integer(3),
            new Top(new Integer(2),
                    new Bot())))
    .remInt(new Integer(3))
```

look familiar?

<sup>41</sup> Yes, it looks like what we just evaluated.

---

What does *remInt* do?

<sup>42</sup> It removes **Integers** from pizza pies just as *remFish* removes fish from pizza pies.

---

Who defined *equals* for **Integer**?

<sup>43</sup> The Machine decided

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

to be true, and the rest was obvious.

---

Define the visitor **RemInt**<sup>V</sup>.

<sup>44</sup> Wonderful! We do the interesting thing first. This visitor is almost identical to **RemFish**<sup>V</sup>. We just need to change the type of what the two methods consume.

```
class RemIntV {
    PieP forBot(Integer i) {
        return new Bot();
    }
    PieP forTop(Object t,PieP r,Integer i) {
        if (i.equals(t))
            return r.remInt(i);
        else
            return new Top(t,r.remInt(i));
    }
}
```

---

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

<sup>45</sup> No, *i* is just a better name than *f*, no other reason. As long as we do such substitutions systematically, we are just fine.

---

Where is the protocol?

<sup>46</sup> It is so simple, let's save it for later.

- 
- Can we remove `Integers` from  $\text{Pie}^{\mathcal{D}}$ s? <sup>47</sup> Yes.
- 
- Can we remove `Fish $^{\mathcal{D}}$`  from  $\text{Pie}^{\mathcal{D}}$ s? <sup>48</sup> Yes, and we use nearly identical definitions.
- 
- Let's combine the two definitions. <sup>49</sup> If we use `Object` instead of the underlined `Integer` above, everything works out.
- 
- Why? <sup>50</sup> Because everything constructed with `new` is an `Object`.
- 
- Just do it! <sup>51</sup> It's done.
- ```
class Rem $^{\mathcal{V}}$  {
    Pie $^{\mathcal{D}}$  forBot(Object o) {
        return new Bot(); }
    Pie $^{\mathcal{D}}$  forTop(Object t,Pie $^{\mathcal{D}}$  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? <sup>52</sup> Now?
- 
- You never know when it might be useful, even if it does not contain any interesting information. <sup>53</sup> Let's just consider  $\text{Rem}^{\mathcal{V}}$ .
- 
- Why not  $\text{RemFish}^{\mathcal{V}}$  and  $\text{RemA}^{\mathcal{V}}$  and  $\text{RemInt}^{\mathcal{V}}$ ? <sup>54</sup> They are unnecessary once we have  $\text{Rem}^{\mathcal{V}}$ .
-

---

Here is the **abstract** portion of  $\text{Pie}^D$ .

```
abstract class PieD {
    RemV remFn = new RemV();
    abstract PieD rem(Object o);
}
```

<sup>55</sup> And here are the pieces for **Bot** and **Top**.

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

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

```
PieD rem(Object o) {
    return remFn.forTop(t,r,o);
}
```

---

Let's remove some things from pizza pies:

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

<sup>56</sup> Works like a charm with the same result as before.

---

And how about

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

<sup>57</sup> Ditto.

---

Next:

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

<sup>58</sup> No problem. This, too, removes 3 and leaves the other layers alone:

```
new Top(new Anchovy(),
        new Top(new Zero(),
               new Bot())).
```

---

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

---

What's wrong with that?

And why didn't it?

Always?

Here is the version of Num<sup>D</sup> (including OneMoreThan) with its own *equals*. Define the new Zero variant.

```
abstract class NumD {}

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

    public boolean equals(Object o) {
        if (o instanceof OneMoreThan)
            return
                predecessor
                    .equals(
                        ((OneMoreThan)o)1.predecessor);
        else
            return false;
    }
}
```

<sup>59</sup> Oops. The answer is  
new Top(new Anchovy(),  
new Top(new Integer(3),  
new Top(new Zero(),  
new Bot())))).

<sup>60</sup> We expected it to remove new Zero() from the pizza.

<sup>61</sup> Because *equals* for Num<sup>D</sup>'s uses Object's *equals*, which always produces *false*—as we discussed above when we introduced *equals*.

<sup>62</sup> Unless we define it anew for those classes whose instances we wish to compare.

<sup>63</sup> Adding *equals* to Zero is easy. We use **instanceof** to determine whether the consumed value is a new Zero().

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

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

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

---

No. What is the type of *o*?

<sup>64</sup> Object, according to  
(Object *o*),  
which is what declares the type of *o*.

---

So what is *o.predecessor*?

<sup>65</sup> Nonsense.

---

Correct. What do we know after if has determined that

(*o instanceof OneMoreThan*)  
is true?

<sup>66</sup> We know that *o*'s type is Object and that it is an instance of OneMoreThan.

---

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

<sup>67</sup> It converts the type of *o* from Object to OneMoreThan.

---

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

<sup>68</sup> Its type is OneMoreThan, and now it makes sense to write  
((OneMoreThan) *o*).*predecessor*.

---

Are *o* and ((OneMoreThan)*o*) interchangeable?

<sup>69</sup> The underlying object is the same. But no, the two expressions are not interchangeable, because the former's type is Object, whereas the latter's is OneMoreThan.

---

Is this complicated?

<sup>70</sup> Someone has been drinking too much coffee.

---

Did you also notice the  
*predecessor*  
.equals(  
((OneMoreThan)*o*).*predecessor*)  
in *equals* for OneMoreThan?

<sup>71</sup> How do the two uses of *predecessor* differ?

---

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.

<sup>72</sup> 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?

<sup>73</sup> If they are similar<sup>1</sup> to the same **int**, they are equal. But most of the time, they are not.

<sup>1</sup> Check chapter 1 for "similar."

---

Time for lunch?

<sup>74</sup> That's just in time.

---

Did you have a good lunch break?

<sup>75</sup> Yes, thank you.

---

Now what is the value of  
    **new Top(new Anchovy(),**  
    **new Top(new Integer(3),**  
    **new Top(new Zero(),**  
        **new Bot()))))**  
.rem(**new Zero()**)?

<sup>76</sup> Now we get  
    **new Top(new Anchovy(),**  
    **new Top(new Integer(3),**  
        **new Bot()))),**  
which is precisely what we want.

---

And why?

<sup>77</sup> Because *equals* now knows how to compare  $\text{Num}^{\mathcal{D}}$ s.

---

Do we always add *equals* to a class?  
substitute one item for another on a pizza pie?

<sup>78</sup> No, only if we need it.

---

What is the value of  
    **new Top(new Anchovy(),**  
    **new Top(new Tuna(),**  
    **new Top(new Anchovy(),**  
        **new Bot()))))**  
.substFish(**new Salmon(),**  
        **new Anchovy()**)?

<sup>79</sup> Yes, we do.  
<sup>80</sup> It is the same pizza pie with all the anchovies replaced by salmon:  
    **new Top(new Salmon(),**  
    **new Top(new Tuna(),**  
    **new Top(new Salmon(),**  
        **new Bot())))).**

---

What kind of values does *substFish* consume? <sup>81</sup> It consumes two *fish* and works on  $\text{Pie}^{\mathcal{D}}$ s.

---

---

And what does it produce?

<sup>82</sup> It always produces a  $\text{Pie}^D$ .

---

What is the value of

```
new Top(new Integer(3),
        new Top(new Integer(2),
                new Top(new Integer(3),
                        new Bot())))
        .substInt(new Integer(5),
                  new Integer(3))?)
```

<sup>83</sup> It is the same pizza pie with all 3s replaced by 5s:

```
new Top(new Integer(5),
        new Top(new Integer(2),
                new Top(new Integer(5),
                        new Bot()))).
```

---

What kind of values does  $\text{substInt}$  consume?

<sup>84</sup> It consumes two **Integers** and works on  $\text{Pie}^D$ s.

---

And what does it produce?

<sup>85</sup> It always produces a  $\text{Pie}^D$ .

---

We can define  $\text{SubstFish}^V$ .

```
class SubstFishV {
    PieD forBot(FishD n,FishD o) {
        return new Bot(); }
    PieD forTop(Object t,
                 PieD r,
                 FishD n,
                 FishD o) {
        if (o.equals(t))
            return new Top(n,r.substFish(n,o));
        else
            return new Top(t,r.substFish(n,o)); }
```

Define  $\text{SubstInt}^V$ .

<sup>86</sup> To get from  $\text{SubstFish}^V$  to  $\text{SubstInt}^V$ , we just need to substitute  $\text{Fish}^D$  by  $\text{Integer}$  everywhere and ‘Fish’ by ‘Int’ in the class and method names.

```
class SubstIntV {
    PieD forBot(Integer n,Integer o) {
        return new Bot(); }
    PieD forTop(Object t,
                 PieD r,
                 Integer n,
                 Integer o) {
        if (o.equals(t))
            return new Top(n,r.substInt(n,o));
        else
            return new Top(t,r.substInt(n,o)); }
```

---

Did we forget the boring parts?

<sup>87</sup> Yes, because there is obviously a more general version like  $\text{Rem}^V$ .

---

---

Yes, we call it  $\text{Subst}^V$ . Define it.

<sup>88</sup> We substitute  $\text{Object}$  for  $\text{Fish}^D$  and  $\text{Integer}$ .

```
class SubstV {
    PieD forBot(Object n, Object o) {
        return new Bot();
    }
    PieD forTop(Object t,
                  PieD r,
                  Object n,
                  Object o) {
        if (o.equals(t))
            return new Top(n, r.subst(n, o));
        else
            return new Top(t, r.subst(n, o));
    }
}
```

---

Now it is time to add the protocol for  $\text{Subst}^V$ <sup>89</sup> to  $\text{Pie}^D$ . Here are the variants.

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

The abstract part is obvious.

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

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

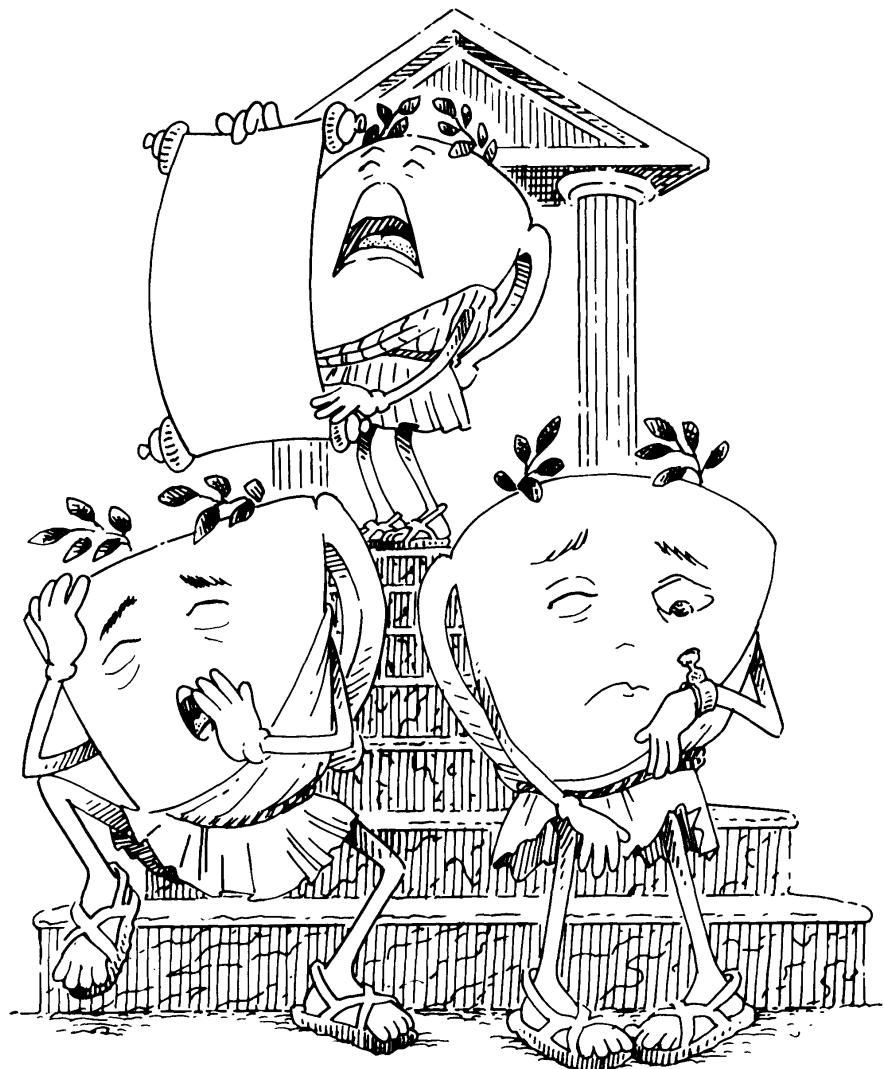
    PieD rem(Object o) {
        return remFn.forTop(t, r, o);
    }
    PieD subst(Object n, Object o) {
        return substFn.forTop(t, r, n, o);
    }
}
```

---

So?

<sup>90</sup> That was some heavy lifting.

6  
Boring Protocols



---

Are protocols truly boring?

<sup>1</sup> 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.

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

<sup>2</sup> Okay, here are the variants again.

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

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

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

---

What is the difference between *rem* and *subst* in *Pie<sup>D</sup>*?

<sup>3</sup> The first one consumes one *Object*, the second one consumes two.

---

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

<sup>4</sup> 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?

<sup>5</sup> Simpler: *rem* asks for the *forTop* service from *remFn* and hands over the field values and the *Object* it consumes; *subst* asks for the *forTop* service from *substFn* and hands over the field values and the two *Objects* it consumes.

---

---

And that is all there is to the methods in the<sup>6</sup>. But *remFn* and *substFn* defined in the variants of a protocol.

---

Let's not create *remFn* and *substFn* in the datatype.

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

---

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*?

Here is how it changes **Top**.

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

---

How does it affect **Bot**?

<sup>7</sup> 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?

<sup>8</sup> The definition of the datatype says that they are a *Rem<sup>V</sup>* and a *Subst<sup>V</sup>*, respectively. And every *Rem<sup>V</sup>* defines *forBot* and *forTop*, and so does every *Subst<sup>V</sup>*.

<sup>9</sup> In the same manner. We just need to change each concrete method's description of what it consumes. The rest remains the same.

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

---

---

That's right. Nothing else changes in the variants. Instead of relying on fields of the datatype, we use what is consumed.

<sup>10</sup> We still have some work to do.

Like what?

<sup>11</sup> Consuming an extra value here also affects how the methods *rem* and *subst* are used.

Where are they used?

<sup>12</sup> In *Rem<sup>V</sup>* and *Subst<sup>V</sup>*, the interesting parts, for example.

Yes. Here is *Rem<sup>V</sup>*.

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

Modify *Subst<sup>V</sup>* accordingly.

<sup>13</sup> That takes all the fun out of it.

```
class SubstV {
    PieD forBot(Object n,
                  Object o) {
        return new Bot(); }
    PieD forTop(Object t,
                  PieD 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)); }
}
```

---

What is **this** all about?

<sup>14</sup> Yes, what about it. Copying is easy.

---

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

<sup>15</sup> Which object?

---

How did we get here?

<sup>16</sup> The protocol is that *rem* in *Bot* and *Top* asks for the *forBot* and *forTop* methods of *remFn*, respectively.

---

How does that happen?

<sup>17</sup> It happens with

*remFn.forBot( . . . )*

and

*remFn.forTop( . . . ),*

respectively.

---

Correct. And now *forBot* and *forTop* can refer to the object *remFn* as **this**.

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

---

That's it. Tricky?

<sup>19</sup> Not really, just self-referential.

---

Why?

<sup>20</sup> Because **this** is a  $\text{Rem}^V$ , and it is exactly what we need to complete the job.

---

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

---

<sup>21</sup> 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?

<sup>22</sup> It creates a  $\text{Rem}^V$  object, which corresponds to the *remFn* in the old  $\text{Pie}^D$ .

---

What is the value of  
`new Top(new Integer(3),  
 new Top(new Integer(2),  
 new Top(new Integer(3),  
 new Bot()))))  
.subst(new SubstV(),  
 new Integer(5),  
 new Integer(3))?`

---

<sup>23</sup> 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?

<sup>24</sup> It creates a **Subst<sup>V</sup>** object, which corresponds to the *remFn* in the old **Pie<sup>D</sup>**.

---

So what is the underlined part about?

<sup>25</sup> We changed the methods in **Pie<sup>D</sup>**, which means that we must also change how it is used.

---

Ready for the next protocol?

<sup>26</sup> Let's grab a quick snack.

---

How about some ice cream?

<sup>27</sup> Cappuccino crunch sounds great. The more coffee, the better.

---

Take a look at *subst* in **Top** and at *forTop* in **Subst<sup>V</sup>**. What happens to the values that they consume?

<sup>28</sup> Nothing really. They get handed back and forth, though *forTop* compares *o* to *t*.

---

Is the handing back and forth necessary?

<sup>29</sup> We don't know any better way, yet.

---

Here is a way to define **Subst<sup>V</sup>** that avoids the handing back and forth of these extra values.

<sup>30</sup> Wow. This visitor has two fields.<sup>1</sup>

```
class SubstV {
    Object n;
    Object o;
    SubstV(Object _n, Object _o) {
        n = _n;
        o = _o; }

    PieD forBot() {
        return new Bot(); }
    PieD forTop(Object t, PieD r) {
        if (o.equals(t))
            return new Top(n, r.subst(this));
        else
            return new Top(t, r.subst(this)); }
}
```

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

---

How do we create a  $\text{Subst}^V$ ?

<sup>31</sup> We use

```
new SubstV(new Integer(5),  
new Integer(3)).
```

---

What does that do?

<sup>32</sup> It creates a  $\text{Subst}^V$  whose methods know how to substitute **new Integer(5)** for all occurrences of **new Integer(3)** in  $\text{Pie}^D$ .

---

How do the methods know that without consuming more values?

<sup>33</sup> The values have now become fields of the  $\text{Subst}^V$  object to which the methods belong. They no longer need to be consumed.

---

Okay, so how would we *substitute* all **new Integer(3)** with **new Integer(5)** in  
**new Top(new Integer(3)),**  
**new Top(new Integer(2)),**  
**new Top(new Integer(3),**  
**new Bot()))**?

<sup>34</sup> We write

```
new Top(new Integer(3),  
new Top(new Integer(2),  
new Top(new Integer(3),  
new Bot()))))  
.subst(new SubstV(  
new Integer(5),  
new Integer(3))).
```

---

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

<sup>35</sup> We write

```
new Top(new Integer(3),  
new Top(new Integer(2),  
new Top(new Integer(3),  
new Bot()))))  
.subst(new SubstV(  
new Integer(7),  
new Integer(2))).
```

---

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

<sup>36</sup> Of course, because the methods *subst* in the **Bot** and **Top** variants consume only one value now.

---

---

That's right. Here are the datatype and its Bot variant. Define the Top variant.

```
abstract class PieD {
    abstract PieD rem(RemV remFn);
    abstract PieD subst(SubstV substFn);
}
```

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

<sup>37</sup> In the Top variant, we still need to hand over both *t* and *r*.

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

```
PieD rem(RemV remFn) {
    return remFn.forTop(t,r);
}
PieD subst(SubstV substFn) {
    return substFn.forTop(t,r);
}
```

---

Is there anything else missing?

<sup>38</sup> We haven't defined Rem<sup>V</sup> for this new protocol. But it is simple and hardly worth our attention.

---

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

<sup>39</sup> Not much. The name of the respective values they consume and the corresponding types.

---

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

<sup>40</sup> Not much. The name of the respective values they consume and the corresponding types.

---

Can we eliminate the differences?

<sup>41</sup> It is easy to make them use the same names. It doesn't matter whether *rem* is defined as it is or as

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

---

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

<sup>42</sup> Both Rem<sup>V</sup> and Subst<sup>V</sup> 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**.

---

Yes! Do it!

<sup>43</sup> Here it is.

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

---

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

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

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

---

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

<sup>44</sup> Okay, that doesn't seem to be a great difference. Can a class extend an **interface** the way it **extends** an **abstract class**?

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

<sup>45</sup> Yes, we can. Assuming we can use **interfaces** like **abstract classes**, we can write

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

and

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

in **Top**.

---

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

<sup>47</sup> 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?

<sup>48</sup> The method accepts a visitor and asks for its services, so we call it *accept*.

---

And what is a better name for *pvFn*?

<sup>49</sup> Easy: *ask*, because we ask for services.

---

Now we can simplify the protocol. Here is the new *Rem*<sup>V</sup>.

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

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

Supply the protocol.

<sup>50</sup> Here we go.

```
abstract class PieD {
    abstract PieD accept(PieVisitorI ask);
}
```

```
class Bot extends PieD {
    PieD accept(PieVisitorI ask) {
        return ask.forBot();
    }
}
```

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

```
PieD accept(PieVisitorI ask) {
    return ask.forTop(t,r);
}
```

---

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

<sup>51</sup> 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.

<sup>52</sup> Why?

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

<sup>53</sup> Looks weird, but let's move on.

---

Correct. They are just icing.

<sup>54</sup> Okay, we still won't forget them.

---

---

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

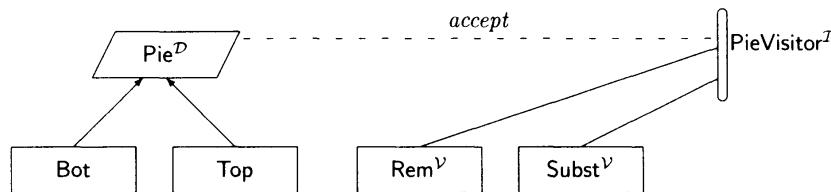
<sup>55</sup> Here it is.

```
class SubstV implements PieVisitorI {  
    Object n;  
    Object o;  
    SubstV(Object _n, Object _o) {  
        n = _n;  
        o = _o; }  
  
    public PieD forBot() {  
        return new Bot(); }  
    public PieD forTop(Object t, PieD 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  $\text{PieVisitor}^I$  and all the classes:  $\text{Pie}^D$ ,  $\text{Bot}$ ,  $\text{Top}$ ,  $\text{Rem}^V$ , and  $\text{Subst}^V$ .

<sup>56</sup> Here is our picture.



---

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

<sup>57</sup> The  $\text{Subst}^V$  visitor **implements**  $\text{PieVisitor}^I$ , it doesn't *extend* it. Arrows mean "extends," lines mean "implements."

---

And the dashed line?

<sup>58</sup> 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()))?
```

<sup>59</sup> Easy:

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

---

Explain what  $\text{LtdSubst}^V$  produces.<sup>1</sup>

<sup>1</sup> A better name is `LimitedSubstitutionV`, and that is how we pronounce it.

<sup>60</sup> The methods of  $\text{LtdSubst}^V$  replace one fish on a pie by another as many times as specified by the first value consumed by  $\text{LtdSubst}^V$ .

---

Good. Define  $\text{LtdSubst}^V$ .

<sup>61</sup> That's easy. We have such a flexible protocol that we only need to define the essence now.

```
class LtdSubstV implements PieVisitorT {  
    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)); }  
}
```

---

---

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<sup>V</sup>(2,**  
            **new Salmon(),**  
            **new Anchovy())?**

---

How come?

<sup>62</sup> Oops, 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())))).**

Why doesn't *c* ever change?

<sup>63</sup> Because *c*, the counting field, never changes.

---

Can we fix **this**?

<sup>64</sup> Because **this**, the **LtdSubst<sup>V</sup>** that performs the substitutions, never changes.

---

If *c* stands for the current count, how do we create a **LtdSubst<sup>V</sup>** that shows that we have just substituted one fish by another.

<sup>65</sup> We can't change **this**, but we can replace **this** with a new **LtdSubst<sup>V</sup>** that reflects the change.

---

<sup>66</sup> Simple, we use  
    **new LtdSubst<sup>V</sup>(*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  $\text{LtdSubst}^V$ .

<sup>67</sup> Voilà.

```
class LtdSubstV implements PieVisitorT {
    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(
                            new LtdSubstV(c - 1, n, o)));
            else
                return
                    new Top(t,
                        r.accept(
                            this)); }
}
```

---

How does  
    **this**  
differ from  
    **new LtdSubst**<sup>V</sup>( $c - 1, n, o$ )?

<sup>68</sup> They are two different  $\text{LtdSubst}^V$ s. One replaces  $c$  occurrences of  $o$  by  $n$  in a pizza pie, and the other one replaces only  $c - 1$  of them.

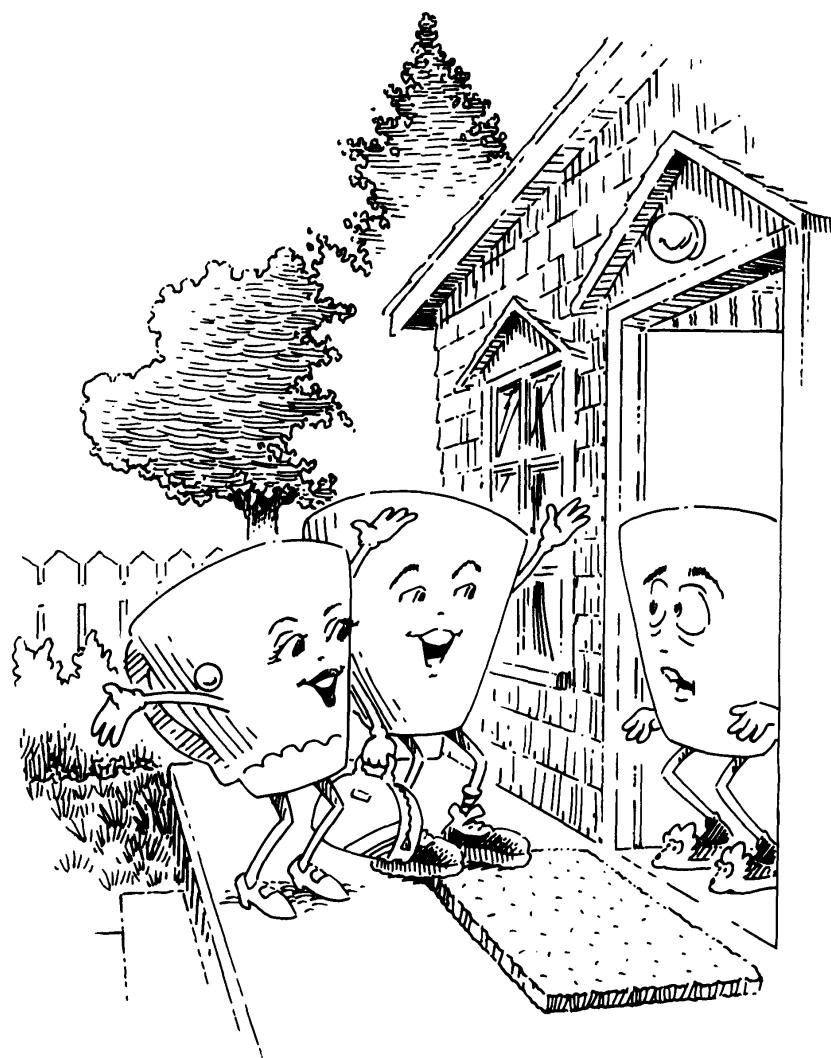
---

How do you feel about protocols now?

<sup>69</sup> They are exciting. Let's do more.

---

Zo  
Oh My !



---

Is

**new Flat(new Apple(),  
new Flat(new Peach(),  
new Bud()))**

a flat  $\text{Tree}^D$ ?

<sup>1</sup> Yes.

•

---

Is

**new Flat(new Pear(),  
new Bud())**

a flat  $\text{Tree}^D$ ?

<sup>2</sup> Yes, it is also a flat  $\text{Tree}^D$ .

---

And how about

**new Split(  
new Bud(),  
new Flat(new Fig(),  
new Split(  
new Bud(),  
new Bud()))))**?

<sup>3</sup> 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()))))**.

<sup>4</sup> No, it isn't flat either.

Is it flat?

---

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

<sup>5</sup> Unless there is anything else to  $\text{Tree}^D$ , it's totally clear.

---

Good. Then let's move on.

<sup>6</sup> Okay, let's.

---

Here are some fruits.

```
abstract class FruitD {}
```

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

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

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

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

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

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

---

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

---

Do `TreeD`'s variants contain `equals`?

---

<sup>7</sup> It does not differ too much from what we have seen before.

```
abstract class TreeD {}
```

```
class Bud extends TreeD {}
```

```
class Flat extends TreeD {  
    FruitD f;  
    TreeD t;  
    Flat(FruitD _f,TreeD _t) {  
        f = _f;  
        t = _t; }  
}
```

```
class Split extends TreeD {  
    TreeD l;  
    TreeD r;  
    Split(TreeD _l,TreeD _r) {  
        l = _l;  
        r = _r; }  
}
```

<sup>8</sup> That probably means that we will need to compare fruits and other things.

---

<sup>9</sup> No, which means we won't compare them, but we could.

---

---

How does the datatype  $\text{Tree}^D$  differ from all the other datatypes we have seen before?

<sup>10</sup> The name of the new datatype occurs twice in its `Split` variant.

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

```
interface bTreeVisitorT {
    boolean forBud();
    boolean forFlat(FruitD f, TreeD t);
    boolean forSplit(TreeD l, TreeD r);
}
```

Here is the new datatype definition.

```
abstract class TreeD {
    abstract
        boolean accept(bTreeVisitorT ask);
}
```

Revise the variants.

<sup>11</sup> That just means extending what we have with one method each.

```
class Bud extends TreeD {
    boolean accept(bTreeVisitorT ask) {
        return ask.forBud();
    }
}
```

```
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);
}
}
```

```
class Split extends TreeD {
    TreeD l;
    TreeD r;
    Split(TreeD _l, TreeD _r) {
        l = _l;
        r = _r;
    }
}
```

```
boolean accept(bTreeVisitorT ask) {
    return ask.forSplit(l, r);
}
}
```

But isn't `bTreeVisitorT` 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**.

<sup>12</sup> Okay.

---

How many methods does the definition of `bIsFlatV` contain, assuming it implements `bTreeVisitorI`?

<sup>13</sup> Three, because it works with `TreeD`s, and the datatype definition for `TreeD`s has three variants.

---

What type of values do the methods of `bIsFlatV` produce?

<sup>14</sup> `booleans`.

---

What visitor does `bIsFlatV` remind us of?

<sup>15</sup> `OnlyOnionsV`.

---

Here is a skeleton for `bIsFlatV`.

```
class bIsFlatV implements bTreeVisitorI {
    public
        boolean forBud() {
            return _____; }
    public
        boolean forFlat(FruitD f,TreeD t) {
            return _____; }
    public
        boolean forSplit(TreeD l,TreeD r) {
            return _____; }
}
```

<sup>16</sup> That's easy now.

```
class bIsFlatV implements bTreeVisitorI {
    public
        boolean forBud() {
            return true; }
    public
        boolean forFlat(FruitD f,TreeD t) {
            return t.accept(this); }
    public
        boolean forSplit(TreeD l,TreeD r) {
            return false; }
}
```

Fill in the blanks.

---

Define the `bIsSplitV` visitor, whose methods check whether a `TreeD` is constructed with Split and Bud only.

<sup>17</sup> Here is the easy part.

```
class bIsSplitV implements bTreeVisitorI {
    public
        boolean forBud() {
            return true; }
    public
        boolean forFlat(FruitD f,TreeD t) {
            return false; }
    public
        boolean forSplit(TreeD l,TreeD r) {
            _____ }
}
```

---

What is difficult about the last line?

<sup>18</sup> We need to check whether both  $l$  and  $r$  are split trees.

---

Isn't that easy?

<sup>19</sup> Yes, we just use the methods of  $\text{blsSplit}^V$  on  $l$  and  $r$ .

---

And then?

<sup>20</sup> Then we need to know that both are true.

---

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

<sup>21</sup> Yes, because if both are true, we have a split tree.

---

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

<sup>22</sup> No, then the answer is **false**.

---

Finish the definition of  $\text{blsSplit}^V$  using

```
if (...)  
  return ...  
else  
  return ....
```

<sup>23</sup> Now we can do it.

```
class blsSplitV implements bTreeVisitorT {  
  public  
    boolean forBud() {  
      return true; }  
  public  
    boolean forFlat(FruitD f, TreeD t) {  
      return false; }  
  public  
    boolean forSplit(TreeD l, TreeD r) {  
      if1 (l.accept(this))  
        return r.accept(this);  
      else  
        return false; }  
}
```

<sup>1</sup> We could have written the if ... as  
   $\text{return } l.\text{accept}(\text{this}) \&\& r.\text{accept}(\text{this}).$

---

---

Give an example of a  $\text{Tree}^D$  for which the methods of  $\text{bIsSplit}^V$  respond with true.

<sup>24</sup> There is a trivial one:  
**new Bud()**.

---

How about one with five uses of Split?

<sup>25</sup> Here is one:  
**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?

<sup>26</sup> No.

---

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

<sup>27</sup> Here it is.

```
class bHasFruitV
    implements bTreeVisitorT {
public
    boolean forBud() {
        return false;
    }
public
    boolean forFlat(FruitD f,TreeD t) {
        return true;
    }
public
    boolean forSplit(TreeD l,TreeD r) {
        if1 (l.accept(this))
            return true;
        else
            return r.accept(this);
    }
}
```

---

<sup>1</sup> We could have written the if ... as  
`return l.accept(this) || r.accept(this)`

---

---

What is the height of

<sup>28</sup> 3.

```
new Split(  
  new Split(  
    new Bud(),  
    new Flat(new Lemon(),  
      new Bud()),  
    new Flat(new Fig(),  
      new Split(  
        new Bud(),  
        new Bud()))))?
```

---

What is the height of

<sup>29</sup> 2.

```
new Split(  
  new Bud(),  
  new Flat(new Lemon(),  
    new Bud()))?
```

---

What is the height of

<sup>30</sup> 1.

```
new Flat(new Lemon(),  
  new Bud())?
```

---

What is the height of

<sup>31</sup> 0.

```
new Bud()?
```

---

So what is the height of a  $\text{Tree}^{\mathcal{D}}$ ?

<sup>32</sup> 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  $i\text{Height}^{\mathcal{V}}$  work on a  $\text{Tree}^{\mathcal{D}}$ ? <sup>33</sup> Yes, and they produce an **int**.

---

Is that what the **i** in front of **Height** is all about?

<sup>34</sup> It looks like **i** stands for **int**, doesn't it?

---

---

What is the value of  
new Split(  
    new Split(  
        new Bud(),  
        new Bud()),  
    new Flat(new Fig(),  
        new Flat(new Lemon(),  
            new Flat(new Apple(),  
                new Bud()))))  
.accept(new iHeight<sup>V</sup>())?

---

Why is the height 4?

<sup>35</sup> 4.

<sup>36</sup> Because the value of  
new Split(  
    new Bud(),  
    new Bud())  
.accept(new iHeight<sup>V</sup>())  
is 1; the value of  
new Flat(new Fig(),  
    new Flat(new Lemon(),  
        new Flat(new Apple(),  
            new Bud()))))  
.accept(new iHeight<sup>V</sup>())  
is 3; and the larger of the two numbers is 3.

---

And how do we get from 3 to 4?

<sup>37</sup> We need to add one to the larger of the numbers so that we don't forget that the original Tree<sup>D</sup> was constructed with Split and those two Tree<sup>D</sup>s.

---

□ picks the larger of two numbers,  $x$  and  $y$ .<sup>1</sup>

<sup>38</sup> Oh, that's nice. What kind of methods does iHeight<sup>V</sup> define?

<sup>1</sup> When you enter this in a file, use  
Math.max(x,y).  
Math is a class that contains max as a (static) method.

---

iHeight<sup>V</sup>'s methods measure the heights of the Tree<sup>D</sup>s to which they correspond.

<sup>39</sup> Now that's a problem.

---

---

Why?

<sup>40</sup> We defined only **interfaces** that produce **booleans** in this chapter.

---

So what?

<sup>41</sup> The methods of  $i\text{Height}^V$  produce **ints**, which are not **booleans**.

---

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

<sup>42</sup> It's almost the same as  $b\text{TreeVisitor}^T$ .

```
interface iTreeVisitorT {  
    int forBud();  
    int forFlat(FruitD f, TreeD t);  
    int forSplit(TreeD l, TreeD r);  
}
```

---

Yes, and once we have that we can add another *accept* method to  $\text{Tree}^D$ .

```
abstract class TreeD {  
    abstract  
    boolean accept(bTreeVisitorT ask);  
    abstract  
    int accept(iTreeVisitorT ask);  
}
```

---

<sup>1</sup> 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.

<sup>44</sup>  $b\text{TreeVisitor}^T$  is indeed different from  $i\text{TreeVisitor}^T$ , so we can have two versions of *accept* in  $\text{Tree}^D$ .

---

Add the new *accept* methods to  $\text{Tree}^D$ 's variants. Start with the easy one.

<sup>45</sup> It is easy.

```
class Bud extends TreeD {  
    boolean accept(bTreeVisitorT ask) {  
        return ask.forBud(); }  
    int accept(iTreeVisitorT ask) {  
        return ask.forBud(); }  
}
```

---

---

The others are easy, too. We duplicate *accept*.

```
class Flat extends TreeD {  
    FruitD f;  
    TreeD t;  
    Flat(FruitD _f,TreeD _t) {  
        f = _f;  
        t = _t; }  
  
    boolean accept(bTreeVisitorT ask) {  
        return ask.forFlat(f,t); }  
    int accept(iTreeVisitorT ask) {  
        return ask.forFlat(f,t); }  
}
```

<sup>46</sup> We must also change the type of what the new *accept* method consumes and produces.

```
class Split extends TreeD {  
    TreeD l;  
    TreeD r;  
    Split(TreeD _l,TreeD _r) {  
        l = _l;  
        r = _r; }  
  
    boolean accept(bTreeVisitorT ask) {  
        return ask.forSplit(l,r); }  
    int accept(iTreeVisitorT ask) {  
        return ask.forSplit(l,r); }  
}
```

---

Here is *iHeight<sup>V</sup>*.

```
class iHeightV implements iTreeVisitorT {  
    public int forBud() {  
        return _____; }  
    public int forFlat(FruitD f,TreeD t) {  
        return _____; }  
    public int forSplit(TreeD l,TreeD r) {  
        return _____; }  
}
```

Complete these methods.

<sup>47</sup> That's easy now.

```
class iHeightV implements iTreeVisitorT {  
    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; }  
}
```

---

What is the value of

```
new Split(  
    new Bud(),  
    new Bud())  
.accept(new iHeightV())?
```

<sup>48</sup> 1, of course.

---

And why is it 1?

<sup>49</sup> Because

```
new Bud().accept(new iHeightV())  
is 0, the larger of 0 and 0 is 0, and one more  
is 1.
```

---

What is the value of

```

new Split(
    new Split(
        new Flat(new Fig(),
            new Bud()),
        new Flat(new Fig(),
            new Bud())),
    new Flat(new Fig(),
        new Flat(new Lemon(),
            new Flat(new Apple(),
                new Bud()))))
.accept(
    new tSubstV(
        new Apple(),
        new Fig())?)
```

---

Correct. Define the  $tSubst^V$  visitor.

What's the problem?

<sup>50</sup> If the visitor  $tSubst^V$  substitutes apples for figs, here is what we get:

```

new Split(
    new Split(
        new Flat(new Apple(),
            new Bud()),
        new Flat(new Apple(),
            new Bud())),
    new Flat(new Apple(),
        new Flat(new Lemon(),
            new Flat(new Apple(),
                new Bud())))).
```

<sup>51</sup> It's like  $SubstFish^V$  and  $SubstInt^V$  from the end of chapter 5, but we can't do it just yet.

Good job. How about the datatype  $\text{Tree}^D$ .

<sup>52</sup> Easy. Here is the abstract one.

```

abstract class TreeD {
abstract
    boolean accept(bTreeVisitorI ask);
abstract
    int accept(iTreeVisitorI ask);
abstract
    TreeD accept(tTreeVisitorI ask);
}
```

---

---

Define the variants of  $\text{Tree}^D$ .

<sup>54</sup> No problem.

```
class Bud extends TreeD {
    boolean accept(bTreeVisitorI ask) {
        return ask.forBud(); }
    int accept(iTreeVisitorI ask) {
        return ask.forBud(); }
    TreeD accept(tTreeVisitorI ask) {
        return ask.forBud(); }
}
```

```
class Flat extends TreeD {
    FruitD f;
    TreeD t;
    Flat(FruitD _f,TreeD _t) {
        f = _f;
        t = _t; }
```

```
boolean accept(bTreeVisitorI ask) {
    return ask.forFlat(f,t); }
int accept(iTreeVisitorI ask) {
    return ask.forFlat(f,t); }
TreeD accept(tTreeVisitorI 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(bTreeVisitorI ask) {
    return ask.forSplit(l,r); }
int accept(iTreeVisitorI ask) {
    return ask.forSplit(l,r); }
TreeD accept(tTreeVisitorI ask) {
    return ask.forSplit(l,r); }
```

---

Then define  $tSubst^V$ .

<sup>55</sup> 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.

```
class tSubstV implements tTreeVisitorT {
    FruitD n;
    FruitD o;
    tSubstV(FruitD _n,FruitD _o) {
        n = _n;
        o = _o;
    }

    public TreeD forBud() {
        return new Bud();
    }
    public TreeD forFlat(FruitD f,TreeD t) {
        if (o.equals(f))
            return new Flat(n,t.accept(this));
        else
            return new Flat(f,t.accept(this));
    }
    public TreeD forSplit(TreeD l,TreeD r) {
        return new Split(l.accept(this),
                         r.accept(this));
    }
}
```

---

Here is a  $Tree^D$  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  $iOccurs^V$ , whose methods count how often some  $Fruit^D$  occurs in a tree.

<sup>56</sup> Even the visitors are no longer interesting.

```
class iOccursV implements iTreeVisitorT {
    FruitD a;
    iOccursV(FruitD _a) {
        a = _a;
    }

    public int forBud() {
        return 0;
    }
    public int forFlat(FruitD f,TreeD t) {
        if (f.equals(a))
            return t.accept(this) + 1;
        else
            return t.accept(this);
    }
    public int forSplit(TreeD l,TreeD r) {
        return
            l.accept(this) + r.accept(this);
    }
}
```

---

---

Do you like your fruit with yogurt?

<sup>57</sup> We prefer coconut sorbet.

---

Is it disturbing that we have three nearly identical versions of *accept* in *Tree<sup>D</sup>*'s and its variants?

<sup>58</sup> 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?

<sup>59</sup> If **boolean** and **int** were classes, we could use **Object** for **boolean**, **int**, and *Tree<sup>D</sup>*. Unfortunately, they are not.

---

Remember **Integer** and **Boolean**? They make it possible.

<sup>60</sup> 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<sup>D</sup>*.

```
interface TreeVisitorI {
    Object forBud();
    Object forFlat(FruitD f, TreeD t);
    Object forSplit(TreeD l, TreeD r);
}
```

Here is the datatype and the Bud variant.

```
abstract class TreeD {
    abstract
        Object accept(TreeVisitorI ask);
}
```

```
class Bud extends TreeD {
    Object accept(TreeVisitorI ask) {
        return ask.forBud();
    }
}
```

Define the remaining variants of *Tree<sup>D</sup>*.

<sup>61</sup> Here they are.

```
class Flat extends TreeD {
    FruitD f;
    TreeD t;
    Flat(FruitD _f, TreeD _t) {
        f = _f;
        t = _t;
    }
}
```

```
Object accept(TreeVisitorI ask) {
    return ask.forFlat(f, t);
}
```

```
class Split extends TreeD {
    TreeD l;
    TreeD r;
    Split(TreeD _l, TreeD _r) {
        l = _l;
        r = _r;
    }
}
```

```
Object accept(TreeVisitorI ask) {
    return ask.forSplit(l, r);
}
```

---

Good. Now define `IsFlatV`, an Object producing version of `blsFlatV`.

<sup>62</sup> That's no big deal.

```
class IsFlatV implements TreeVisitorT {
    public Object forBud() {
        return new Boolean(true); }
    public Object forFlat(FruitD f, TreeD t) {
        return t.accept(this); }
    public Object forSplit(TreeD l, TreeD r) {
        return new Boolean(false); }
}
```

---

And how about `IsSplitV`?

<sup>63</sup> 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.

```
class IsSplitV implements TreeVisitorT {
    public Object forBud() {
        return new Boolean(true); }
    public Object forFlat(FruitD f, TreeD t) {
        return new Boolean(false); }
    public Object forSplit(TreeD l, TreeD r) {
        if (((Boolean) (l.accept(this)))
            .booleanValue())
            return r.accept(this);
        else
            return new Boolean(false); }
}
```

---

<sup>64</sup> Oh, because `l.accept(this)` produces an Object, we must first convert<sup>1</sup> 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`.

<sup>1</sup> 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).

---

Will the conversion always work?

<sup>65</sup> 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.*

---

Did you think that was bad? Then study  
this definition during your next break.

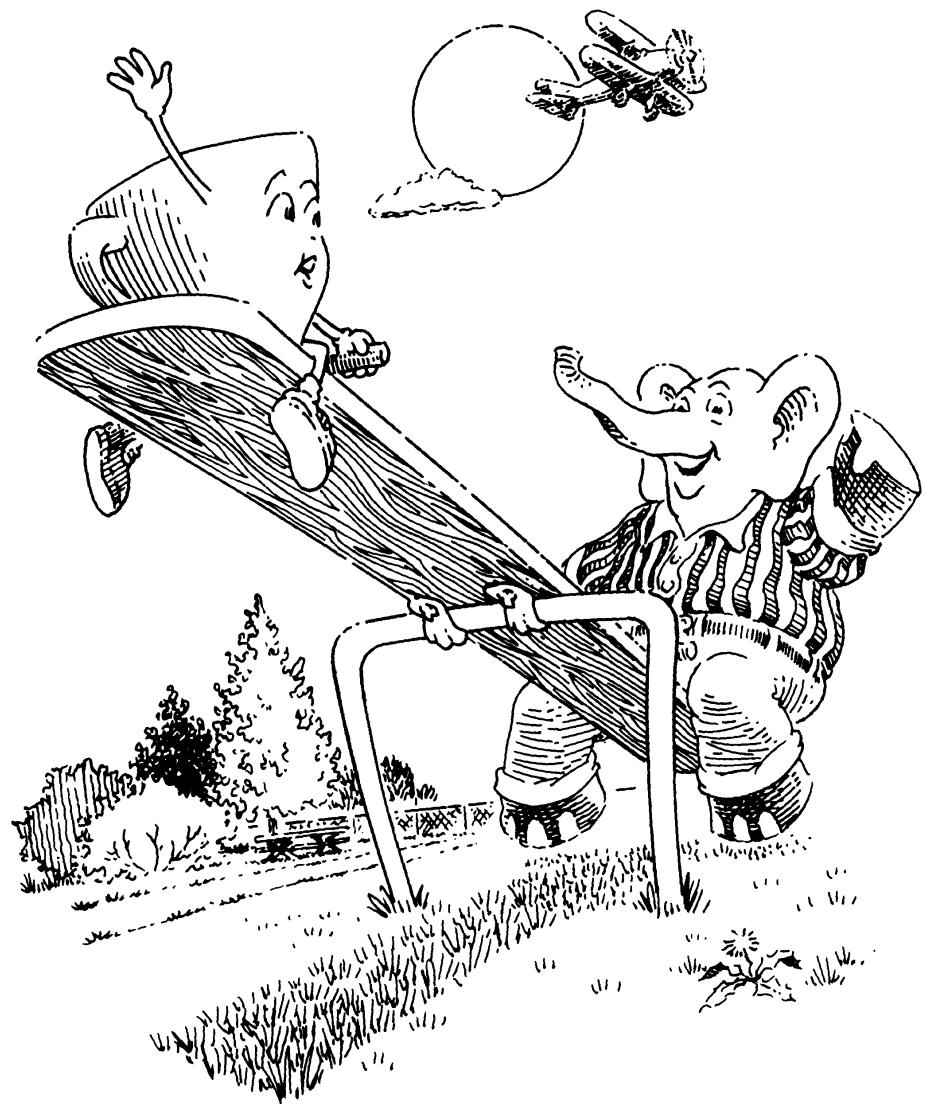
<sup>66</sup> Oh my!

```
class OccursV implements TreeVisitorZ {
    FruitD a;
    OccursV(FruitD _a) {
        a = _a; }

    public Object forBud() {
        return new Integer(0); }
    public Object forFlat(FruitD f,TreeD t) {
        if (f.equals(a))
            return
                new Integer(((Integer)
                    (t.accept(this)))
                    .intValue()
                    + 1);
        else
            return t.accept(this); }
    public Object forSplit(TreeD l,TreeD r) {
        return
            new Integer(((Integer)
                (l.accept(this)))
                .intValue()
                +
                ((Integer)
                    (r.accept(this)))
                    .intValue()); }
}
```

---

80  
Like Father,  
Like Son



---

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

---

<sup>1</sup> 12.

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

---

<sup>2</sup> 12,

because we have just rewritten the previous expression with prefix operators.

---

What is the value of  
`new Plus(  
 new Const(new Integer(7)),  
 new Prod(  
 new Diff(  
 new Const(new Integer(4)),  
 new Const(new Integer(3))),  
 new Const(new Integer(5))))?`

---

<sup>3</sup> `new Integer(12)`,

because we have just rewritten the previous expression using `Integer` and constructors.

---

Where do the constructors come from?

---

<sup>4</sup> A datatype and its variants that represent arithmetic expressions.

---

Did you like that?

---

<sup>5</sup> So far, so good.

---

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

---

<sup>6</sup>  $\{7, 5\}$ .

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

---

<sup>7</sup>  $\{7, 5\}$ ,

we just went from infix to prefix notation.

---

What is the value of  
 $(+ \{7, 5\} (\times (- \{4\} \{3\}) \{5\}))?$

---

<sup>8</sup>  $\{7, 5\}$ ,

we just renamed the operators.

---

---

What is the value of

```
new Plus(  
  new Const(new Empty()  
    .add(new Integer(7))  
    .add(new Integer(5))),  
  new Prod(  
    new Diff(  
      new Const(new Empty()  
        .add(new Integer(4))),  
      new Const(new Empty()  
        .add(new Integer(3)))),  
    new Const(new Empty()  
      .add(new Integer(5))))?
```

<sup>9</sup> `new Empty()`

```
.add(new Integer(7))  
.add(new Integer(5)),  
because we have just rewritten the  
previous expression using the constructors.
```

Where do the constructors come from?

<sup>10</sup> A datatype and its variants that represent set expressions.

---

Do you still like it?

<sup>11</sup> Sure, why not.

---

Does the arithmetic expression look like the set expression?

<sup>12</sup> Yes, they look the same except for the constants:

```
new Plus(  
  new Const(•),  
  new Prod(  
    new Diff(  
      new Const(•),  
      new Const(•)),  
    new Const(•))).
```

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?

<sup>13</sup> That's a tricky question.

```
interface ExprVisitorI {  
  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.

14

```
abstract class ExprD {  
    abstract  
        Object accept(ExprVisitorT ask);  
}
```

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

```
class Plus extends ExprD {  
    ExprD l;  
    ExprD r;  
    Plus(ExprD _l,ExprD _r) {  
        l = _l;  
        r = _r; }  
}
```

```
Object accept(ExprVisitorT ask) {  
    return ask.forPlus(l,r); }  
}
```

```
class Diff extends ExprD {  
    ExprD l;  
    ExprD r;  
    Diff(ExprD _l,ExprD _r) {  
        l = _l;  
        r = _r; }  
}
```

```
Object accept(ExprVisitorT ask) {  
    return ask.forDiff(l,r); }  
}
```

```
class Prod extends ExprD {  
    ExprD l;  
    ExprD r;  
    Prod(ExprD _l,ExprD _r) {  
        l = _l;  
        r = _r; }  
}
```

```
Object accept(ExprVisitorT ask) {  
    return ask.forProd(l,r); }  
}
```

```
class Const extends ExprD {  
    Object c;  
    Const(Object _c) {  
        c = _c; }  
}
```

```
Object accept(ExprVisitorT ask) {  
    return ask.forConst(c); }  
}
```

---

Can we now define a visitor whose methods determine the value of an arithmetic expression?

<sup>15</sup> Yes, we can. It must have four methods, one per variant, and it is like  $\text{Occurs}^V$  from the previous chapter.

---

How do we add

**new Integer(3)**  
and  
**new Integer(2)?**

<sup>16</sup> We have done this before. We use the method *intValue* to determine the **ints** that correspond to the **Integers**, and then add them together.

---

But what is the result of

**new Integer(3).intValue()**  
+  
**new Integer(2).intValue()?**

<sup>17</sup> An **int**, what else?

---

How do we turn that into an **Integer**?

<sup>18</sup> We use **new Integer(...)**.

---

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

```
class IntEvalV implements ExprVisitorT {
    public Object forPlus(ExprD l, ExprD r) {
        return plus(l.accept(this),
                    r.accept(this));
    }
    public Object forDiff(ExprD l, ExprD r) {
        return diff(l.accept(this),
                    r.accept(this));
    }
    public Object forProd(ExprD l, ExprD r) {
        return prod(l.accept(this),
                    r.accept(this));
    }
    public Object forConst(Object c) {
        return c;
    }
    Object plus(__1 l, __2 r) {
        return __3;
    }
    Object diff(__1 l, __2 r) {
        return __4;
    }
    Object prod(__1 l, __2 r) {
        return __5;
    }
}
```

---

<sup>19</sup> 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?

<sup>20</sup> It consumes two  $\text{Expr}^{\mathcal{D}}$ s, determines their respective values, and *pluses* them.

---

How are the values represented?

<sup>21</sup> As **Objects**, because we are using our most general kind of (and most recent) visitor.

---

So what kind of values must *plus* consume?

<sup>22</sup> **Objects**,  
because that's what  
*l.accept(this)*  
and  
*r.accept(this)*  
produce.

---

What must we put in the first and second blanks?

<sup>23</sup> **Object**.

---

Can we add **Objects**?

<sup>24</sup> No, we must convert them to **Integers** first and extract their underlying **ints**.

---

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

<sup>25</sup> No, but all **Objects** produced by  $\text{IntEval}^{\mathcal{V}}$  are made with **new Integer(...)**, so that this conversion always succeeds.

---

Is that true? What is the value of  
**new Plus(**  
    **new Const(new Empty()),**  
    **new Const(new Integer(5)))**  
.accept(**new IntEval**<sup>V</sup>())?

<sup>26</sup> Wow. At some level, this is nonsense.

---

Correct, so sometimes the conversion may fail, because we use an instance of  $\text{IntEval}^{\mathcal{V}}$  on nonsensical arithmetic expressions.

<sup>27</sup> What should we do?

---

We agree to avoid such arithmetic expressions.<sup>1</sup>

<sup>28</sup> And their set expressions, too.

<sup>1</sup> 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 [1: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

```
new Integer(  
    ((Integer)l).intValue()  
    +  
    ((Integer)r).intValue());
```

Complete the definition now.

<sup>29</sup> Now it's easy. Here we go.

```
class IntEvalV implements ExprVisitorT {  
    public Object forPlus(ExprD l,ExprD r) {  
        return plus(l.accept(this),  
                    r.accept(this)); }  
    public Object forDiff(ExprD l,ExprD r) {  
        return diff(l.accept(this),  
                    r.accept(this)); }  
    public Object forProd(ExprD l,ExprD 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()); }  
}
```

---

That one was pretty easy, wasn't it?

<sup>30</sup> Yes. Let's implement an `ExprVisitorT` for sets.

---

What do we need to implement one for sets? <sup>31</sup> We certainly need methods for *plusing*, *diffing*, and *prodig* sets.

---

That's correct, and here is everything.

<sup>32</sup> Whoa.

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

---

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

<sup>33</sup> 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?

<sup>34</sup> Constructors always construct, and `add` does not always construct.

---

Do we need to understand that?

<sup>35</sup> Not now, but feel free to absorb it when you have the time.

---

---

Define the variants `Empty` and `Add` for  $\text{Set}^D$ .<sup>36</sup> Here we go.

```
class Empty extends SetD {
    boolean mem(Integer i) {
        return false;
    }
    SetD plus(SetD s) {
        return s;
    }
    SetD diff(SetD s) {
        return new Empty();
    }
    SetD prod(SetD s) {
        return new Empty();
    }
}
```

```
class Add extends SetD {
    Integer i;
    SetD s;
    Add(Integer _i, SetD _s) {
        i = _i;
        s = _s;
    }

    boolean mem(Integer n) {
        if (i.equals(n))
            return true;
        else
            return s.mem(n);
    }
    SetD plus(SetD t) {
        return s.plus(t.add(i));
    }
    SetD diff(SetD t) {
        if (t.mem(i))
            return s.diff(t);
        else
            return s.diff(t).add(i);
    }
    SetD prod(SetD t) {
        if (t.mem(i))
            return s.prod(t).add(i);
        else
            return s.prod(t);
    }
}
```

---

---

Do we need to understand these definitions?

<sup>37</sup> 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?

---

How should we do that?

<sup>38</sup> Oh, that's a piece of pie. We just copy the definition of  $\text{IntEval}^V$  and replace its *plus*, *diff*, and *prod* methods.

---

That's the worst way of doing that.

<sup>40</sup> What?

---

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

<sup>41</sup> That's true. If we copied the definition and changed it, we would have identical copies of *forPlus*, *forDiff*, *forProd*, and *forConst*. We should reuse this definition.<sup>1</sup>

<sup>1</sup> Sometimes we do not have license to see the definitions, so copying might not even be an option.

---

Yes, and we are about to show you better ways. How do we have to change *plus*, *diff*, and *prod*?

<sup>42</sup> That part is easy:

```
Object plus(Object l,Object r) {  
    return ((SetD)l).plus((SetD)r); }
```

and

```
Object diff(Object l,Object r) {  
    return ((SetD)l).diff((SetD)r); }
```

and

```
Object prod(Object l,Object r) {  
    return ((SetD)l).prod((SetD)r); }.
```

---

---

Very good, and if we define `SetEvalV` as an extension of `IntEvalV`, that's all we have to put inside of `SetEvalV`.

```
class SetEvalV extends IntEvalV {  
    Object plus(Object l, Object r) {  
        return ((SetD)l).plus((SetD)r); }  
    Object diff(Object l, Object r) {  
        return ((SetD)l).diff((SetD)r); }  
    Object prod(Object l, Object r) {  
        return ((SetD)l).prod((SetD)r); }  
}
```

---

<sup>43</sup> Now that's much easier than copying and modifying.

Is it like `equals`?

<sup>44</sup> Yes, when we include `equals` in our class definitions, we override the one in `Object`. Here, we override the methods `plus`, `diff`, and `prod` as we extend `IntEvalV`.

---

How many methods from `IntEvalV` are overridden in `SetEvalV`?

<sup>45</sup> Three.

How many methods from `IntEvalV` are not overridden in `SetEvalV`?

<sup>46</sup> Four: `forPlus`, `forDiff`, `forProd`, and `forConst`.

---

Does `SetEvalV` implement `ExprVisitorT`?

<sup>47</sup> It doesn't say so.

---

Does `SetEvalV` extend `IntEvalV`?

<sup>48</sup> It says so.

---

Does `IntEvalV` implement `ExprVisitorT`?

<sup>49</sup> It says so.

---

Does `SetEvalV` implement `ExprVisitorT`?

<sup>50</sup> By implication.

---

---

That's correct. What is the value of  
`new Prod(  
 new Const(new Empty()  
 .add(new Integer(7))),  
 new Const(new Empty()  
 .add(new Integer(3))))  
.accept(new SetEvalV())?`

---

<sup>51</sup> Interesting question. How does this work now?

What type of value is

`new Prod(  
 new Const(new Empty()  
 .add(new Integer(7))),  
 new Const(new Empty()  
 .add(new Integer(3))))?`

---

<sup>52</sup> It is a Prod and therefore an Expr<sup>D</sup>.

And what does *accept* consume?

<sup>53</sup> An instance of SetEval<sup>V</sup>, but its type is ExprVisitor<sup>T</sup>.

---

What is

`new SetEvalV().forProd(  
 new Const(new Empty()  
 .add(new Integer(7))),  
 new Const(new Empty()  
 .add(new Integer(3))))?`

---

<sup>54</sup> That's what we need to determine the value of next, because it is *ask.forProd(l,r)*, with *ask*, *l*, and *r* replaced by what they stand for.

Where is the definition of SetEval<sup>V</sup>'s method  
*forProd*?

---

<sup>55</sup> It is in IntEval<sup>V</sup>.

Suppose we had the values of

`new Const(new Empty()  
 .add(new Integer(7)))  
.accept(this)`

and

`new Const(new Empty()  
 .add(new Integer(3)))  
.accept(this).`

What would we have to evaluate next?

---

<sup>56</sup> If their values were *A* and *B*, we would have to determine the value of *prod(A,B)*.

---

Isn't that strange?

<sup>57</sup> Why?

---

So far, we have always used a method on a particular object.

<sup>58</sup> That's true. What is the object with which we use *prod(A,B)*?

---

It is **this** object.

<sup>59</sup> Oh, does that mean we should evaluate  
**new SetEval<sup>V</sup>()**.*prod(A,B)*?

---

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

<sup>60</sup> That clarifies things.

---

Good. And now what?

<sup>61</sup> Now we still need to determine the values of  
**new Const(new Empty())**  
.add(**new Integer(7)**)  
.accept(**this**)  
and  
**new Const(new Empty())**  
.add(**new Integer(3)**)  
.accept(**this**).

---

The values are obviously

<sup>62</sup> It, too, is in **IntEval<sup>V</sup>**.

**new Empty()**  
.add(**new Integer(7)**)

and

**new Empty()**  
.add(**new Integer(3)**).

Where is the definition of *forConst* that determines these values?

---

Here is the next expression in our sequence:

<sup>63</sup> The object is an instance of **SetEval<sup>V</sup>**, which overrides the *prod* method in **IntEval<sup>V</sup>** with its own.

**new SetEval<sup>V</sup>()**  
.prod(**new Empty()**  
.add(**new Integer(7)**),  
**new Empty()**  
.add(**new Integer(3)**)).

Where does *prod* come from?

---

---

What next?

<sup>64</sup> Next we need to determine the value of

$((\text{Set}^D)(\text{new Empty}()\text{.add}(\text{new Integer}(7))))\text{.prod}((\text{Set}^D)\text{new Empty}()\text{.add}(\text{new Integer}(3)))$ ,

because it is

$((\text{Set}^D)(l.\text{accept}(\text{this})))\text{.prod}((\text{Set}^D)r.\text{accept}(\text{this}))$

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

---

Is

$\text{new Empty}().\text{add}(\text{new Integer}(7))$   
an instance of  $\text{Set}^D$ ?

<sup>65</sup> Of course it is, but the type of  $l.\text{accept}(\text{this})$ , which is where it comes from, is `Object`.

---

And how about

$\text{new Empty}().\text{add}(\text{new Integer}(3))$ ?

<sup>66</sup> It's the same.

---

And that is why the method must contain a conversion from `Object` to  $\text{Set}^D$ s.

<sup>67</sup> This example makes the need for conversions obvious again.

---

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

<sup>68</sup> This one belongs to  $\text{Set}^D$  or more precisely its `Empty` and `Add` variants.

---

And what does  $\text{prod}$  do?

<sup>69</sup> It determines the intersection of one  $\text{Set}^D$  with another  $\text{Set}^D$ , but didn't we agree that the previous question was the last question on that topic?

---

We overrode that, too.

<sup>70</sup> Thanks, guys.

---

Is it natural that  $\text{SetEval}^V$  extends  $\text{IntEval}^V$ ? <sup>71</sup> No, not at all.

---

---

Why did we do that?

<sup>72</sup> Because we defined `IntEvalV` first.<sup>1</sup>

<sup>1</sup> 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.

<sup>73</sup> Yes, let's do better. We have defined all these classes ourselves, so we are free to rearrange them any way we want.

---

What distinguishes `IntEvalV` from `SetEvalV`?

<sup>74</sup> The methods `plus`, `diff`, and `prod`.

---

What are the pieces that they have in common?

<sup>75</sup> They share the methods `forPlus`, `forDiff`, `forProd`, and `forConst`.

---

Good. Here is how we express that.

<sup>76</sup> Isn't this abstract class like `PointD`?

```
abstract class EvalD
    implements ExprVisitorT {
    public Object forPlus(ExprD l,ExprD r) {
        return plus(l.accept(this),
                    r.accept(this));
    }
    public Object forDiff(ExprD l,ExprD r) {
        return diff(l.accept(this),
                    r.accept(this));
    }
    public Object forProd(ExprD l,ExprD 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  $\text{Eval}^D$ .

```
class IntEvalV extends EvalD {
    Object plus(Object l, Object r) {
        return
            new Integer(
                ((Integer)l).intValue()
                +
                ((Integer)r).intValue()); }
    Object diff(Object l, Object r) {
        return
            new Integer(
                ((Integer)l).intValue()
                -
                ((Integer)r).intValue()); }
    Object prod(Object l, Object r) {
        return
            new Integer(
                ((Integer)l).intValue()
                *
                ((Integer)r).intValue()); }
}
```

<sup>77</sup> What do we do now?

<sup>78</sup> It is basically like the original but extends  $\text{Eval}^D$ , not  $\text{IntEval}^V$ .

```
class SetEvalV extends EvalD {
    Object plus(Object l, Object r) {
        return ((SetD)l).plus((SetD)r); }
    Object diff(Object l, Object r) {
        return ((SetD)l).diff((SetD)r); }
    Object prod(Object l, Object r) {
        return ((SetD)l).prod((SetD)r); }
}
```

Define  $\text{SetEval}^V$ .

---

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

Time for supper.

<sup>79</sup> Much more so than one extending the other.

<sup>80</sup> If you are neither hungry nor tired, you may continue.

---

Remember  $\text{Subst}^V$  from chapter 6?

<sup>81</sup> Yes, and  $\text{LtdSubst}^V$ , too.

```
class SubstV implements PieVisitorT {
    Object n;
    Object o;
    SubstV(Object _n, Object _o) {
        n = _n;
        o = _o; }

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

```
class LtdSubstV implements PieVisitorT {
    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(
                            new LtdSubstV(c - 1, n, o)));
            else
                return
                    new Top(t, r.accept(this)); }
```

---

What do the two visitors have in common?

<sup>82</sup> Many things:  $n$ ,  $o$ , and *forBot*.

---

Where do they differ?

<sup>83</sup> They differ in *forTop*, but  $\text{LtdSubst}^V$  also has an extra field.

---

And where do we put the pieces that two classes have in common?

<sup>84</sup> We put them into an abstract class.

---

What else does the abstract class contain?

<sup>85</sup> It specifies the pieces that are different if they are needed for all extensions.

---

Define the **abstract** class `SubstD`, which contains all the common pieces and specifies what a concrete pie substituter must contain in addition.

<sup>86</sup> It's not a big deal, except for the fields.

```
abstract class SubstD
    implements PieVisitorI {
    Object n;
    Object o;
    public PieD forBot() {
        return new Bot(); }
    public
        abstract PieD forTop(Object t,PieD r);
    }
```

---

We can define `SubstV` by extending `SubstD`.

```
class SubstV extends SubstD {
    SubstV(Object _n, Object _o) {
        n = _n;
        o = _o; }

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

Define `LtdSubstV`.

<sup>87</sup> It also extends `SubstD`.

```
class LtdSubstV extends SubstD {
    int c;
    LtdSubstV(int _c, Object _n, Object _o) {
        n = _n;
        o = _o;
        c = _c; }

    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(
                            new LtdSubstV(c - 1,n,o)));
            else
                return
                    new Top(t,r.accept(this)); }
}
```

---

Do the two remaining classes still have things in common?

<sup>88</sup> No, but the constructors have some overlap. Shouldn't we lift the `SubstV` constructor into `SubstD`, because it holds the common elements?

---

That's a great idea. Here is the new version of  $\text{Subst}^D$ .

```
abstract class SubstD
    implements PieVisitorI {
    Object n;
    Object o;
    SubstD(Object _n, Object _o) {
        n = _n;
        o = _o; }

    public PieD forBot() {
        return new Bot(); }
    public
        abstract PieD forTop(Object t, PieD r);
    }
```

Revise  $\text{Subst}^V$  and  $\text{LtdSubst}^V$ .

<sup>89</sup> We must use **super** in the constructors.

```
class SubstV extends SubstD {
    SubstV(Object _n, Object _o) {
        super(_n, _o); }

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

```
class LtdSubstV extends SubstD {
    int c;
    LtdSubstV(int _c, Object _n, Object _o) {
        super(_n, _o);
        c = _c; }

    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(
                            new LtdSubstV(c - 1, n, o)));
            else
                return
                    new Top(t, r.accept(this)); }
    }
```

---

Was that first part easy?

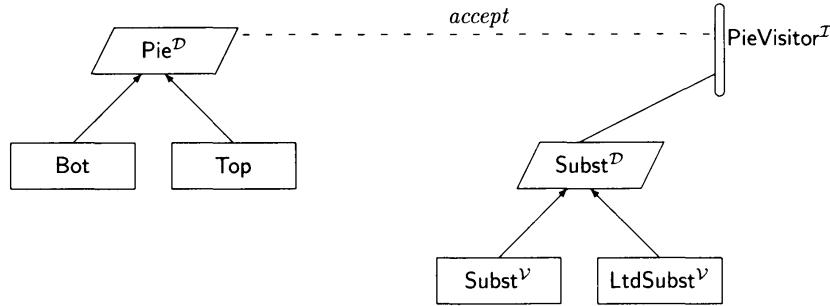
<sup>90</sup> As pie.

---

---

That's neat. How about some art work?

<sup>91</sup> Is this called a pie chart?



---

No, but the picture captures the important <sup>92</sup> Fine.  
relationships.

---

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

<sup>93</sup> It may even be better. In some sense,  $\text{LtdSubst}^V$  just adds a service to  $\text{Subst}^V$ : It counts as it substitutes.

---

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

<sup>94</sup> 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.*

---

Here is the good old definition of  $\text{Subst}^V$  from chapter 6 one more time.

```
class SubstV implements PieVisitorI {
    Object n;
    Object o;
    SubstV(Object _n, Object _o) {
        n = _n;
        o = _o; }

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

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

<sup>95</sup> The rest follows naturally, just as with the evaluators and the previous version of these two classes.

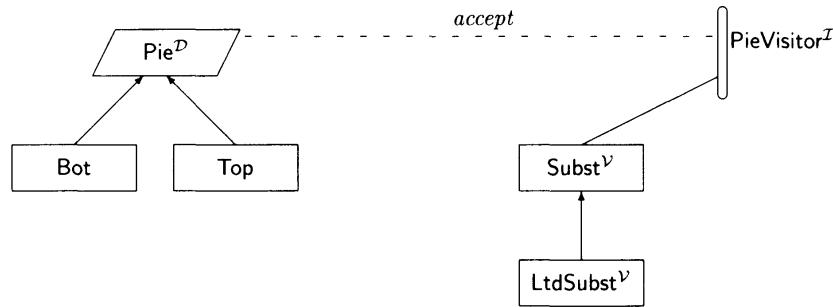
```
class LtdSubstV extends SubstV {
    int c;
    LtdSubstV(int _c, Object _n, Object _o) {
        super(_n, _o);
        c = _c; }

    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(
                            new LtdSubstV(c - 1, n, o)));
            else
                return
                    new Top(t, r.accept(this)); }
```

---

Let's draw a picture.

<sup>96</sup> Fine, and don't forget to use lines, rather than arrows, for **implements**.



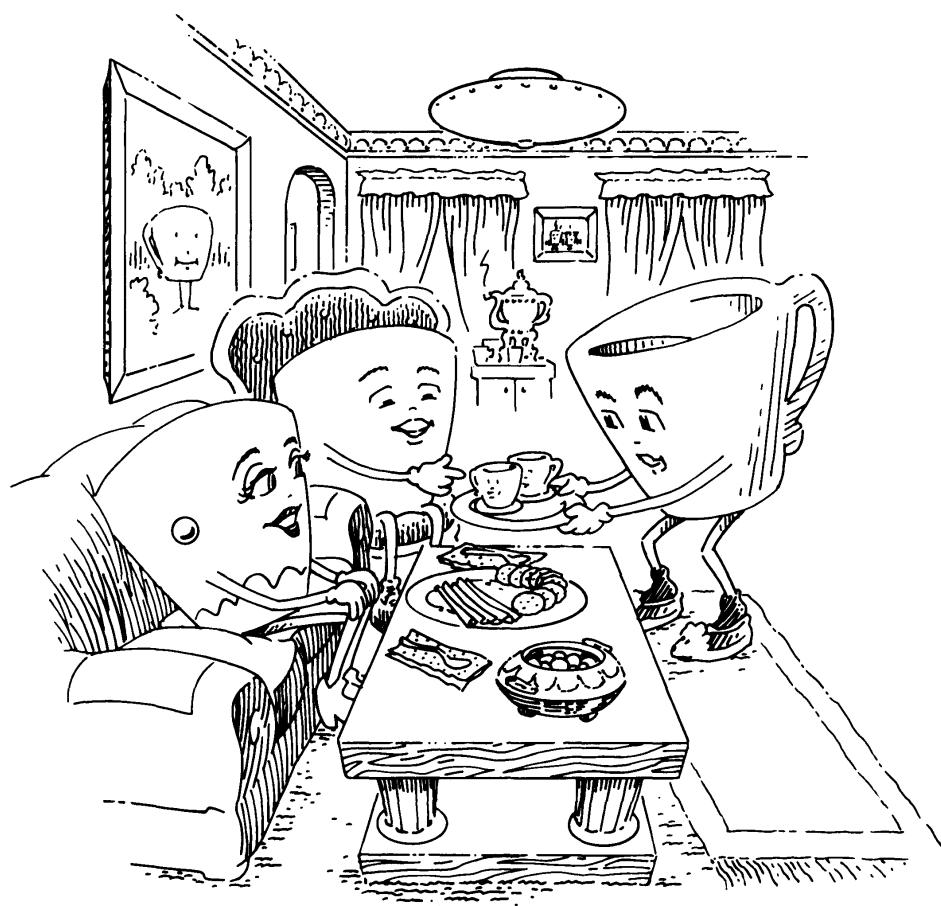
---

You deserve a super-deluxe pizza now.

<sup>97</sup> It's already on its way.

---

9.  
Be a Good Visitor



Remember `PointD`? 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 `PointD`'s variants.

```
abstract class PointD {
    int x;
    int y;
    PointD(int _x,int _y) {
        x = _x;
        y = _y;
    }

    boolean closerToO(PointD p) {
        return
            distanceToO() ≤ p.distanceToO();
    }
    PointD minus(PointD p) {
        return
            new CartesianPt(x - p.x,y - p.y);
    }
    abstract int distanceToO();
}
```

<sup>1</sup> It has been a long time since we discussed the datatype `PointD` and its variants, but they are not that easy to forget.

```
class CartesianPt extends PointD {
    CartesianPt(int _x,int _y) {
        super(_x,_y);
    }

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

```
class ManhattanPt extends PointD {
    ManhattanPt(int _x,int _y) {
        super(_x,_y);
    }

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

Good. Take a look at this extension of `ManhattanPt`.

```
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;
    }
}
```

<sup>2</sup> It uses  
 $\Delta_x = \lceil x \rceil$ ;  
 $\Delta_y = \lceil y \rceil$ ;  
in addition to `super(_x,_y)`.

What is unusual about the constructor?

---

And what does that mean?

<sup>3</sup> 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**?

<sup>4</sup> It is a **ManhattanPt** with two additional fields:  $\Delta_x$  and  $\Delta_y$ . These two represent the information that determines how far the shadow is from the point with the fields *x* and *y*.

---

Is this a **ShadowedManhattanPt**:

`new ShadowedManhattanPt(2,3,1,0)`?

<sup>5</sup> Yes.

---

What is unusual about *distanceToO*?

<sup>6</sup> 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.

<sup>7</sup> 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*?

<sup>8</sup> 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()?

<sup>9</sup> It is 6, because 2 + 3 is 5, and then we have to add 1 and 0.

---

Precisely. Now take a look at this extension of `CartesianPt`.

```
class ShadowedCartesianPt
    extends CartesianPt {
    int Δx;
    int Δy;
    ShadowedCartesianPt(int _x,
                         int _y,
                         int Δx,
                         int Δy) {
        super(_x,_y);
        Δx = _Δx;
        Δy = _Δy; }

    int distanceToO() {
        return
            super.distanceToO()
            +
            ⌊√(Δx2 + Δy2)⌋; }
}
```

<sup>10</sup> Nothing. We just discussed this kind of constructor for `ShadowedManhattanPt`.

What is unusual about the constructor?

---

Is this a `ShadowedCartesianPt`:  
`new ShadowedCartesianPt(12,5,3,4)?`

---

<sup>11</sup> Yes.

And what is the value of  
`new ShadowedCartesianPt(12,5,3,4)`  
.distanceToO()?

<sup>12</sup> 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

$$\sqrt{\Delta_x^2 + \Delta_y^2}$$

with  $\Delta_x$  replaced by 3 and  $\Delta_y$  replaced by 4.

---

What do we expect?

<sup>13</sup> 17, obviously.

---

---

Why 17?

<sup>14</sup> 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`.

<sup>15</sup> 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?

<sup>16</sup> 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.

<sup>17</sup> Okay.

```
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
            new CartesianPt(x + Δx,y + Δy)
            .distanceToO(); }
}
```

---

Do we still need the new `CartesianPt` after `distanceToO` has determined the distance?

<sup>18</sup> No, once we have the distance, we have no need for this point.<sup>1</sup>

<sup>1</sup> 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))?
```

<sup>19</sup> 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?

<sup>20</sup> That's obvious.

---

Is the rest of this chapter obvious, too?

<sup>21</sup> What?

---

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

<sup>22</sup> Oh. Well, that makes the hint obvious.

---

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

<sup>23</sup> Fine.

---

Are sandwiches square meals for you?

<sup>24</sup> They can be well-rounded.

---

Here are circles and squares.

<sup>25</sup> Then this must be the datatype that goes  
with it.

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

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

```
abstract class ShapeD {
    abstract
    boolean accept(ShapeVisitorI ask);
}
```

```
class Square extends ShapeD {
    int s;
    Square(int _s) {
        s = _s; }

    boolean accept(ShapeVisitorI ask) {
        return ask.forName(s); }
}
```

---

Very good. We also need an interface, and here it is.

```
interface ShapeVisitorI {
    boolean forCircle(int r);
    boolean forSquare(int s);
    boolean forTrans(PointD q,ShapeD s);
}
```

<sup>26</sup> It suggests that there is another variant: **Trans**.

---

Yes and we will need this third variant.

```
class Trans1 extends ShapeD {
    PointD q;
    ShapeD s;
    Trans(PointD _q,ShapeD _s) {
        q = _q;
        s = _s; }

    boolean accept(ShapeVisitorI ask) {
        return ask.forTrans(q,s); }
}
```

<sup>27</sup> Okay, now this looks pretty straightforward, but what's the point?

<sup>1</sup> A better name is **Translation**.

---

Let's create a circle.

<sup>28</sup> No problem:  
**new Circle(10)**.

---

How should we think about that circle?

<sup>29</sup> We should think about it as a circle with radius 10.

---

Good. So how should we think about  
**new Square(10)**?

<sup>30</sup> Well, that's a square whose sides are 10 units long.

---

Where are our circle and square located?

<sup>31</sup> What does that mean?

---

Suppose we wish to determine whether some <sup>[32](#)</sup> In that case, we must think of the circle as **CartesianPt** is inside of the circle? being drawn around the origin.

---

And how about the square?

<sup>[33](#)</sup> There are many ways to think about the location of the square.

---

Pick one.

<sup>[34](#)</sup> 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?

<sup>[35](#)</sup> Yes, it is, but barely.

---

And how about the circle?

<sup>[36](#)</sup> 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?

<sup>[37](#)</sup> 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))**?

<sup>[38](#)</sup> 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 <sup>[39](#)</sup> Also with **Trans**:

at **new CartesianPt(5,6)?**

**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?

<sup>[40](#)</sup> It is inside both of them.

---

---

How do we determine whether some point is inside a circle?

<sup>41</sup> 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?

<sup>42</sup> 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?

<sup>43</sup> No, we also need to do that for the  $y$  coordinate.

---

Aren't we on a roll?

<sup>44</sup> We have only done the easy stuff so far. It is not clear how to check these things when the circle or the square are not located at the origin.

---

Let's take a look at our circle around  
new `CartesianPt(5,6)`  
again. Can we think of this point as the  
origin?

<sup>45</sup> We can if we translate all other points by an appropriate amount.

---

By how much?

<sup>46</sup> By 5 in the  $x$  direction and 6 in the  $y$  direction, respectively.

---

How could we translate the points by an appropriate amount?

<sup>47</sup> We could subtract the appropriate amount from each point.

---

Is there a method in `PointD` that accomplishes that?

<sup>48</sup> Yes. Is that why we included *minus* in the new definition of `PointD`?

---

Indeed. And now we can define the visitor  $\text{HasPt}^V$ , whose methods determine whether some  $\text{Shape}^D$  has a  $\text{Point}^D$  inside of it.

```
class HasPtV implements ShapeVisitorT {
    PointD p;
    HasPtV(PointD _p) {
        p = _p; }

    public boolean forCircle(int r) {
        return p.distanceToO() ≤ r; }
    public boolean forSquare(int s) {
        if1 (p.x ≤ s)
            return (p.y ≤ s);
        else
            return false; }
    public
        boolean forTrans(PointD q,ShapeD s) {
            return s.accept(
                new HasPtV(p.minus(q))); }
}
```

<sup>49</sup> The three methods put into algebra what we just discussed.

What is the value of  
`new Circle(10)`  
`.accept(`  
`new HasPtV(new CartesianPt(10,10)))?`

<sup>50</sup> We said that this point wasn't inside of that circle, so the answer is **false**.

Good. And what is the value of  
`new Square(10)`  
`.accept(`  
`new HasPtV(new CartesianPt(10,10)))?`

<sup>51</sup> **true**.

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

```
new Trans(
    new CartesianPt(5,6),
    new Circle(10))
.accept(
    new HasPtV(new CartesianPt(10,10)))?
```

<sup>52</sup> We already considered that one, too. The value is **true**, because the circle's origin is at `new CartesianPt(5,6)`.

---

Right. And how about this:

```
new Trans(  
    new CartesianPt(5,4),  
    new Trans(  
        new CartesianPt(5,6),  
        new Circle(10)))  
.accept(  
    new HasPtV(new CartesianPt(10,10)))?
```

<sup>53</sup> Now that is tricky. We used **Trans** twice, which we should have expected given **Trans**'s definition.

But what is the value?

<sup>54</sup> First, we have to find out whether

```
new Trans(  
    new CartesianPt(5,6),  
    new Circle(10))  
.accept(  
    new HasPtV(new CartesianPt(5,6)))
```

is true or false.

---

And then?

<sup>55</sup> Second, we need to look at

```
new Circle(10)  
.accept(  
    new HasPtV(new CartesianPt(0,0))),
```

but the value of this is obviously true.

---

Very good. Can we nest **Trans** three times?

<sup>56</sup> Ten times, if we wish, because a **Trans** contains a **Shape<sup>D</sup>**, and that allows us to nest things as often as needed.

---

Ready to begin?

<sup>57</sup> What? Wasn't that it?

---

No. The exciting part is about to start.

<sup>58</sup> We are all eyes.

---

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

<sup>59</sup> It becomes a square, obviously.

---

And the orange on top?

<sup>60</sup> A circle, **Transed** appropriately.

---

Can we think of the two objects as one?

<sup>61</sup> 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(ShapeVisitorI ask) {  
        return _____; }  
}
```

---

<sup>62</sup> That looks obvious after the fact. But why is there a blank in *accept*?

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

<sup>63</sup> We know that a ShapeVisitor<sup>I</sup> 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?

<sup>64</sup> We need to change ShapeVisitor<sup>I</sup> 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 <sup>65</sup> Why can't we change it? to change ShapeVisitor<sup>I</sup>.

---

Just to make the problem more interesting. <sup>66</sup> In that case, we're stuck.

---

---

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

```
interface UnionVisitorI
  extends ShapeVisitorI {
  boolean forUnion(ShapeD s,ShapeD t);
}
```

---

Basically.<sup>1</sup> This extension produces an interface that contains all the obligations (*i.e.*, names of methods and what they consume and produce) of **ShapeVisitor<sup>I</sup>** and the additional one named *forUnion*.

<sup>1</sup> Unlike a class, an interface can actually extend several other interfaces. A class can implement several different interfaces.

---

Yes it should, but because **UnionVisitor<sup>I</sup>** extends **ShapeVisitor<sup>I</sup>**, it is also a **ShapeVisitor<sup>I</sup>**.

<sup>67</sup> Which means that we extend **interfaces** the way we extend **classes**.

<sup>68</sup> Does that mean *accept* in **Union** should receive a **UnionVisitor<sup>I</sup>**, so that it can use the *forUnion* method?

<sup>69</sup> We have been here before. Our *accept* method must consume a **ShapeVisitor<sup>I</sup>** and fortunately every **UnionVisitor<sup>I</sup>** implements a **ShapeVisitor<sup>I</sup>**, too. But if we know that *accept* consumes a **UnionVisitor<sup>I</sup>**, we can convert the **ShapeVisitor<sup>I</sup>** to a **UnionVisitor<sup>I</sup>** and invoke the *forUnion* method.

---

Perfect reasoning. Here is the completed definition of **Union**.

```
class Union extends ShapeD {
  ShapeD s;
  ShapeD t;
  Union(ShapeD _s,ShapeD _t) {
    s = _s;
    t = _t; }

  boolean accept(ShapeVisitorI ask) {
    return
      ((UnionVisitorI)ask).forUnion(s,t); }
}
```

<sup>70</sup> And it makes complete sense.

---

Let's create a Union shape.

<sup>71</sup> That's trivial.

```
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 <sup>72</sup> We can't.  $\text{HasPt}^V$  is only a  $\text{ShapeVisitor}^T$ , it whether

`new CartesianPt(12,16)`  
is inside?

---

Could it be a  $\text{UnionVisitor}^T$ ?

<sup>73</sup> No. It does not provide the method  
*forUnion*.

---

Define  $\text{UnionHasPt}^V$ , which extends  $\text{HasPt}^V$  with an appropriate method *forUnion*.

<sup>74</sup> Here it is. Its method checks whether the point is in one or the other part of a union. The other methods come from  $\text{HasPt}^V$ .

```
class UnionHasPtV extends HasPtV {  
    UnionHasPtV(PointD p) {  
        super(p); }  
  
    boolean forUnion(ShapeD s,ShapeD t) {  
        if1 (s.accept(this))  
            return true;  
        else  
            return t.accept(this); }  
}
```

---

<sup>1</sup> We could have written the if ... as  
`return s.accept(this) || t.accept(this);`

---

Does  $\text{UnionHasPt}^V$  contain *forUnion*?

<sup>75</sup> Of course, we just put it in.

---

---

Is `UnionHasPtV` a `UnionVisitorI`?

<sup>76</sup> It provides the required methods: `forCircle`, `forSquare`, `forTrans`, and `forUnion`.

---

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

```
class UnionHasPtV
    extends HasPtV
    implements UnionVisitorI {
    UnionHasPtV(PointD -p) {
        super(-p); }

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

---

<sup>77</sup> The first two additional words have an obvious meaning. They explicitly say that this visitor provides the services of `UnionVisitorI`. 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

```
new Trans(
    new CartesianPt(3,7),
    new Union(
        new Square(10),
        new Circle(10)))
.accept(
    new UnionHasPtV(
        new CartesianPt(13,17)))?
```

---

<sup>78</sup> We know how `forTrans` works, so we're really asking whether

`new CartesianPt(10,10)`  
is inside the `Union` shape.

So?

<sup>79</sup> 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?

<sup>80</sup> It should be true.

---

Let's see whether the value of  
**new Trans(**  
    **new CartesianPt(3,7),**  
    **new Union(**  
        **new Square(10),**  
        **new Circle(10))**  
.accept(  
    **new UnionHasPt<sup>V</sup>(**  
        **new CartesianPt(13,17))**)  
is true?

---

And?

<sup>82</sup> It's a Shape<sup>D</sup>.

---

How did we construct this shape?

<sup>83</sup> With Trans.

---

Which method should we use on it?

<sup>84</sup> *forTrans*, of course.

---

Where is *forTrans* defined?

<sup>85</sup> It is defined in HasPt<sup>V</sup>.

---

So what should we do now?

<sup>86</sup> We should determine the value of  
**new Union(**  
    **new Square(10),**  
    **new Circle(10))**  
.accept(  
    **new HasPt<sup>V</sup>(**  
        **new CartesianPt(10,10))**).

---

What type of object is

**new Union(**  
    **new Square(10),**  
    **new Circle(10))**?

---

<sup>87</sup> It's a Shape<sup>D</sup>.

- 
- How did we construct this `ShapeD`? <sup>88</sup> With `Union`.
- 
- So which method should we use on it? <sup>89</sup> `forUnion`, of course.
- 
- How do we find the appropriate `forUnion` method? <sup>90</sup> In `accept`, which is defined in `Union`, we confirm that  
`new HasPtV(  
    new CartesianPt(10,10))`  
is a `UnionVisitorT` and then invoke its `forUnion`.
- 
- Is an instance of `HasPtV` a `UnionVisitorT`? <sup>91</sup> No!
- 
- Does it contain a method `forUnion`? <sup>92</sup> No!
- 
- Then what is the value of  
`new Union(  
    new Square(10),  
    new Circle(10))  
.accept(  
    new HasPtV(  
        new CartesianPt(10,10)))?` <sup>93</sup> It doesn't have a value. We are stuck.<sup>1</sup>
- 
- What do we do next? <sup>94</sup> Relax. Read a novel. Take a nap.
- 
- Which of those is best? <sup>95</sup> You guessed it: whatever you did is best.
- 
- We should have prepared this extension in a better way. <sup>96</sup> How could we have done that?

---

Here is the definition of `HasPtV` that we should have provided if we wanted to extend it without making changes.

```
class HasPtV implements ShapeVisitorT {
    PointD p;
    HasPtV(PointD _p) {
        p = _p; }
    ShapeVisitorT newHasPt(PointD p) {
        return new HasPtV(p); }

    public boolean forCircle(int r) {
        return p.distanceToO() ≤ r; }
    public boolean forSquare(int s) {
        ifl (p.x ≤ s)
            return (p.y ≤ s);
        else
            return false; }
    public
    boolean forTrans(PointD q,ShapeD s) {
        return
            s.accept(newHasPt(p.minus(q))); }
}
```

<sup>97</sup> In two ways. First, it contains a new method: `newHasPt`. Second, it uses the new method in place of `new HasPtV` in `forTrans`.

How does this definition differ from the previous one?

---

Good. What does `newHasPt` produce?

<sup>98</sup> A new `ShapeVisitorT`, as its interface implies.

---

And how does it produce that?

<sup>99</sup> By constructing a `new` instance of `HasPtV`.

---

Is `newHasPt` like a constructor?

<sup>100</sup> It is virtually indistinguishable from a constructor, which is why it is above the line that separates constructors from methods.

---

---

Does that mean the new definition of `HasPtV`<sup>101</sup> and the previous one are really the same?<sup>1</sup>

<sup>1</sup> A functional programmer would say that `newHasPt` and `HasPtV` are  $\eta$ -equivalent.

They are mostly indistinguishable. Both `forTranses`, the one in the previous and the one in the new definition of `HasPtV`, produce the same values when they consume the same values.

---

Very well. But how does that help us with our problem?

---

Can we override `newHasPt` when we extend `HasPtV`?

Let's override `newHasPt` in `UnionHasPtV`.

That's true. Should it produce a `HasPtV` or a `UnionHasPtV`?

Good answer. Should we repeat it?

<sup>102</sup> That's not obvious.

<sup>103</sup> Yes, we can override any method that we wish to override.

<sup>104</sup> When we override it, we need to make sure it produces a `ShapeVisitorT`.

<sup>105</sup> The latter. Then `forTrans` in `HasPtV` keeps producing a `UnionHasPtV`, if we start with a `UnionHasPtV`.

<sup>106</sup> 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.*

---

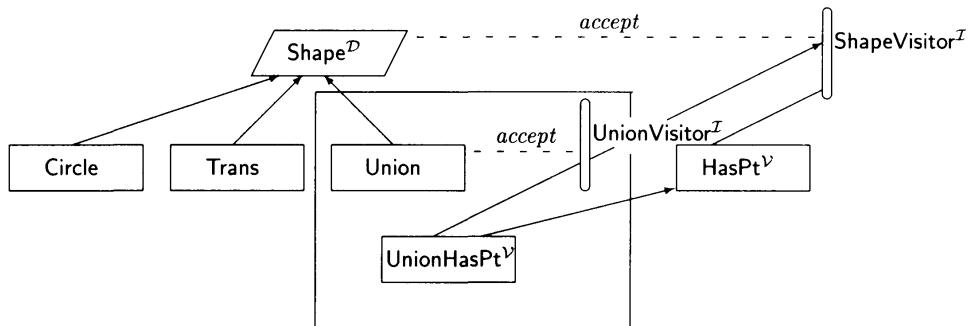
And that's exactly what we need. Revise the <sup>107</sup> Here it is.  
definition of `UnionHasPtV`.<sup>1</sup>

```
class UnionHasPtV
    extends HasPtV
    implements UnionVisitorI {
    UnionHasPtV(PointD -p) {
        super(-p);
    }
    ShapeVisitorI newHasPt(PointD p) {
        return new UnionHasPtV(p);
    }
}
public
boolean forUnion(ShapeD s,ShapeD t){
    if (s.accept(this))
        return true;
    else
        return t.accept(this);
}
```

<sup>1</sup> This is an instance of the *factory method* pattern [4].

---

If we assemble all this into one picture, what <sup>108</sup> A drawing that helps our understanding of  
do we get?



---

What does the box mean?

<sup>109</sup> Everything outside of the box is what we designed originally and considered to be unchangeable; everything inside is our extension.

---

Does the picture convey the key idea of this chapter?

<sup>110</sup> No. It does not show the addition of a constructor-like method to `HasPtV` and how it is overridden in `UnionHasPtV`.

---

Is anything missing?

<sup>111</sup> `Square`, but that's okay.

---

Let's see whether this definition works.  
What is the value of

```
new Trans(  
    new CartesianPt(3,7),  
    new Union(  
        new Square(10),  
        new Circle(10)))  
.accept(  
    new UnionHasPtV(  
        new CartesianPt(13,17)))?
```

---

Which method should we use on it?

<sup>113</sup> `forTrans`, of course.

---

Where is `forTrans` defined?

<sup>114</sup> It is defined in `HasPtV`.

---

So what should we do now?

<sup>115</sup> We should determine the value of  
new Union(  
 new Square(10),  
 new Circle(10))  
.accept(  
 this.newHasPt(  
 new CartesianPt(10,10))).

---

What is `this`?

<sup>116</sup> The current visitor, of course.

---

And how does that work?

<sup>117</sup> We determine the value of  
this.newHasPt(  
 new CartesianPt(10,10))  
and then use `accept` for the rest.

---

And what do we create?

<sup>118</sup> The new `UnionVisitorT`:  
    `new UnionHasPtV(`  
    `new CartesianPt(10,10)).`

---

What is the value of  
    `new Union(`  
        `new Square(10),`  
        `new Circle(10))`  
    `.accept(`  
        `new UnionHasPtV(`  
        `new CartesianPt(10,10)))?`

---

<sup>119</sup> `UnionHasPtV` also satisfies the interface  
`ShapeVisitorT`, so now we can invoke the  
*forUnion* method.

How do we do that?

<sup>120</sup> We first determine the value of  
    `new Square(10)`  
    `.accept(`  
        `new UnionHasPtV(`  
        `new CartesianPt(10,10))).`

If it is true, we're done.

---

Is it true?

<sup>121</sup> It is. So we're done and we got the value we  
expected.

---

Are we happy now?

<sup>122</sup> Ecstatic.

---

Is it good to have extensible definitions?

<sup>123</sup> 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?

<sup>124</sup> Yes, we can and should always do so.

---

And why is that?

<sup>125</sup> Because no program is ever finished.

---

Are you hungry yet?

<sup>126</sup> Are our meals ever finished?

---

10.  
The State of  
Things to Come



---

Have you ever wondered where the pizza pies<sup>1</sup> You should have, because someone needs to come from?

Here is our pizza pieman.

<sup>2</sup> This is beyond anything we have seen before.

```
class PiemanM implements PiemanT {
    PieD p = new Bot();
    public int addTop(Object t) {
        p = new Top(t,p)
        ;
        return occTop(t);
    }
    public int remTop(Object t) {
        p = (PieD)p.accept(new RemV(t))
        ;
        return occTop(t);
    }
    public int substTop(Object n,Object o) {
        p = (PieD)p.accept(new SubstV(n,o))
        ;
        return occTop(n);
    }
    public int occTop(Object o) {
        return
            ((Integer)p.accept(new OccursV(o)))
            .intValue();
    }
}
```

<sup>M</sup> This superscript is a reminder that the class manages a data structure. Lower superscripts when you enter this kind of definition in a file: `PiemanM`.

---

How so? Haven't we seen `PieD`, `Top`, and `Bot`<sup>3</sup> We have seen them. before?

And haven't we seen visitors like `RemV`, `SubstV`, and `OccursV` for various datatypes?

<sup>4</sup> Yes, yes. But what are the stand-alone semicolons about?

Let's not worry about them for a while.

<sup>5</sup> Fine, but they are weird.

---

Here is the **interface** for Pieman<sup>M</sup>.

```
interface PiemanT {
    int addTop(Object t);
    int remTop(Object t);
    int substTop(Object n, Object o);
    int occTop(Object o);
}
```

<sup>6</sup> 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*.

Here are PieVisitor<sup>T</sup> and Pie<sup>D</sup>.

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

```
abstract class PieD {
    abstract
        Object accept(PieVisitorT ask);
}
```

Define Bot and Top.

<sup>8</sup> They are very familiar.

```
class Bot extends PieD {
    Object accept(PieVisitorT ask) {
        return ask.forBot();
    }
}
```

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

```
Object accept(PieVisitorT ask) {
    return ask.forTop(t, r);
}
```

---

Here is  $\text{Occurs}^V$ . It counts how often some topping occurs on a pie.

```
class OccursV implements PieVisitorI {
    Object a;
    OccursV(Object _a) {
        a = _a; }

    public Object forBot() {
        return new Integer(0); }
    public Object forTop(Object t,PieD r) {
        if (t.equals(a))
            return
                new Integer(((Integer)
                    (r.accept(this)))
                    .intValue()
                    + 1);
        else
            return r.accept(this); }
}
```

<sup>9</sup> And this little visitor substitutes one good topping for another.

```
class SubstV implements PieVisitorI {
    Object n;
    Object o;
    SubstV(Object _n,Object _o) {
        n = _n;
        o = _o; }

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

---

Great! Now we have almost all the visitors for our pieman. Define  $\text{Rem}^V$ , which removes a topping from a pie.

<sup>10</sup> We remember that one, too.

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

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

---

Now we are ready to talk. What is the value of<sup>11</sup> We first create a  $\text{Pieman}^M$  and then ask how many anchovies occur on the pie.

**new Pieman<sup>M</sup>().occTop(new Anchovy())?**

---

Which pie?

<sup>12</sup> The pie named  $p$  in the new  $\text{Pieman}^M$ .

---

And how many anchovies are on that pie?<sup>13</sup> None.

---

And what is the value of

**new Pieman<sup>M</sup>().addTop(new Anchovy())?**

<sup>14</sup> That's where those stand-alone semicolons come in again. They were never explained.

---

True. If we wish to determine the value of

**new Pieman<sup>M</sup>().addTop(new Anchovy()),**

we must understand what

$p = \text{new Top(new Anchovy(), } p)$   
;  
**return occTop(new Anchovy())**

means?

<sup>15</sup> 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  $p$  such that

$$p = \text{new Top(new Anchovy(), } p)?$$

---

That's right. But that's what happens when<sup>16</sup> So what does it mean?  
you have one too many double espressos.

---

Here it means that  $p$  changes and that future references to  $p$  reflect the change.

<sup>17</sup> And the change is that  $p$  has a new topping, right?

---

When does the future begin?

<sup>18</sup> Does it begin below the stand-alone semicolon?

---

That's precisely what a stand-alone semicolon means. Now do we know what

**return occTop(new Anchovy())**  
produces?

<sup>19</sup> It produces the number of anchovies on  $p$ .

---

And how many are there?

<sup>20</sup> We added one, so the value is 1.

---

And now what is the value of

**new Pieman<sup>M</sup>().addTop(new Anchovy())?**

<sup>21</sup> 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*.

<sup>22</sup> Oh, isn't there a way to place several  
requests with the same pieman?

---

Yes, there is. Take a look at this:

**Pieman<sup>T</sup> y = new Pieman<sup>M</sup>().**

<sup>23</sup> Okay, *y* stands for some pieman.

---

What is the value of

*y.addTop(new Anchovy())?*

<sup>24</sup> 1. We know that.

---

And now what is the value of

*y.substTop(new Tuna(),new Anchovy())?*

<sup>25</sup> 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.occTop(new Anchovy())?*

<sup>26</sup> 0, because *y*'s pie no longer contains any  
anchovies.

---

Very good. And now take a look at this:

**Pieman<sup>T</sup> yy = new Pieman<sup>M</sup>().**

What is the value of

*yy.addTop(new Anchovy())*

*;*

*yy.addTop(new Anchovy())*

*;*

*yy.addTop(new Salmon())*

*;*

*...*

<sup>27</sup> What are the

*..*

doing at the end?

---

---

Because this is only half of what we want to look at. Here is the other half:

```
yy.addTop(new Tuna())
;
yy.addTop(new Tuna())
;
yy.substTop(new Tuna(),new Anchovy())?
```

<sup>28</sup> 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  
*yy.remTop(new Tuna())*  
after we are through with all that?

<sup>29</sup> It's 0, because *remTop* first removes all tunas and then counts how many there are left.

---

Does that mean *remTop* always produces 0? <sup>30</sup> Yes, it always does.

---

Now what is the value of  
*yy.occTop(new Salmon())*?

<sup>31</sup> 1.

---

And how about  
*y.occTop(new Salmon())*?

<sup>32</sup> 0, because *y* and *yy* are two different piemen.

---

Is *yy* the same pieman as before?

<sup>33</sup> No, it changed.

---

So is it the same one?

<sup>34</sup> When we eat a pizza pie, we change, but we are still the same.

---

When we asked *yy* to substitute all anchovies <sup>35</sup> The *p* in *yy* changed, nothing else. by tunas, did the pie change?

---

Does that mean that anybody can write  
*yy.p = new Bot()*  
and thus change a pieman like *yy*?

<sup>36</sup> No, because *yy*'s type is *Pieman $\tau$* , *p* isn't available. Only *addTop*, *remTop*, *substTop*, and *occTop* are visible.

---

Isn't it good that we didn't include  $p$  in  $\text{Pieman}^{\mathcal{I}}$ ?

<sup>37</sup> 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?

<sup>38</sup> Just like chicken soup.

---

Can we define a different version of  $\text{Subst}^{\mathcal{V}}$  so <sup>39</sup> We can't do that yet. that it changes toppings the way a pieman changes his pies?

---

And that's what we discuss next. Do you need a break?

<sup>40</sup> No, a cup of coffee will do.

---

Compare this new  $\text{PieVisitor}^{\mathcal{I}}$  with the first one in this chapter.

```
interface PieVisitor $^{\mathcal{I}}$  {
    Object forBot(Bot that);
    Object forTop(Top that);
}
```

<sup>41</sup> It isn't all that different. A  $\text{PieVisitor}^{\mathcal{I}}$  must still provide two methods: *forBot* and *forTop*, except that the former now consumes a *Bot* and the latter a *Top*.

---

True. Here is the unchanged datatype.

<sup>42</sup> The definition is straightforward.

```
abstract class Pie $^{\mathcal{D}}$  {
    abstract
        Object accept(PieVisitor $^{\mathcal{I}}$  ask);
}
```

```
class Bot extends Pie $^{\mathcal{D}}$  {
    Object accept(PieVisitor $^{\mathcal{I}}$  ask) {
        return ask.forBot(this);
    }
}
```

Define the *Bot* variant.

---

Is it? Why does it use **this**?

<sup>43</sup> We only have one instance of *Bot* when we use *forBot*, namely **this**, so *forBot* is clearly supposed to consume **this**.

---

That's progress. And that's what happens in <sup>44</sup> Interesting.  
Top, too.

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

    Object accept(PieVisitorT ask) {
        return ask.forTop(this); }
}
```

---

Modify this version of Occurs<sup>V</sup> so that it implements the new PieVisitor<sup>T</sup>.

```
class OccursV implements PieVisitorT {
    Object a;
    OccursV(Object _a) {
        a = _a; }

    public Object forBot() {
        return new Integer(0); }
    public Object forTop(Object t,PieD r) {
        if (t.equals(a))
            return
                new Integer(((Integer)
                    (r.accept(this)))
                    .intValue()
                    + 1);
        else
            return r.accept(this); }
}
```

<sup>45</sup> The *forBot* method basically stays the same, but *forTop* changes somewhat.

```
class OccursV implements PieVisitorT {
    Object a;
    OccursV(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?

<sup>46</sup> It now consumes a Bot, which is why we had to add (Bot *that*) behind its name.

---

---

How does *forTop* change?

<sup>47</sup> 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?

<sup>48</sup> With that, we can replace the fields *t* and *r* with *that.t* and *that.r*.

---

Isn't that easy?

<sup>49</sup> This modification of *Occurs<sup>V</sup>* certainly is.

---

Then try *Rem<sup>V</sup>*.

<sup>50</sup> It's easy; we use the same trick.

```
class RemV implements PieVisitorT {  
    Object o;  
    RemV(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,  
                    (PieD)that.r.accept(this)); }  
}
```

---

Do we need to do *Subst<sup>V</sup>*?

<sup>51</sup> Not really. It should be just like *Rem<sup>V</sup>*.

---

And indeed, it is. Happy now?

<sup>52</sup> So far, so good. But what's the point of this exercise?

---

Oh, *Point<sup>D</sup>*s? They will show up later.

<sup>53</sup> Seriously.

---

---

Here is the point. What is new about this version of  $\text{Subst}^V$ ?

```
class SubstV implements PieVisitorT {
    Object n;
    Object o;
    SubstV(Object _n, Object _o) {
        n = _n;
        o = _o;
    }

    public Object forBot(Bot that) {
        return that;
    }
    public Object forTop(Top that) {
        if (o.equals(that.t)) {
            that.t = n
        ;
            that.r.accept(this)
        ;
            return that;
        }
        else {
            that.r.accept(this)
        ;
            return that;
        }
    }
}
```

<sup>54</sup> There are no **news**.

Don't they say "no **news** is good news?"

<sup>55</sup> Does this saying apply here, too?

---

Yes, because we want to define a version of  $\text{Subst}^V$  that modifies toppings without constructing a **new** pie.

<sup>56</sup> That's a way of putting it.

---

What do the methods of  $\text{Subst}^V$  always **return**?

<sup>57</sup> They always return *that*, which is the object that they consume.

---

So how do they substitute toppings?

<sup>58</sup> By changing the *that* before they return it. Specifically, they change the *t* field of *that* to *n* when it equals *o*.

---

What?

<sup>59</sup> The  
*that.t = n*  
does it.

---

Correct. And from here on, *that.t* holds the new topping. What is

*that.r.accept(this)*  
about?

<sup>60</sup> In the previous *Subst<sup>V</sup>*, *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<sup>V</sup>*?

<sup>61</sup> Not really. It does what it does, which is what we wanted.<sup>1</sup>

<sup>1</sup> This is a true instance of the *visitor* pattern [4]. What we previously called “visitor” pattern instances, were simple variations on the theme.

---

Do we have to change *Pieman<sup>M</sup>*?

<sup>62</sup> 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?

<sup>63</sup> Yes, because the *Pieman<sup>M</sup>* manages the toppings of *p*, and nobody else sees *p*.

---

Can we do *LtdSubst<sup>V</sup>* now without creating new instances of *LtdSubst<sup>V</sup>* or *Top*?

<sup>64</sup> 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.*

---

Here is a true dessert. It will help us understand what the point of state is.

```
abstract class PointD {
    int x;
    int y;
    PointD(int _x,int _y) {
        x = _x;
        y = _y; }

    boolean closerToO(PointD p) {
        return
            distanceToO() ≤ p.distanceToO(); }
    PointD minus(PointD p) {
        return
            new CartesianPt(x - p.x,y - p.y); }
    abstract int distanceToO();
}
```

<sup>65</sup> The datatype has three extensions.

```
class CartesianPt extends PointD {
    CartesianPt(int _x,int _y) {
        super(_x,_y); }

    int distanceToO() {
        return ⌊√x² + y²⌋; }
}
```

```
class ManhattanPt extends PointD {
    ManhattanPt(int _x,int _y) {
        super(_x,_y); }

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

```
class ShadowedManhattanPt
    extends ManhattanPt {
    int Δx;
    int Δy;
    ShadowedManhattanPt(int _x,
                         int _y,
                         int Δx,
                         int Δy) {
        super(_x,_y);
        Δx = Δx;
        Δy = Δy; }

    int distanceToO() {
        return
            super.distanceToO() + Δx + Δy; }
}
```

---

Aren't we missing a variant?

<sup>66</sup> Yes, we are missing ShadowedCartesianPt.

---

---

Good enough. We won't need it. Here is one<sup>67</sup> Shouldn't we add a method that changes all point:

**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.

<sup>68</sup> First we must know what the method is supposed to produce.

---

The method should return the new distance<sup>69</sup> Now we know how to do this.  
to the origin.

```
abstract class PointD {
    int x;1
    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

<sup>70</sup> 5.

**new ManhattanPt(1,4).**

What is the value of

`ptChild.distanceToO()`?

---

---

What is the value of  
*ptChild.moveBy(2,8)?*

<sup>71</sup> 15.

---

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()?*

<sup>72</sup> 7.

---

What is the value of  
*ptChildBalloon.moveBy(2,8)?*

<sup>73</sup> 17, of course.

---

Did the balloon move, too?

<sup>74</sup> Yes, it just moved along as we moved the  
point.

---

Isn't that powerful?

<sup>75</sup> It sure is. We added one method, used it,  
and everything moved.

---

The more things change, the cheaper our  
desserts get.

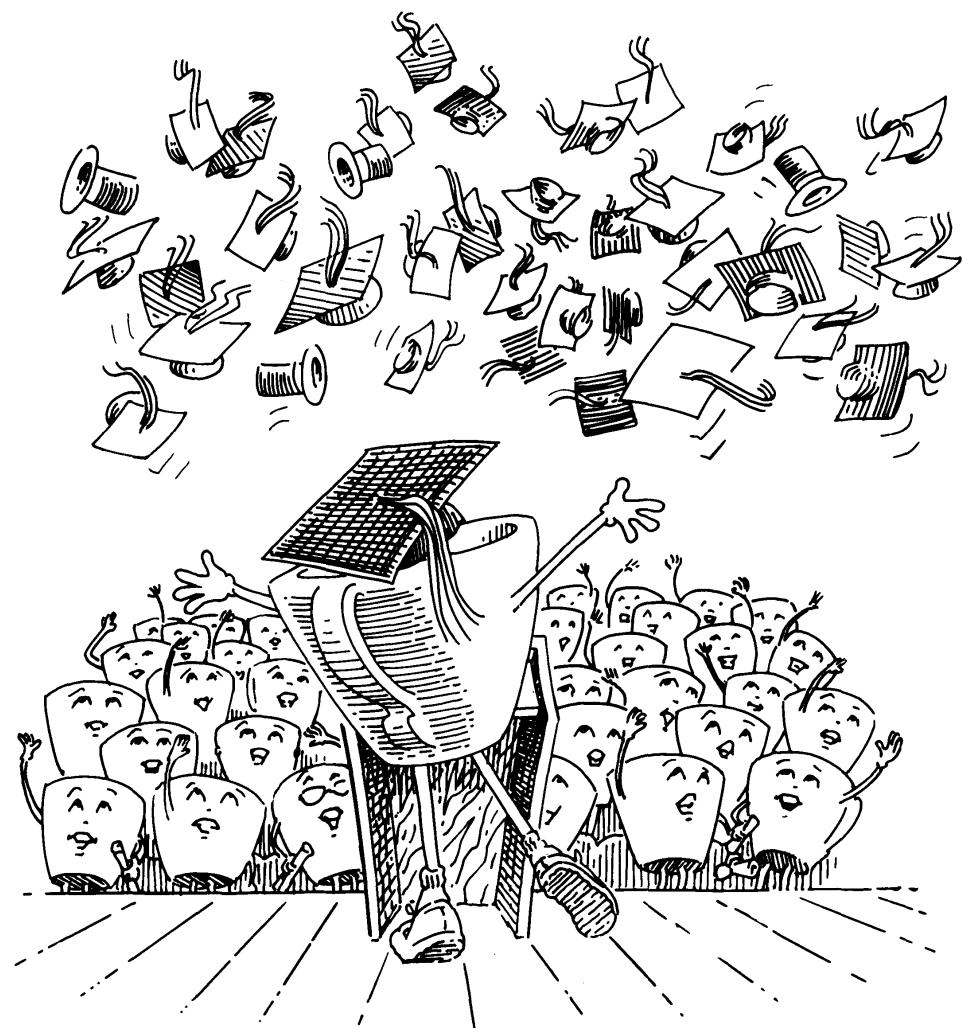
<sup>76</sup> Yes, but to get to the dessert, we had to  
work quite hard.

---

Correct but now we are through and it is  
time to go out and to celebrate with a grand  
dinner.

<sup>77</sup> Don't forget to leave a tip.

# Consciousness



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

1. Arnold and Gosling. *The Java Programming Language*. Addison-Wesley, Reading, Massachusetts, 1996.
2. Buschmann, Meunier, Rohnert, Sommerlad, and Stal. *A System of Patterns: Pattern-Oriented Software Architecture*. John Wiley & Sons, Ltd. Chichester, Europe, 1996.
3. Firesmith and Eykholt. *Dictionary of Object Technology*. SIGS Books, Inc., New York, New York, 1995 and Prentice Hall, Englewood Cliffs, New Jersey, 1995.
4. Gamma, Helm, Johnson, and Vlissides. *Design Patterns*. Addison-Wesley, Reading, Massachusetts, 1994.
5. Gosling, Joy, and Steele. *The Java Language Specification*. Addison-Wesley, Reading, Massachusetts, 1996.
6. Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, Reading, Massachusetts, 1994.

---

This is for the loyal Schemers and MLers.

```
interface TI {
    o→oI apply(TI x);
}
```

```
interface o→oI {
    Object apply(Object x);
}
```

```
interface oo→ooI {
    o→oI apply(o→oI x);
}
```

```
interface oo→oo→ooI {
    o→oI apply(oo→ooI x);
}
```

```
class Y implements oo→oo→ooI {
    public o→oI apply(oo→ooI f) {
        return new H(f).apply(new H(f));
    }
}
```

```
class H implements TI {
    oo→ooI f;
    H(oo→ooI _f) {
        f = _f;
    }
    public o→oI apply(TI x) {
        return f.apply(new G(x));
    }
}
```

```
class G implements o→oI {
    TI x;
    G(TI _x) {
        x = _x;
    }
    public Object apply(Object y) {
        return (x.apply(x)).apply(y);
    }
}
```

---

No, we wouldn't forget factorial.

```
class MkFact implements oo→ooI {
    public o→oI apply(o→oI fact) {
        return new Fact(fact);
    }
}
```

```
class Fact implements o→oI {
    o→oI fact;
    Fact(o→oI _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());
    }
}
```

## Copyrighted material

### Index

- Add, 124
- addTop*, 161
- Anchovy, 43, 53, 64, 65, 69, 72
- Apple, 100
- Base, 10
- bHasFruit<sup>V</sup>, 104
- bIsFlat<sup>V</sup>, 102
- bIsSplit<sup>V</sup>, 102, 103
- Bot, 69, 78, 83, 85, 86, 91, 93, 162, 167
- Brass, 29
- Bud, 100, 101, 107, 110, 112
- bTreeVisitor<sup>T</sup>, 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<sup>D</sup>, 130
- Expr<sup>D</sup>, 119
- ExprVisitor<sup>T</sup>, 118
- Fig, 100
- Fish<sup>D</sup>, 69, 72
- Flat, 100, 101, 108, 110, 112
- Fruit<sup>D</sup>, 100
- Gold, 29
- HasPt<sup>V</sup>, 147, 155
- Holder, 28, 36
- iHeight<sup>V</sup>, 108
- IntEval<sup>V</sup>, 120, 122, 131
- iOccurs<sup>V</sup>, 111
- IsFlat<sup>V</sup>, 113
- IsSplit<sup>V</sup>, 113
- isVegetarian, 25, 27
- IsVegetarian<sup>V</sup>, 63
- isVeggie, 30, 31, 36
- iTreeVisitor<sup>T</sup>, 107
- Kebab<sup>D</sup>, 28, 36
- Lamb, 16, 27, 28, 36, 59, 62
- Layer<sup>D</sup>, 10
- Lemon, 100
- LtdSubst<sup>V</sup>, 95, 97, 132–136
- ManhattanPt, 5, 13, 38, 39, 139, 172
- mem*, 124
- minus*, 139
- moveBy*, 173
- newHasPt*, 155, 157
- Num<sup>D</sup>, 7, 79
- occTop*, 161
- Occurs<sup>V</sup>, 114, 163, 168
- Olive, 43, 53, 64, 65
- OneMoreThan, 7, 79
- Onion, 16, 27, 28, 36, 59, 62
- onlyOnions*, 18, 27
- OnlyOnions<sup>V</sup>, 57, 59
- Peach, 100
- Pear, 100
- Pepper, 4, 37
- Pie<sup>D</sup>, 69, 78, 83, 85, 86, 91, 93, 162, 167
- PieVisitor<sup>T</sup>, 92, 162, 167
- Pieman<sup>T</sup>, 162
- Pieman<sup>M</sup>, 161
- Pizza<sup>D</sup>, 43, 53, 63–65
- Plate<sup>D</sup>, 29
- plus*, 124
- Plus, 119
- Point<sup>D</sup>, 5, 13, 40, 139, 172, 173
- prod, 124
- Prod, 119

Radish, 28, 36  
 $\text{rem}_A$ , 45, 49, 70  
 $\text{Rem}^V$ , 66, 71, 73  
 $\text{RemFish}^V$ , 74  
 $\text{Rem}^V$ , 77, 87, 93, 163, 169  
 $\text{RemInt}^V$ , 76  
 $\text{remTop}$ , 161  
 $\text{Rod}^D$ , 29  
  
 Sabre, 29  
 Sage, 5  
 Salmon, 69, 72  
 Salt, 4  
 Sausage, 43, 53, 63–65  
 Seasoning $^D$ , 4  
 $\text{Set}^D$ , 123  
 $\text{SetEval}^V$ , 126, 131  
 ShadowedCartesianPt, 141, 142  
 ShadowedManhattanPt, 139, 172  
 Shallot, 28, 36  
 $\text{Shape}^D$ , 143  
 $\text{ShapeVisitor}^T$ , 144  
 $\text{Shish}^D$ , 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  
  
 $\text{subAbC}$ , 52, 53  
 $\text{SubAbC}^V$ , 66  
 $\text{Subst}^D$ , 133, 134  
 $\text{SubstFish}^V$ , 82  
 $\text{Subst}^V$ , 83, 87, 89, 94, 132–136, 163, 170  
 $\text{SubstInt}^V$ , 82  
 $\text{substTop}$ , 161  
 Sword, 29  
  
 Thyme, 5  
 Tomato, 16, 27, 28, 36, 59, 62  
 Top, 69, 78, 83, 85, 86, 91, 93, 162, 168  
 $\text{topAwC}$ , 50, 53  
 $\text{TopAwC}^V$ , 66  
 Trans, 144  
 $\text{Tree}^D$ , 100, 101, 107, 109, 112  
 $\text{TreeVisitor}^T$ , 112  
 $\text{tSubst}^V$ , 111  
 $\text{tTreeVisitor}^T$ , 109  
 Tuna, 69, 72  
  
 Union, 149, 150  
 $\text{UnionHasPt}^V$ , 151, 152, 157  
 $\text{UnionVisitor}^T$ , 150  
  
 $\text{whatHolder}$ , 33–36  
 Wood, 29  
  
 Zero, 7, 79  
 Zucchini, 37,

## Copyrighted material

### A Little Java, A Few Patterns

Matthias Felleisen and Daniel P. Friedman

Foreword by Ralph E. Johnson

Drawings by Duane Bibby

Java is a new object-oriented programming language for programming the Internet and intelligent appliances. In a very short time it has become one of the most widely used programming languages in education as well as for commercial applications. Design patterns, which have moved object-oriented programming to a new level, provide programmers with a language to communicate with others about their designs. As a result, programs become more readable, reusable, and easily extensible.

Matthias Felleisen and Daniel Friedman use a small subset of Java to introduce pattern-directed program design. With their usual clarity and flair, they gently guide readers through the fundamentals of object-oriented programming and pattern-based design. Readers new to programming, as well as those with some background, will enjoy their learning experience as they work their way through Felleisen and Friedman's lessons.

Matthias Felleisen is Professor of Computer Science at Rice University. Daniel P. Friedman is Professor of Computer Science at Indiana University. They are the authors of *The Little Schemer*, *The Seasoned Schemer*, and *The Little MLer*.

"This is a book of 'why' not 'how.' If you are interested in the nature of computation and curious about the very idea behind object orientation, this book is for you. This book will engage your brain (if not your tummy). Through its sparkling interactive style, you will learn about three essential OO concepts: interfaces, visitors, and factories. A refreshing change from the 'yet another Java book' phenomenon. Every serious Java programmer should own a copy."—Gary McGraw, Ph.D., Research Scientist at Reliable Software Technologies

The MIT Press

Massachusetts Institute of Technology • Cambridge, Massachusetts 02142

<http://mitpress.mit.edu> • FELTP 0-262-56115-8

