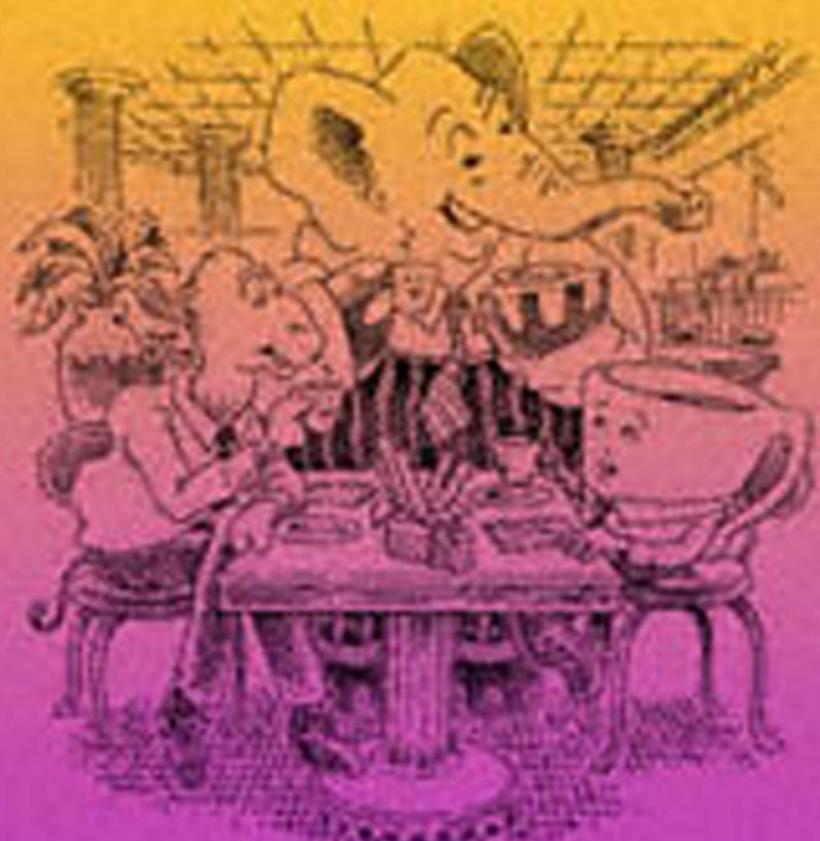


A Little Java, A Few Patterns



Matthias Felleisen and Daniel P. Friedman
Foreword by Ralph E. Johnson

A Little Java, A Few Patterns

Matthias Felleisen

*Rice University
Houston, Texas*

Daniel P. Friedman

*Indiana University
Bloomington, Indiana*

Drawings by Duane Bibby

Foreword by Ralph E. Johnson

The MIT Press
Cambridge, Massachusetts
London, England

© 1998 Massachusetts Institute of Technology

All rights reserved. No part of this book may be reproduced in any form by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the publisher.

This book was set by the authors and was printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Felleisen, Matthias

A little Java, a few patterns. / Matthias Felleisen and Daniel P. Friedman;
drawings by Duane Bibby; foreword by Ralph E. Johnson

p. cm.

Includes index and bibliographical references.

ISBN 0-262-56115-8 (pbk : alk. paper)

1. Java (Computer program language) I. Friedman, Daniel P. II. Title.

QA76.73.J38F45 1998

005.13'3—dc21

97-40548

CIP

To Helga, Christopher, and Sebastian.

*To Mary, Rob, Rachel, Sara,
and to the memory of Brian.*

This page intentionally left blank

Contents

Foreword **ix**

Preface **xi**

Experimenting with Java **xiii**

1. Modern Toys **3**

2. Methods to Our Madness **13**

3. What's New? **43**

4. Come to Our Carousel **57**

5. Objects Are People, Too **69**

6. Boring Protocols **85**

7. Oh My! **99**

8. Like Father, Like Son **117**

9. Be a Good Visitor **139**

10. The State of Things to Come **161**

Commencement **177**

Index **178**

This page intentionally left blank

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

This page intentionally left blank

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 SIATEX. 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:¹

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() ); } }
```

¹See 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... - - - - -
```

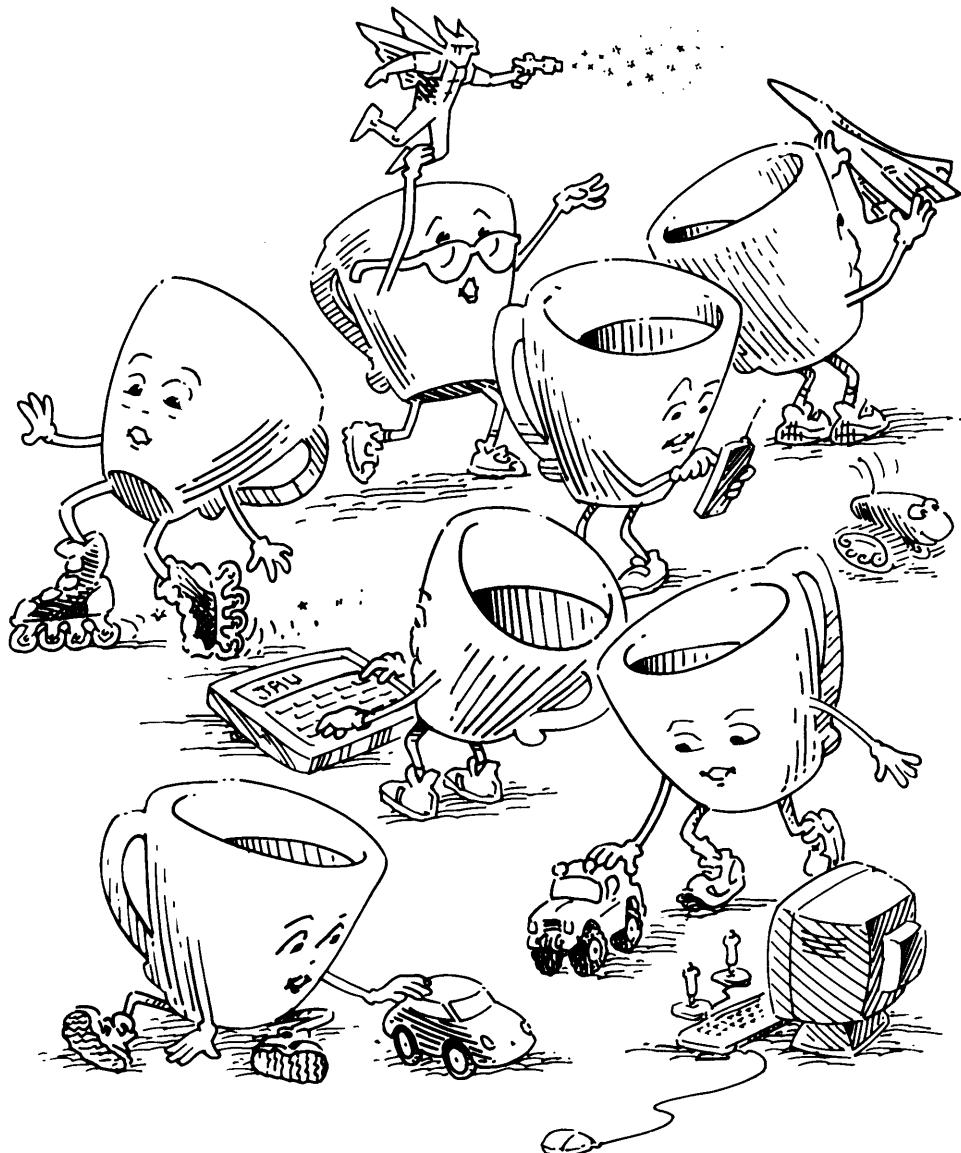
For example, if you wish to experiment with the methods of **PiemanM** as defined in chapter 10, add the following definition to the end of your file:

```
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**.

A Little Java, A Few Patterns

1.
Modern Toys



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

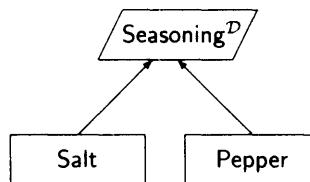
Draw the picture that characterizes the essential relationships among the following classes.

abstract class Seasoning^D {}

class Salt extends Seasoning^D {}

class Pepper extends Seasoning^D {}

¹⁴ Is this it?



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

Yes. We say Seasoning^D is a datatype, and Salt and Pepper are its variants.

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

Yes, in a way. Now, is

new Salt()
a Seasoning^D?

¹⁶ Yes, it is, because **new Salt()** creates an instance of Salt, and every instance of Salt is also a Seasoning^D.

And

new Pepper()?

¹⁷ It's also a Seasoning^D, because **new Pepper()** creates an instance of Pepper, and every instance of Pepper is also a Seasoning^D.

What are **abstract**, **class**, and **extends**?

¹⁸ Easy:
abstract class introduces a datatype,
class introduces a variant, and
extends connects a variant to a datatype.

Is there any other Seasoning^D?

¹⁹ No, because only Salt and Pepper extend Seasoning^D.¹

¹ Evaluating **new Salt()** twice does not produce the same value, but we ignore the distinction for now.

Correct, Salt and Pepper are the only variants of the datatype `SeasoningD`. Have we seen a datatype like `SeasoningD` before?

Let's define more `SeasoningD`s.

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

²⁰ No, but `boolean` is a type that also has just two values.

²¹ We can have lots of `SeasoningD`s.

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

And then there were four.

²² Yes.

What is a Cartesian point?

²³ It is basically a pair of numbers.

What is a point in Manhattan?

²⁴ An intersection where two city streets meet.

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

```
abstract class PointD {}
```

²⁵ Each of them contains three things between { and }. The `x` and the `y` are obviously the coordinates of the points. But what is the remaining thing above the bold bar?¹

```
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; }
```

```
}
```

¹ This bar indicates the end of the constructor definition. It is used as an eye-catching separator. We recommend that you use

// -----

when you enter it in a file.

The underlined occurrences of `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`.

Correct. Is this a `ManhattanPt`:
`new ManhattanPt(2,3)?`

Isn't all this obvious?

When a **class** does not contain any fields, as in `Salt` and `Pepper`, a constructor is included by default.

Yes, that's correct. Default constructors never consume values, and, when used with `new`, always create objects without fields.

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

²⁶ How do we use these constructors?

²⁷ Obvious!

²⁸ 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`.

²⁹ Yes, and its *x* field is 2 and its *y* field is 3.

³⁰ Mostly, but that means we have used constructors before without defining them. How does that work?

³¹ And that's the constructor we used before, right?

³² Good. But what is `new PointD()`?

³³ That makes sense. Let's move on.

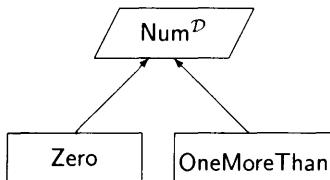
Do the following classes define another datatype with variants?

```
abstract class NumD {}
```

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

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

³⁴ Yes, they define a datatype and two variants.



Draw the picture, too.

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

³⁵ Obviously, just like `new Salt()` is a Seasoning^D.

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

³⁶ Yes, because OneMoreThan constructs a Num^D from a Num^D, and every instance of OneMoreThan is also a Num^D.

How does OneMoreThan do that?

³⁷ We give it `new Zero()`, which is a Num^D, and it constructs a new Num^D.

And what does it mean to construct this new instance?

³⁸ 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?

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

What is

`new OneMoreThan(
 new OneMoreThan(
 new Zero()))?`

⁴⁰ A $\text{Num}^{\mathcal{D}}$, 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)?`

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

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

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

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

Is

`new OneMoreThan(
 new Zero())`

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

like

1?

And what is

`new OneMoreThan(
 new OneMoreThan(
 new OneMoreThan(
 new OneMoreThan(
 new Zero()))))`

⁴⁴ 4.

similar to?

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

⁴⁵ Lots.

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

⁴⁶ No.¹

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

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

⁴⁷ Easy: `new Zero()` is an instance of `Zero` and, by implication, is a `NumD`, 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.

⁴⁸ So are types just names for different collections with no common instances?

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

⁴⁹ 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 `NumD`, which is extended by `Zero`. Indeed, it is of any type that `NumD` extends, too.

⁵⁰ And what is that?

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

⁵¹ This must mean that everything is an `Object`.

Almost. We will soon see what that means.

⁵² 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 {}
```

```
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())?**

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

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

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

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

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

⁵⁶ 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**?

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

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

Correct. Is this a **Layer**^D:

new Base(
 5)?

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

Is this a **Layer**^D:

new Base(
 false)?

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

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

new Base(
 new Integer(5))?

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

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

⁶¹ Easy now:

new Base(
 new Boolean(false))

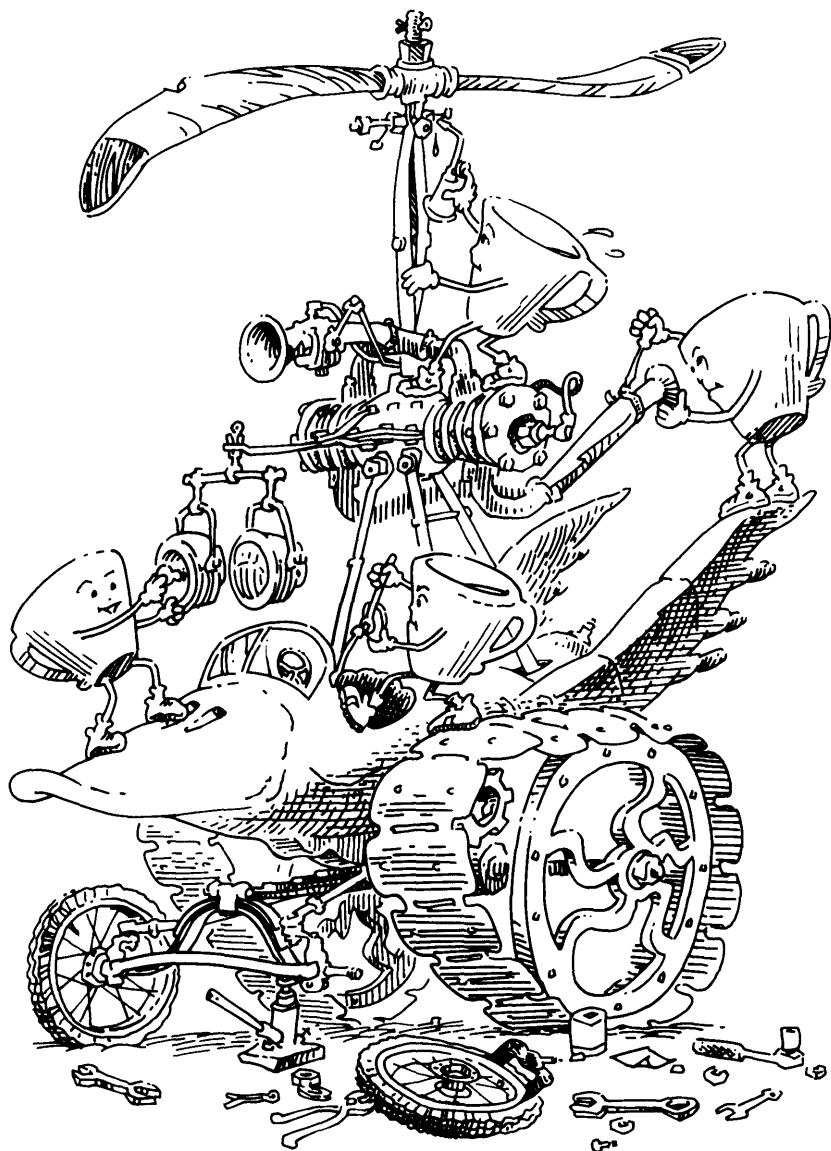
Is it confusing that we need to connect **int** with **Integer** and **boolean** with **Boolean**?

⁶² Too much coffee does that.

Ready for more?

⁶³ Can't wait.

2.
Technicks to
Our Madness



Remember points?

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

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

```
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 ² white space on the right?

It must be for drawing the picture of the classes.

How far is

new ManhattanPt(3,4)

from the Empire State Building?

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

And how far is

new CartesianPt(3,4)

from the origin?

⁴ 5, which is $\sqrt{3^2 + 4^2}$.

Write the methods *distanceToO* using {}, {}, (,), ::, **return**, **int**, +, $\lfloor \sqrt{\cdot} \rfloor$, and \cdot^2 , which determine how far a point is from the origin.

⁵ Of course, you can't write these methods, yet. Okay, you deserve something sweet for enduring this last question.

What do the methods produce?

⁶ **ints**, which represent the distances to the origin.

Here they are.

⁷ They correspond to the unexplained labels in the definition of the datatype and its variants.

abstract int *distanceToO()*:

Point

int *distanceToO()* {
 return $\lfloor \sqrt{x^2 + y^2} \rfloor$ ¹; }

CartesianPt

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

ManhattanPt

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

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

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

⁸ That's simple enough.

How many times have we defined the method *distanceToO*?

⁹ 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?

¹⁰ Yes, they always do.

An **abstract** method in an **abstract** class
introduces an obligation, which says that all
concrete classes that extend this abstract
class¹ must contain a matching method
definition.

¹¹ Okay.

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

What is the value of
`new ManhattanPt(3,4)`
`.distanceToO()`?

¹² 7.

How do we arrive at that value?

¹³ 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()`?

¹⁴ 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?

¹⁵ The largest **int** that does not exceed the
square root of x .

Time for a short break?

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

Here is another datatype with its variants.

¹⁷ It is like 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?

¹⁸ Yes, isn't it for drawing the picture of the classes?

Construct a Shish^D.

¹⁹ How about
new Skewer()?

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

²⁰ Here's one:
new Onion(
new Skewer()).

And a third?

²¹ Here's one more:
new Onion(
new Lamb(
new Onion(
new Skewer()))).

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

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

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

²³ true.

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

²⁴ false.

Is it true that
new Onion(
new Onion(
new Onion(
new Skewer()))))

²⁵ true.

contains only Onions?

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

²⁶ false.

Write the methods `onlyOnions`¹ using `{`, `}`, `(`, `)`, `,`, `:`, `true`, `false`, `return`, and `boolean`.

¹ A better name for these methods would be `nothingButOnions`.

And what do they produce?

Here are the methods.

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?

³⁰ Five, but the first one is **abstract**: the others are concrete.

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

³¹ Yes, we said so.

Does each variant of **Shish^D** contain a method called *onlyOnions*?

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

Is this always the case?

³³ Always.

What do these concrete methods consume?

³⁴ Nothing, just as the **abstract** method says.

What do these concrete methods produce?

³⁵ **booleans**, just as the **abstract** method says.

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

³⁶ **true**.

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

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

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

³⁸ This object is an instance of **Onion**, so we need to use the definition of *onlyOnions* that belongs to the **Onion** variant.

What follows the word **return** in the
onlyOnions method in **Onion**?³⁹

What is the field *s* of the object

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

s.onlyOnions().

⁴⁰ It is

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

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

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

Then what is *s.onlyOnions()*?⁴²

It should be

```
new Onion(  
    new Skewer())  
.onlyOnions(),  
right?
```

Why do we need to know the meaning of

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

⁴³ Because the answer for

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

is also the answer for

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

How do we determine the answer for

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

⁴⁴ Let's see.

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

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

⁴⁵ 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**?

⁴⁶ *s.onlyOnions()*.

What is the field *s* of the object

new Onion(
 new Skewer())?

⁴⁷ **new Skewer()**.

Then what is *s.onlyOnions()*?

⁴⁸ It is
new Skewer()
.onlyOnions(),
just as we would have expected.

Why do we need to know the meaning of

new Skewer()
.onlyOnions()?

⁴⁹ 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()?

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

Is 51 Obviously.
new Skewer()
a
Skewer?

Then what is the answer? 52 true.

Why? 53 Because **true** is what the *onlyOnions* method in **Skewer** always **returns**.

Are we done? 54 Yes! The answer for
new Onion(
 new Onion(
 new Onion(
 new Skewer()))
 .onlyOnions()
)
 .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 55 false, isn't it?
new Onion(
 new Lamb(
 new Skewer()))
 .onlyOnions()?

Which definition of *onlyOnions* must we use 56 This object is an instance of **Onion**, so we to determine the value of
new Onion(
 new Lamb(
 new Skewer()))
 .onlyOnions()?

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

⁵⁷ *s.onlyOnions()*.

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

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

Then what is *s.onlyOnions()*?

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

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

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

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

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

And?

⁶² We use the one that belongs to Lamb.

And now what is the answer?

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

Are we done?

⁶⁴ Yes! The answer for
`new Onion(`
`new Lamb(`
`new Skewer()))`
`.onlyOnions()`

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

which is

`false.`

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

⁶⁵ Here are our words:

“The methods determine for a `ShishD`
whether its contents are edible by an onion
lover.”

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

⁶⁶ Here are our words again:

“For each layer of the `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`?

⁶⁷ Yes.

Is
`new Onion(`
`new Tomato(`
`new Skewer())`)
a `ShishD`?

⁶⁸ 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`?

⁶⁹ Sure.

Is

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

a vegetarian shish kebab?

⁷⁰ Of course, there is no Lamb on it.

And

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

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

Define the methods (*i.e.*, the function)

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.

⁷² 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? ⁷³ Five: one **abstract**, the others concrete.

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

⁷⁴ Yes, they always do.

Does each variant of Shish^D contain a method called *isVegetarian*?

⁷⁵ Yes, because Shish^D contains an **abstract** method called *isVegetarian*.

Is this always the case?

⁷⁶ Always.

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

What do these concrete methods produce? ⁷⁸ **booleans**, just as the **abstract** method says.

The Second Bit of Advice

When writing a function over a datatype, place a method in each of the variants that make up the datatype. If a field of a variant belongs to the same datatype, the method may call the corresponding method of the field in computing the function.

Collect all the pieces of Shish^D . Here is the datatype.

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

⁷⁹ 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 ——————  
}
```

⁸⁰ They define a datatype and four variants that are similar in shape to Shish^D.

```
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?

⁸¹ They are placed onto different Holders.

Here are some holders.

abstract class Rod^D {}

⁸² 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 Rod^D {}

class Sabre extends Rod^D {}

class Sword extends Rod^D {}

Are they good ones?

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

⁸³ We could move all of the food to various forms of plates.

abstract class Plate^D {}

class Gold extends Plate^D {}

class Silver extends Plate^D {}

class Brass extends Plate^D {}

class Copper extends Plate^D {}

class Wood extends Plate^D {}

What is

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

⁸⁴ It's a Kebab^D.

-
- Is
**new Shallot(
 new Radish(
 new Holder(
 new Dagger()))))**
a vegetarian Kebab^D?
- Is
**new Shallot(
 new Radish(
 new Holder(
 new Gold()))))**
a Kebab^D?
- Is
**new Shallot(
 new Radish(
 new Holder(
 new Gold()))))**
a vegetarian kebab?
- ⁸⁵ Sure it is. It only contains radishes and shallots.
- ⁸⁶ Sure, because Gold is a Plate^D, Plate^D is used as a Holder, and radishes and shallots can be put on any Holder.
- ⁸⁷ Sure it is. It is basically like
**new Shallot(
 new Radish(
 new Holder(
 new Dagger()))),**
except that we have moved all the food from a Dagger to a Gold plate.

-
- Let's define the methods (*i.e.*, the function) *isVeggie*, which check whether a kebab contains only vegetarian foods, regardless of what Holder it is on.
- ⁸⁸ If you can, you may rest now.

-
- Write the abstract method *isVeggie*.
- ⁸⁹ That's possible now.
- abstract boolean isVeggie()**
- Kebab
- Of course, *isVeggie* belongs to Kebab^D and *isVegetarian* to Shish^D.

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

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

Holder

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

Shallot

⁹⁰ Except for the names of the methods and fields, the definitions are the same as they were for Shish^D.

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

Shrimp

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

Radish

What is the value of

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

⁹¹ true.

What is

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

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

What is the value of

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

⁹³ It is true, too.

And what is

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

⁹⁴ It is also a Kebab^D, because any kind of Holder will do.

What type of value is

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

⁹⁵ **boolean.**

What type of value is

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

⁹⁶ **boolean.**

What type of value is

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

⁹⁷ **boolean.**

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

⁹⁸ 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()))))?

⁹⁹ All the food is on a Dagger.

What is the holder of

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

¹⁰⁰ All the food is now on a Gold plate.

What is the holder of

new Shallot(
 new Radish(
 new Holder(
 new Integer(52))))?

¹⁰¹ All the food is on an **Integer**.

What is the value of

new Shallot(
 new Radish(
 new Holder(
 new Dagger())))
.whatHolder()?

¹⁰² The dagger.

What is the value of

new Shallot(
 new Radish(
 new Holder(
 new Gold())))
.whatHolder()?

¹⁰³ The gold plate.

What is the value of

new Shallot(
 new Radish(
 new Holder(
 new Integer(52))))
.whatHolder()?

¹⁰⁴ An **Integer**, whose underlying **int** is 52.

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

¹⁰⁵ 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?

¹⁰⁶ 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

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

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

¹⁰⁸ **new Integer(52).**

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

¹⁰⁹ **new Sword().**

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

¹¹⁰ It is *b*.

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

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

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

Holder

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

¹¹² **new Integer(52).**

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

¹¹³ **new Integer(52).**

What is the value of

```
new Shrimp(  
    new Holder(  
        new Integer(52)))  
.whatHolder()?
```

¹¹⁴ **new Integer(52).**

Does that mean that the value of

```
new Radish(  
    new Shallot(  
        new Shrimp(  
            new Holder(  
                new Integer(52)))))  
.whatHolder()
```

¹¹⁵ 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

¹¹⁶ They are all the same.

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

Shrimp

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

Radish

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

Here is the datatype and one of its variants.

```
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; }
```

```
}
```

Collect the remaining variants.

¹¹⁷ There are only three left.

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

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

```
class Shrimp extends KebabD {
```

```
    KebabD k;  
    Shrimp(KebabD _k) {  
        k = _k; }
```

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

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

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

Are there any other **Kebab^D** foods besides Shallot, Shrimp, and Radish?

¹¹⁸ No, these are the only kinds of foods on a **Kebab^D**.

Can we add more foods?

¹¹⁹ Sure. We did something like that when we added Thyme and Sage to Seasoning^D.

Let's define another Kebab^D.

```
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?

Is it obvious how the new methods work?

¹²⁰ A concrete class that extends Kebab^D must define these two methods. That's what the **abstract** specifications say. We can define as many Kebab^D'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(); }  
}
```

And then there were six.

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

Which of these points is closer to the origin:

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

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

new CartesianPt(3,4)
or
new CartesianPt(12,5)?

¹²² Yes, now Kebab^D has six variants.

¹²³ The second one,
because its distance to the origin is 6 while the first point's distance is 7.

¹²⁴ 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  $\lfloor \sqrt{x^2 + y^2} \rfloor$ ; }  
    boolean closerToO(CartesianPt p) {  
        return  
            distanceToO() <= p.distanceToO(); }  
}
```

Add the corresponding method to `ManhattanPt`.

What is the value of
`new ManhattanPt(3.4)`
`.closerToO(new ManhattanPt(1.5))`?

What is the value of
`new ManhattanPt(1.5)`
`.closerToO(new ManhattanPt(3.4))`?

What is the value of
`new CartesianPt(12.5)`
`.closerToO(new CartesianPt(3.4))`?

So finally, what is the value of
`new CartesianPt(3.4)`
`.closerToO(new ManhattanPt(1.5))`?

¹²⁵ 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(); }  
}
```

¹ This is the two character symbol `<=`.

¹²⁶ `false`.

¹²⁷ `true`, obviously.

¹²⁸ `false`, and `true` is the value of
`new CartesianPt(3.4)`
`.closerToO(new CartesianPt(12.5))`.

¹²⁹ That's nonsense.

Why?

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

How can we fix that?

¹³¹ We could replace
(**CartesianPt** *p*)
by
(**Point^D** *p*)
in the definition of *closerToO* for **CartesianPt**.

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

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

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

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

Here is the improved **CartesianPt**.

¹³⁴ Easy.

```
class CartesianPt extends 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(); }  
}
```

```
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**?

¹³⁵ 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();  
}
```

¹³⁶ Looks correct.

¹ 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();  
}
```

¹³⁷ Their fields. Shouldn't we lift them, too?

¹³⁸ 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)⌋; }  
}
```

¹³⁹ 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`.

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

Is this a long chapter?

¹⁴⁰ That's simple.

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

¹⁴² Yes, let's have a short snack break.

3.
Twinkie's News



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; }  
    ——————  
}
```

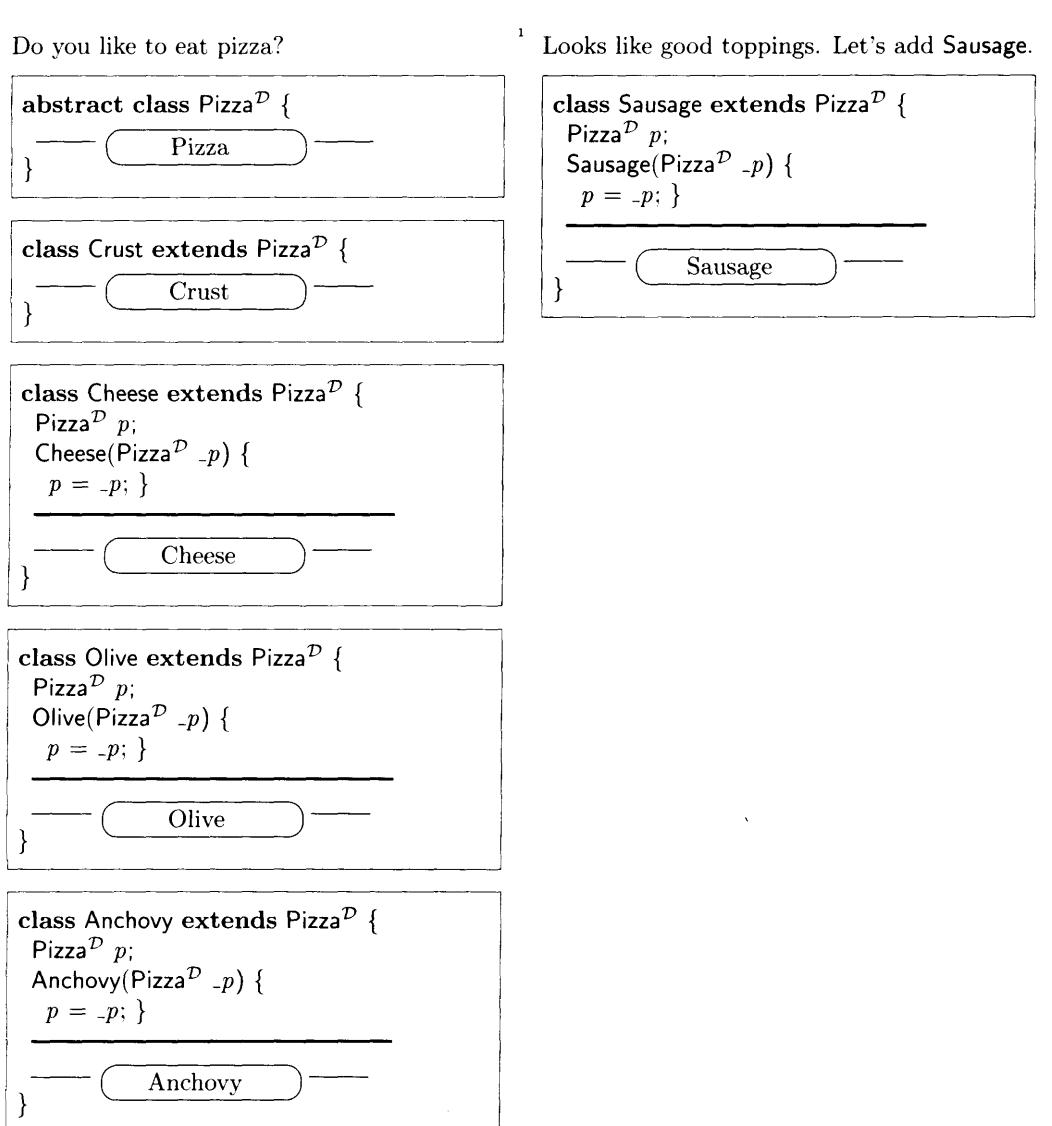
```
class Olive extends PizzaD {  
    PizzaD p;  
    Olive(PizzaD _p) {  
        p = _p; }  
    ——————  
}
```

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

¹ 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()))))).
```

² This looks too salty.

How about removing the anchovies?

Let's remove them. What is the value of

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

³ That would make it less salty.

⁴ It should be a cheese and olive pizza, like this:

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

¹ A better name for these methods would be `removeAnchovy`, but then our definitions wouldn't fit into these columns.

What is the value of

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

⁵ 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 Pizza^P and its variants?

⁶ Yes, and it produces them, too.

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

```
abstract PizzaD remA();
```

Pizza

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

Crust

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

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

Cheese

⁸ The Olive and Sausage methods are similar to the Cheese method.

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

Olive

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

Sausage

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

⁹ 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^D, so let's use new Crust(), the simplest Pizza^D, for the method in Anchovy.

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

Anchovy

¹⁰ 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().
```

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

No problem. Here is an example:

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

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

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

Then, what is the value of

```
new Olive(p.remA())
```

where *p* stands for

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

¹¹ That's easy. The object is an Anchovy. So
the answer is new Crust().

¹² Absolutely, but what if we had more
anchovies?

¹³ That's easy again. As before, the object is an
Anchovy so that the answer must still be
new Crust().

¹⁴ Well, this object is an Olive and its *p* stands
for

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

¹⁵ 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()?

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

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

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

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

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

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

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

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

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

So what is the final answer?

²¹ It is
new Olive(
 new Cheese(
 new Crust())).

Let's try one more example:

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

What kind of pizza should this make?

Check it out!

Doesn't that mean that the result is

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

What about

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

And the answer is

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

Does that mean the final answer is

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

²² It should be a double-cheese pizza.

²³ 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()))).
```

²⁴ Yes, it puts the cheese on whatever we get for

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

²⁵ Now the object is an anchovy.

²⁶ Yes, but we need to add cheese on top.

²⁷ Yes, though it's not the answer we want.

What do we want?

²⁸ 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?

²⁹ The one in Anchovy.

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

Anchovy

Does this *remA* still belong to **Pizza^D**?

³⁰ 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. ³¹ Yes, that would hide their taste, too.

What kind of pizza is

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

³² Easy, it adds cheese on top of each anchovy:

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

¹ A better name for these methods would be **topAnchovywithCheese**.

Did you notice the underlines?

³³ Yes, they show where we added cheese.

And what is

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

Define the remaining methods.

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

Pizza

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

Crust

Take a look at this method.

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

Anchovy

³⁴ Here we don't add any cheese, because the pizza does not contain any anchovies:

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

³⁵ 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

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

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

Anchovy

How many layers of cheese are in the result of

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

How many occurrences of cheese are in the result of

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

Perhaps we should replace every anchovy with cheese.

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

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.

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

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

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

³⁹ We just did something like that.

⁴⁰ Yes, and it does more. Once all the cheese is added, the anchovies are removed.

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*.¹ Here is the abstract method.

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

Pizza

⁴² 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

¹ A better name for these methods would be `substituteAnchovybyCheese`.

Does this skeleton look familiar?

⁴³ Yes, this skeleton looks just like those of *topAwC* and *remA*.

Define the method that belongs in Anchovy.

⁴⁴ Here it is.

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

Anchovy

Collection time.¹

```
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()); }  
}
```

⁴⁵ The classes are getting larger.

```
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()); }  
}
```

¹ This is similar to the *interpreter* and *composite* patterns [4].

Let's add more **Pizza^D** foods.

⁴⁶ Good idea.

Here is one addition: Spinach.

⁴⁷ Yes, we must define three concrete methods for each variant of **Pizza^D**.

```
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 **Pizza^D**, **Crust**, **Cheese**, **Olive**, **Anchovy**, or **Sausage**?

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

Isn't that neat?

⁴⁹ Yes, this is a really flexible way of defining classes and methods. Unfortunately, the more things we want to do with **Pizza^D**s, the more methods we must add.

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

⁵⁰ That would be great, because these definitions are painful to the eye. But we don't know of a better way to organize these definitions yet.

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

⁵¹ Great.

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

⁵² We will stick with anchovies.

This page intentionally left blank

450

Come To Our Carousel



Wasn't this last collection overwhelming?

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

Could it get worse?

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

Why does that become overwhelming?

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

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

⁴ 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; }  
}
```

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

Almost. Each of them corresponds to an *onlyOnions* method in one of the variants of **Shish^D**.

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

Is *onlyOnions* different from **OnlyOnions^V**?

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

And that's the whole point.

⁷ What point?

We want all the methods in one class.

⁸ What methods?

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

⁹ 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.

¹⁰ It's about time.

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

¹¹ Everything following **return** is the same.

Right. What is the difference?

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

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

¹³ Indeed, they are.

It is time to discuss the boring part.

¹⁴ What boring part?

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

¹⁵ True, we still don't know how to make **Shish^D** 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();  
}
```

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

```
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); }  
}
```

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

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

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

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

Explain *s.onlyOnions()*.

Explain *ooFn.forOnion(s)*.

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

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

¹⁷ If "consume" refers to what follows the name between parentheses, the method consumes *s*, which is the rest of the shish.

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

¹⁹ 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.

²⁰ That makes sense and explains *s.onlyOnions()*.

²¹ Here are our words:
"s is a **Shish $^{\mathcal{D}}$** , and therefore *s.onlyOnions()* determines whether what is below the onion is also edible by an onion lover."

²² 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 $^{\mathcal{D}}$** , and *s* is the **Shish $^{\mathcal{D}}$** that represents what is below the onion."

²³ It is still true.

²⁴ 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 <i>forOnion</i> method do?	²⁵ It checks whether the rest of this shish is edible by onion lovers.
How does it do that?	²⁶ It uses the method <i>onlyOnions</i> on <i>s</i> .
Isn't that where we started from?	²⁷ Yes, we're going round and round.
Welcome to the carousel.	²⁸ Fortunately, the shish shrinks as it goes around, and when we get to the skewer we stop.
And then the ride is over and we know that for this example the answer is true.	²⁹ That's exactly it.
Do we need to remember that we are on a carousel?	³⁰ 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?	³¹ No, let's look at some of the other actions on shishes and pizzas.

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

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

¹ *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!

³³ 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.

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

How do we define the visitor?

³⁵ Does that refer to the class that contains the actions?

Yes, that one. Define the visitor.

³⁶ 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?

³⁷ 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?

³⁸ 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?

³⁹ No, it's a pizza.

```
abstract class PizzaD {}
```

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

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

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

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

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

So what do we do next?

⁴⁰ 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.

⁴¹ 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.

- ⁴² Is it time to define the visitors that correspond to the methods *remA*, *topAwC*, and *subAbC*?

Okay, here is *RemAV*.

```
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 *TopAwCV*.

- ⁴³ 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()); }  
}
```

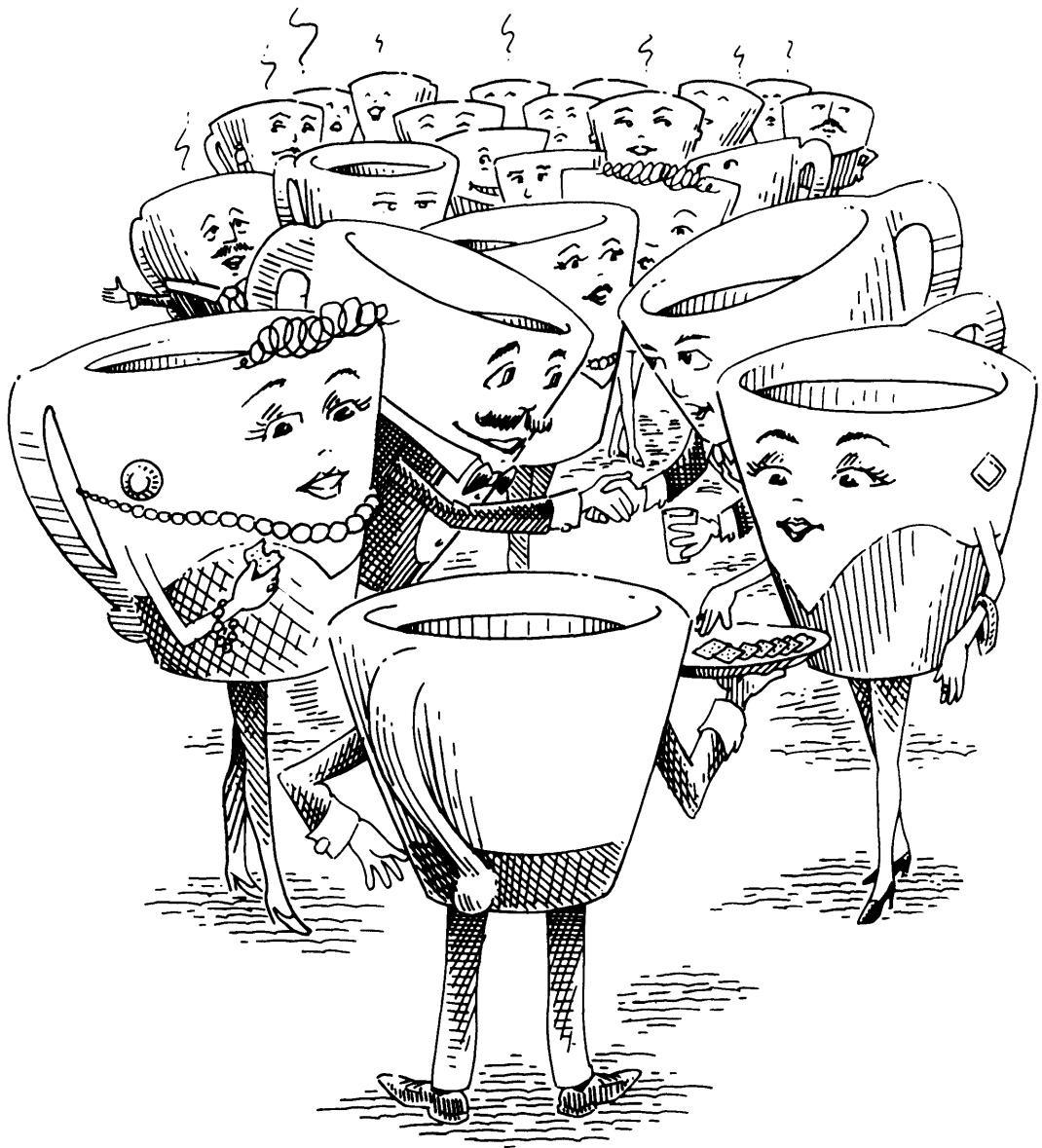
The last one, *SubAbCV*, is a piece of cake.

- ⁴⁴ 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()); }  
}
```

This page intentionally left blank

5.
Objects Are
People, Too



Have we seen this kind of definition before?¹ 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 ——————  
}
```

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

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

² 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?

³ 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?

What is the value of

**new Top(new Salmon(),
new Top(new Anchovy(),
new Top(new Tuna(),
new Top(new Anchovy(),
new Bot()))))**

.remA()?

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()))?**

Does *remA* belong to Pie^D ?

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

**RemA $^\mathcal{V}$ raFn = new RemA $^\mathcal{V}()$;
abstract Pie D remA();**

Pie

⁴ 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()))))**.

⁵ It is this fishy pizza pie:

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

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

⁷ Yes, and it produces pizza pies.

⁸ This is easy by now.

**Pie D remA() {
return raFn.forBot(); }**

Bot

**Pie D remA() {
return raFn.forTop(t,r); }**

Top

Great. Isn't that easy?

⁹ Easy and boring.

What part of this exercise differs from datatype to datatype?

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

Anything else?

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

Why (*t,r*)?

¹² Because these are the fields of **Top**.

Let's define the visitor **RemA^V**.

¹³ 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?

¹⁴ We guess:

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

And what does

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

¹⁵ We could guess:

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

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

¹⁶ What?

What is the value of

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

¹⁷ The “Not yet.” implies that the value is `false`.

Yes! And what is the value of

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

¹⁸ `false`,

because no anchovy is a tuna.

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

¹⁹ If we know that `equals`'s answer is always `false`, why bother to use it?

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

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

²⁰ Okay. How?

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

```
abstract class FishD {
```

```
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); }
}
```

²¹ Assuming that

`(o instanceof Tuna)`

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

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

Aren't they? Is every value constructed with **new** an instance of **Object**? ²² 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**? ²³ Yes, and of the class that **B** extends and so on.

Now, what is the value of
new Anchovy().equals(new Anchovy())? ²⁴ **true**,
because **new Anchovy()** is an instance of Anchovy.

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

Could we have written **RemA^V** without using **equals**? ²⁶ 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^V** so that it works for any kind of fish topping. ²⁷ We can do that, but when we use the methods of the more general visitor, we need to say which kind of fish we want to remove.

What are good names for the more general methods and visitor? ²⁸ How about **remFish** and **RemFish^V**?

How do we use *remFish*?

²⁹ We give it a Fish^D .

Add the protocol for RemFish^V . We designed the abstract portion.

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

Pie

³⁰ 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?

³¹ The f stands for the Fish^D we want to remove in both cases. The t and the r are the fields of **Top**; **Bot** doesn't have any.

Let's define RemFish^V and its two methods.

³² Instead of comparing the top layer t of the pizza to **Anchovy**, we now determine whether it equals the $\text{Fish}^D f$, 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 RemFish^V ?

³³ 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()).
```

Yes. What values does *forTop* consume?

And now?

So?

What is the value of

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

What does *forBot* in *RemFish^V* produce?

All clear?

³⁴ The object is a topping, so we use *forTop* from *RemFish^V*.

³⁵ It consumes three values: **new Anchovy()**, which is *t*, the top-most layer of the pizza; **new Bot()**, which is *r*, the rest of the pizza; and **new Anchovy()**, which is *f*, the Fish^D to be removed.

³⁶ 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.

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

³⁸ It is the same as

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

³⁹ It produces **new Bot()**, no matter what *f* is.

⁴⁰ 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?

What does *remInt* do?

⁴¹ Yes, it looks like what we just evaluated.

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

⁴² The Machine decided

```
new Integer(0).equals(new Integer(0))  
to be true, and the rest was obvious.
```

Define the visitor **RemInt**^V.

⁴³ Wonderful! We do the interesting thing first. This visitor is almost identical to **RemFish**^V. We just need to change the type of what the two methods consume.

```
class RemIntV {  
    PieD forBot(Integer i) {  
        return new Bot(); }  
    PieD forTop(Object t,PieD 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*?

⁴⁵ 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?

⁴⁶ It is so simple, let's save it for later.

Can we remove **Integers** from **Pie^Ds**?

⁴⁷ Yes.

Can we remove **Fish^D** from **Pie^Ds**?

⁴⁸ Yes, and we use nearly identical definitions.

Let's combine the two definitions.

⁴⁹ If we use **Object** instead of the underlined **Integer** above, everything works out.

Why?

⁵⁰ Because everything constructed with **new** is an **Object**.

Just do it!

⁵¹ It's done.

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

⁵² Now?

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

⁵³ Let's just consider **Rem^V**.

Why not **RemFish^V** and **RemA^V** and **RemInt^V**?

⁵⁴ They are unnecessary once we have **Rem^V**.

Here is the **abstract** portion of Pie^D .

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

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)).
```

⁵⁶ Works like a charm with the same result as before.

And how about

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

⁵⁷ Ditto.

Next:

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

⁵⁸ 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^D (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; }  
    }
```

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

⁶⁰ We expected it to remove new Zero() from the pizza.

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

⁶² Unless we define it anew for those classes whose instances we wish to compare.

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

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

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

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

No. What is the type of <i>o</i> ?	⁶⁴ Object, according to (Object <i>o</i>), which is what declares the type of <i>o</i> .
So what is <i>o.predecessor</i> ?	⁶⁵ Nonsense.
Correct. What do we know after if has determined that (<i>o instanceof OneMoreThan</i>) is true?	⁶⁶ We know that <i>o</i> 's type is Object and that it is an instance of OneMoreThan.
Precisely. So what does ((OneMoreThan) <i>o</i>) do?	⁶⁷ It converts the type of <i>o</i> from Object to OneMoreThan.
What is ((OneMoreThan) <i>o</i>)'s type?	⁶⁸ Its type is OneMoreThan, and now it makes sense to write ((OneMoreThan) <i>o</i>). <i>predecessor</i> .
Are <i>o</i> and ((OneMoreThan) <i>o</i>) interchangeable?	⁶⁹ 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?	⁷⁰ Someone has been drinking too much coffee.
Did you also notice the <i>predecessor</i> .equals(((OneMoreThan) <i>o</i>). <i>predecessor</i>) in <i>equals</i> for OneMoreThan?	⁷¹ How do the two uses of <i>predecessor</i> differ?
The first one, <i>predecessor</i> , refers to the <i>predecessor</i> field of the instance of OneMoreThan on which we are using <i>equals</i> . And that field might not be a OneMoreThan.	⁷² So the second one, ((OneMoreThan) <i>o</i>). <i>predecessor</i> , refers to the <i>predecessor</i> field of the instance of OneMoreThan consumed by <i>equals</i> .

Yes. Are these two objects equal?

⁷³ If they are similar¹ to the same **int**, they are equal. But most of the time, they are not.

¹ Check chapter 1 for “similar.”

Time for lunch?

⁷⁴ That's just in time.

Did you have a good lunch break?

⁷⁵ Yes, thank you.

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

⁷⁶ Now we get
new Top(new Anchovy(),
new Top(new Integer(3),
new Bot()))),
which is precisely what we want.

And why?

⁷⁷ Because *equals* now knows how to compare
Num^Ds.

Do we always add *equals* to a class?

⁷⁸ No, only if we need it.

Do we need *equals* when we want to
substitute one item for another on a pizza
pie?

⁷⁹ Yes, we do.

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

⁸⁰ 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? ⁸¹ It consumes two *fish* and works on Pie^Ds.

And what does it produce?

⁸² It always produces a Pie^D .

What is the value of

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

⁸³ It is the same pizza pie with all 3s replaced by 5s:

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

What kind of values does substInt consume? ⁸⁴ It consumes two **Integers** and works on Pie^D s.

And what does it produce?

⁸⁵ It always produces a Pie^D .

We can define 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 SubstInt^V .

⁸⁶ To get from SubstFish^V to SubstInt^V , we just need to substitute Fish^D by Integer everywhere and ‘Fish’ by “Int” in the class and method names.

```
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?

⁸⁷ Yes, because there is obviously a more general version like Rem^V .

Yes, we call it Subst^V . Define it.

⁸⁸ We substitute Object for Fish D and 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 Subst^V ⁸⁹ to Pie^D . Here are the variants.

The abstract part is obvious.

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

```
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;
    }

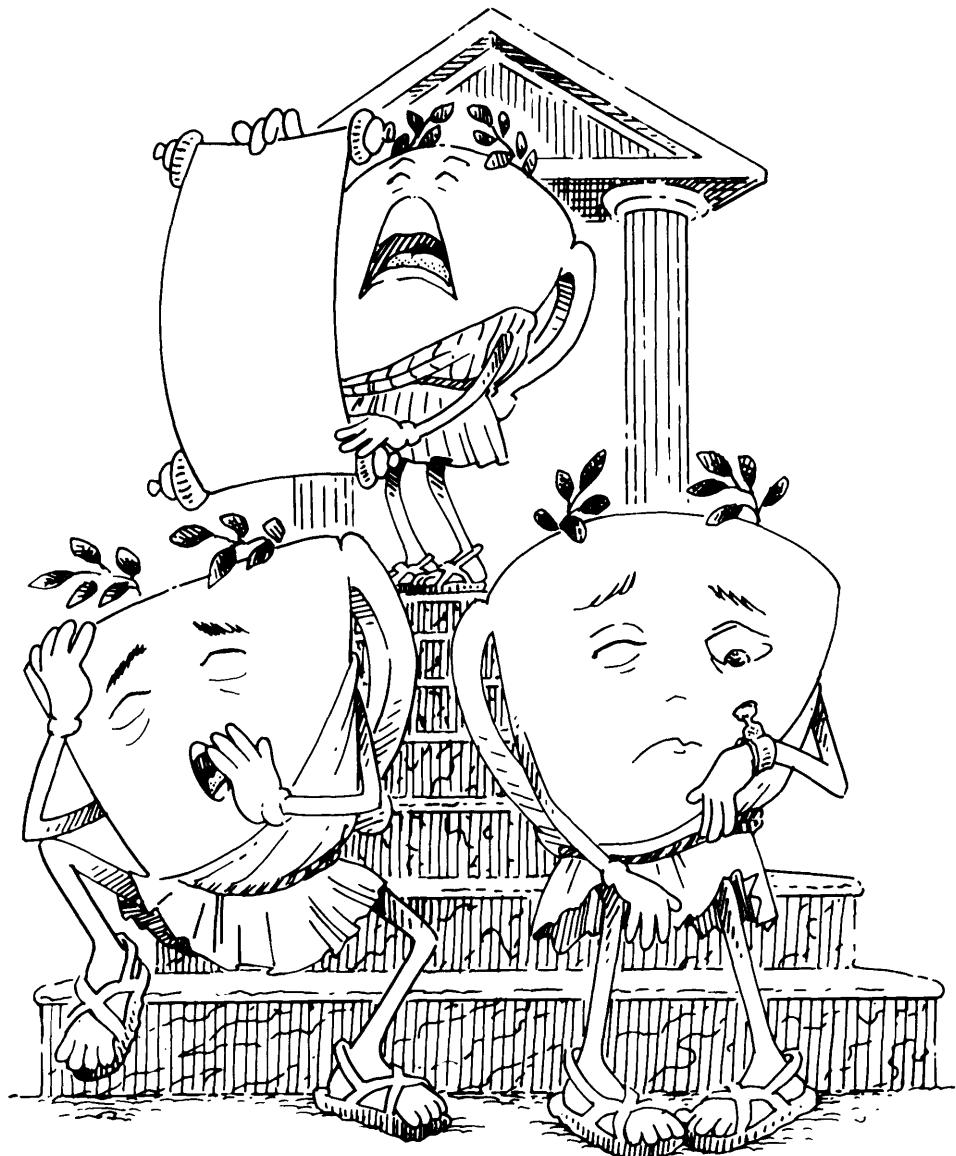
    PieD rem(Object o) {
        return remFn.forTop(t, r, o);
    }
    PieD subst(Object n, Object o) {
        return substFn.forTop(t, r, n, o);
    }
}
```

So?

⁹⁰ That was some heavy lifting.

6.

Boiling Protocols



Are protocols truly boring?

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

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

What is the difference between *rem* and *subst* in *Pie^D*?

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

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

¹ We acted as if they were.

² 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); }  
}
```

³ The first one consumes one Object, the second one consumes two.

⁴ 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.

⁵ 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 the⁶ 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**?

⁶ But *remFn* and *substFn* defined in the datatype are still a bit mysterious.

⁷ 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?

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

⁹ 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.

Like what?

Where are they used?

Yes. Here is Rem^V .

```
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^V accordingly.

What is **this** all about?

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

How did we get here?

¹⁰ We still have some work to do.

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

¹² In Rem^V and Subst^V , the interesting parts, for example.

¹³ 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));
    }
}
```

¹⁴ Yes, what about it. Copying is easy.

¹⁵ Which object?

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

How does that happen?

¹⁷ It happens with
remFn.forBot(…)
and
remFn.forTop(…),
respectively.

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

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

That's it. Tricky?

¹⁹ Not really, just self-referential.

Why?

²⁰ Because **this** is a 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 Rem^V(),
new Zero())?

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

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

²² It creates a Rem^V object, which corresponds to the *remFn* in the old 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 Subst^V(),
new Integer(5),
new Integer(3))?

²³ We did the same example in the preceding chapter, and the result remains the same.

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

²⁴ It creates a `SubstV` object, which corresponds to the `remFn` in the old `PieD`.

So what is the underlined part about?

²⁵ We changed the methods in `PieD`, which means that we must also change how it is used.

Ready for the next protocol?

²⁶ Let's grab a quick snack.

How about some ice cream?

²⁷ Cappuccino crunch sounds great. The more coffee, the better.

Take a look at `subst` in `Top` and at `forTop` in `SubstV`. What happens to the values that they consume?

²⁸ Nothing really. They get handed back and forth, though `forTop` compares `o` to `t`.

Is the handing back and forth necessary?

²⁹ We don't know any better way, yet.

Here is a way to define `SubstV` that avoids the handing back and forth of these extra values.

³⁰ Wow. This visitor has two fields.¹

```
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));
    }
}
```

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

How do we create a Subst^V ?

³¹ We use

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

What does that do?

³² It creates a Subst^V whose methods know how to substitute **new Integer(5)** for all occurrences of **new Integer(3)** in Pie^D .

How do the methods know that without consuming more values?

³³ The values have now become fields of the 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())))?

³⁴ 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?

³⁵ 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?

³⁶ 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();
    }
}
```

Is there anything else missing?

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

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

Can we eliminate the differences?

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

³⁷ 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);
}
```

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

³⁹ Not much. The name of the respective values they consume and the corresponding types.

⁴⁰ Not much. The name of the respective values they consume and the corresponding types.

⁴¹ 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);
}.
```

⁴² Both Rem^V and Subst^V are visitors that contain the same method names and those methods consume and produce the same types of values. We can think of them as extensions of a common **abstract class**.

Yes! Do it!

⁴³ 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);  
}
```

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

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

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

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

⁴⁵ Fine.

⁴⁶ 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?

⁴⁷ 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?

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

And what is a better name for *puFn*?

⁴⁹ Easy: *ask*, because we ask for services.

Now we can simplify the protocol. Here is the new *Rem*^V.

```
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.

⁵⁰ 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**?

⁵¹ 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.

⁵² Why?

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

⁵³ Looks weird, but let's move on.

Correct. They are just icing.

⁵⁴ Okay, we still won't forget them.

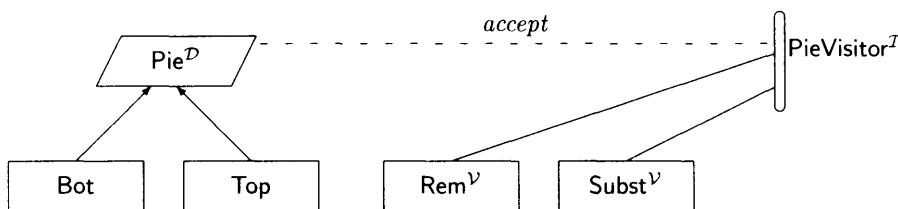
Now define the new Subst^V .

⁵⁵ 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 PieVisitor^I and all the classes: Pie^D , Bot , Top , Rem^V , and Subst^V .

⁵⁶ Here is our picture.



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

⁵⁷ The Subst^V visitor **implements** PieVisitor^I , it doesn't *extend* it. Arrows mean “extends,” lines mean “implements.”

And the dashed line?

⁵⁸ 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()))?
```

⁵⁹ 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 LtdSubst^V produces.¹

¹ A better name is `LimitedSubstitutionV`, and that is how we pronounce it.

Good. Define LtdSubst^V .

⁶⁰ The methods of LtdSubst^V replace one fish on a pie by another as many times as specified by the first value consumed by LtdSubst^V .

⁶¹ 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 LtdSubstV(2,  
                           new Salmon(),  
                           new Anchovy()))?
```

⁶² 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()))))).
```

How come?

⁶³ Because *c*, the counting field, never changes.

Why doesn't *c* ever change?

⁶⁴ Because **this**, the *LtdSubst^V* that performs the substitutions, never changes.

Can we fix **this**?

⁶⁵ We can't change **this**, but we can replace **this** with a new *LtdSubst^V* that reflects the change.

If *c* stands for the current count, how do we create a *LtdSubst^V* that shows that we have just substituted one fish by another.

⁶⁶ Simple, we use
new LtdSubst^V(c - 1, n, o)
in place of **this**.

The Sixth Bit of Advice

When the additional consumed values change for a self-referential use of a visitor, don't forget to create a new visitor.

Define the new and improved version of LtdSubst^V .

⁶⁷ 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));
    }
}
```

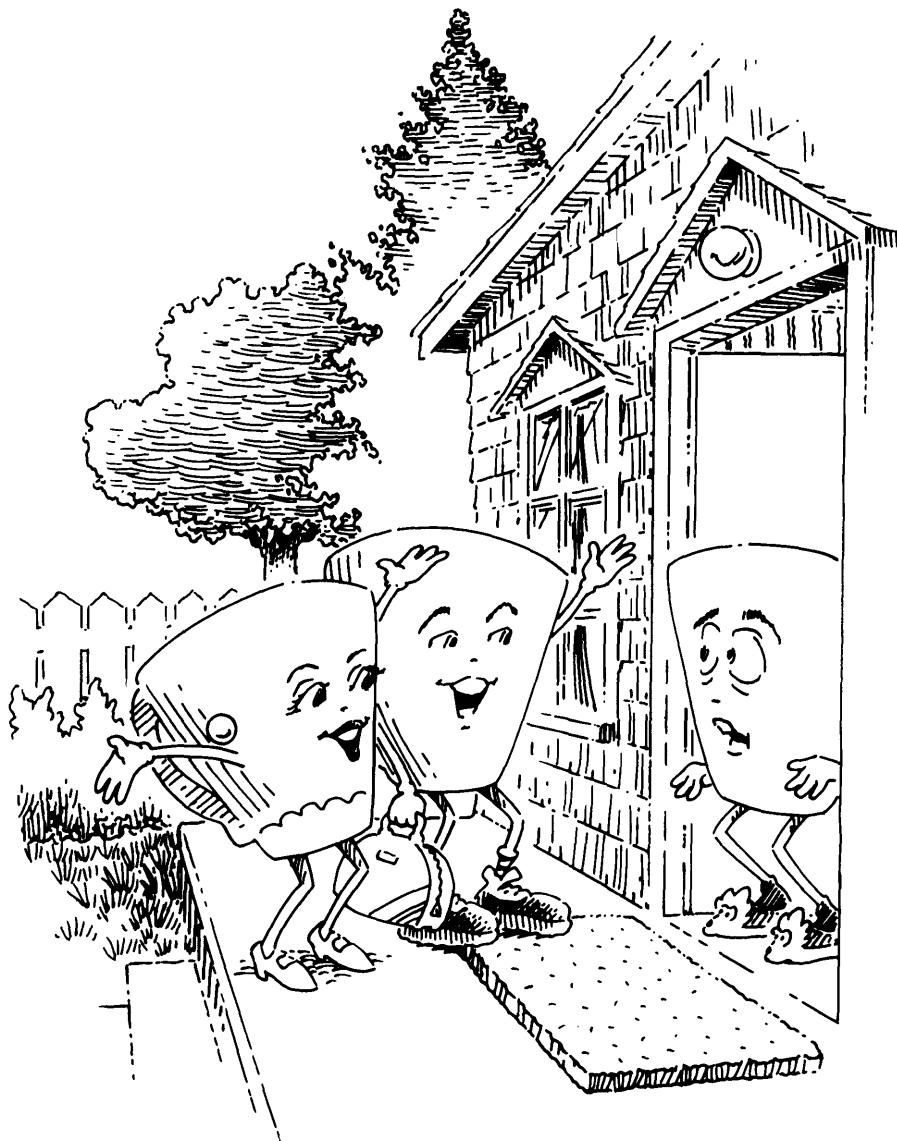
How does
 this
differ from
 new LtdSubst^V(c - 1, n, o)?

⁶⁸ They are two different LtdSubst^V s. One replaces c occurrences of o by n in a pizza pie, and the other one replaces only $c - 1$ of them.

How do you feel about protocols now?

⁶⁹ They are exciting. Let's do more.

Zo
Oh My !



Is

**new Flat(new Apple(),
new Flat(new Peach(),
new Bud()))**

a flat Tree^D?

¹ Yes.

Is

**new Flat(new Pear(),
new Bud())**

a flat Tree^D?

² Yes, it is also a flat Tree^D.

And how about

**new Split(
new Bud(),
new Flat(new Fig()),
new Split(
new Bud(),
new Bud()))))**?

³ No, it is split, so it can't be flat.

Here is one more example:

**new Split(
new Split(
new Bud(),
new Flat(new Lemon(),
new Bud()))),
new Flat(new Fig()),
new Split(
new Bud(),
new Bud()))).**

Is it flat?

⁴ No, it isn't flat either.

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

⁵ Unless there is anything else to Tree^D, it's totally clear.

Good. Then let's move on.

⁶ 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 Tree^D's are either flat, split, or bud. Formulate a rigorous description for Tree^D's.

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

Do Tree^D's variants contain *equals*?

⁷ 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; }  
}
```

⁸ That probably means that we will need to compare fruits and other things.

⁹ No, which means we won't compare them, but we could.

How does the datatype Tree^D differ from all the other datatypes we have seen before?

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

```
interface bTreeVisitor $\mathcal{I}$  {  
    boolean forBud();  
    boolean forFlat( $\text{Tree}^D f$ ,  $\text{Tree}^D t$ );  
    boolean forSplit( $\text{Tree}^D l$ ,  $\text{Tree}^D r$ );  
}
```

Here is the new datatype definition.

```
abstract class Tree $D$  {  
    abstract  
    boolean accept(bTreeVisitor $\mathcal{I}$  ask);  
}
```

Revise the variants.

¹⁰ The name of the new datatype occurs twice in its Split variant.

¹¹ That just means extending what we have with one method each.

```
class Bud extends Tree $D$  {  
    boolean accept(bTreeVisitor $\mathcal{I}$  ask) {  
        return ask.forBud();  
    }  
}
```

```
class Flat extends Tree $D$  {  
    Fruit $D$  f;  
    Tree $D$  t;  
    Flat(Fruit $D$  _f, Tree $D$  _t) {  
        f = _f;  
        t = _t; }  
}
```

```
boolean accept(bTreeVisitor $\mathcal{I}$  ask) {  
    return ask.forFlat(f,t);  
}
```

```
class Split extends Tree $D$  {  
    Tree $D$  l;  
    Tree $D$  r;  
    Split(Tree $D$  _l, Tree $D$  _r) {  
        l = _l;  
        r = _r; }  
}
```

```
boolean accept(bTreeVisitor $\mathcal{I}$  ask) {  
    return ask.forSplit(l,r);  
}
```

But isn't $\text{bTreeVisitor}^{\mathcal{I}}$ 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**.

¹² Okay.

How many methods does the definition of `bIsFlatV` contain, assuming it implements `bTreeVisitorT`?

¹³ 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?

¹⁴ `booleans`.

What visitor does `bIsFlatV` remind us of?

¹⁵ `OnlyOnionsV`.

Here is a skeleton for `bIsFlatV`.

```
class bIsFlatV implements bTreeVisitorT {  
    public  
        boolean forBud() {  
            return _____; }  
    public  
        boolean forFlat(FruitD f,TreeD t) {  
            return _____; }  
    public  
        boolean forSplit(TreeD l,TreeD r) {  
            return _____; }  
}
```

¹⁶ That's easy now.

```
class bIsFlatV implements bTreeVisitorT {  
    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.

¹⁷ Here is the easy part.

```
class bIsSplitV implements bTreeVisitorT {  
    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?

¹⁸ We need to check whether both l and r are split trees.

Isn't that easy?

¹⁹ Yes, we just use the methods of `bIsSplitV` on l and r .

And then?

²⁰ Then we need to know that both are true.

If

`l.accept(this)`

is true, do we need to know whether

`r.accept(this)`

is true?

²¹ Yes, because if both are true, we have a split tree.

If

`l.accept(this)`

is false, do we need to know whether

`r.accept(this)`

is true?

²² No, then the answer is false.

Finish the definition of `bIsSplitV` using

```
if (...)  
    return ...  
else  
    return ....
```

²³ Now we can do it.

```
class bIsSplitV 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; }  
}
```

¹ We could have written the if... as
`return l.accept(this) && r.accept(this).`

Give an example of a Tree^D for which the methods of bIsSplit^V respond with true.

²⁴ There is a trivial one:
new Bud().

How about one with five uses of Split?

²⁵ 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 Tree^D have any fruit?

²⁶ No.

Define the bHasFruit^V visitor.

²⁷ Here it is.

```
class bHasFruitV
    implements bTreeVisitorI {
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); }
}
```

¹ We could have written the if ... as
`return l.accept(this) || r.accept(this)`.

What is the height of

²⁸ 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

²⁹ 2.

```
new Split(  
  new Bud(),  
  new Flat(new Lemon(),  
    new Bud()))?
```

What is the height of

³⁰ 1.

```
new Flat(new Lemon(),  
  new Bud())?
```

What is the height of

³¹ 0.

```
new Bud()?
```

So what is the height of a Tree^D?

³² Just as in nature, the height of a tree is the distance from the beginning to the highest bud in the tree.

Do the methods of iHeight^V work on a Tree^D? ³³ Yes, and they produce an int.

Is that what the i in front of Height is all about?

³⁴ It looks like i stands for int, doesn't it?

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 iHeightV())?
```

³⁵ 4.

Why is the height 4?

³⁶ Because the value of
new Split(
 new Bud(),
 new Bud())
.accept(new iHeight^V())

is 1; the value of

```
new Flat(new Fig(),  
    new Flat(new Lemon(),  
        new Flat(new Apple(),  
            new Bud()))))  
.accept(new iHeightV())
```

is 3; and the larger of the two numbers is 3.

And how do we get from 3 to 4?

³⁷ We need to add one to the larger of the numbers so that we don't forget that the original Tree^D was constructed with Split and those two Tree^Ds.

□ picks the larger of two numbers, x and y .¹ ³⁸ Oh, that's nice. What kind of methods does iHeight^V define?

¹ When you enter this in a file, use
Math.max(x,y).

Math is a class that contains max as a (static) method.

iHeight^V's methods measure the heights of the Tree^Ds to which they correspond.

³⁹ Now that's a problem.

Why?

- ⁴⁰ We defined only **interfaces** that produce **booleans** in this chapter.
-

So what?

- ⁴¹ The methods of $i\text{Height}^V$ produce **ints**, which are not **booleans**.
-

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

- ⁴² 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 Tree^D .

```
abstract class TreeD {  
    abstract  
    boolean accept(bTreeVisitorT ask);  
    abstract  
    int accept(iTreeVisitorT ask);  
}
```

We can have two methods with the same name in the same class as long as the types of the things they consume are distinct.

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

- ⁴³ Does that mean we can have two methods with the same name in one class?¹

¹ In Java, defining multiple methods with the same name and different input types is called "overloading."

- ⁴⁴ $b\text{TreeVisitor}^T$ is indeed different from $i\text{TreeVisitor}^T$, so we can have two versions of *accept* in Tree^D .
-

- ⁴⁵ 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); }  
}
```

⁴⁶ 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^V*.

```
class iHeightV implements iTreeVisitorT {  
    public int forBud() {  
        return _____; }  
    public int forFlat(FruitD f,TreeD t) {  
        return _____; }  
    public int forSplit(TreeD l,TreeD r) {  
        return _____; }  
}
```

⁴⁷ 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)  $\sqcup$  r.accept(this))  
            + 1; }  
}
```

Complete these methods.

What is the value of

```
new Split(  
    new Bud(),  
    new Bud())  
.accept(new iHeightV())?
```

⁴⁸ 1, of course.

And why is it 1?

⁴⁹ 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?

Good job. How about the datatype Tree^D .

⁵⁰ 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())))).
```

⁵¹ It's like SubstFish^V and SubstInt^V from the end of chapter 5, but we can't do it just yet.

⁵² Its methods produce Tree^D 's, neither **ints** nor **booleans**, which means that we need to add yet another interface.

```
interface tTreeVisitorT {  
    TreeD forBud();  
    TreeD forFlat(FruitD f, TreeD t);  
    TreeD forSplit(TreeD l, TreeD r);  
}
```

⁵³ Easy. Here is the abstract one.

```
abstract class TreeD {  
    abstract  
    boolean accept(bTreeVisitorT ask);  
    abstract  
    int accept(iTreeVisitorT ask);  
    abstract  
    TreeD accept(tTreeVisitorT ask);  
}
```

Define the variants of Tree^D .

⁵⁴ No problem.

```
class Bud extends TreeD {  
    boolean accept(bTreeVisitorT ask) {  
        return ask.forBud(); }  
    int accept(iTreeVisitorT ask) {  
        return ask.forBud(); }  
    TreeD accept(tTreeVisitorT 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); }  
int accept(iTreeVisitorT ask) {  
    return ask.forFlat(f,t); }  
TreeD accept(tTreeVisitorT 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); }  
int accept(iTreeVisitorT ask) {  
    return ask.forSplit(l,r); }  
TreeD accept(tTreeVisitorT ask) {  
    return ask.forSplit(l,r); }  
}
```

Then define $tSubst^V$.

⁵⁵ 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.

⁵⁶ 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?

⁵⁷ We prefer coconut sorbet.

Is it disturbing that we have three nearly identical versions of *accept* in *Tree^D*'s and its variants?

⁵⁸ 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?

⁵⁹ If **boolean** and **int** were classes, we could use **Object** for **boolean**, **int**, and *Tree^D*. Unfortunately, they are not.

Remember **Integer** and **Boolean**? They make it possible.

⁶⁰ Yes, **Boolean** is the class that corresponds to **boolean**, and **Integer** corresponds to **int**.

Here is the **interface** for a protocol that produces **Object** in place of **boolean**, **int**, and *Tree^D*.

```
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^D*.

⁶¹ 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`.

⁶² 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`?

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); }  
}
```

Will the conversion always work?

⁶³ 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.

⁶⁴ Oh, because `l.accept(this)` produces an Object, we must first convert¹ 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`.

¹ If Java had parametric polymorphism for methods, no downward cast would be necessary for our visitors (Martin Odersky and Philip Wadler, *Pizza into Java: Translating Theory into Practice, Conference Record on Principles of Programming Languages*, 146–159. Paris, 1997).

⁶⁵ 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.

⁶⁶ Oh my!

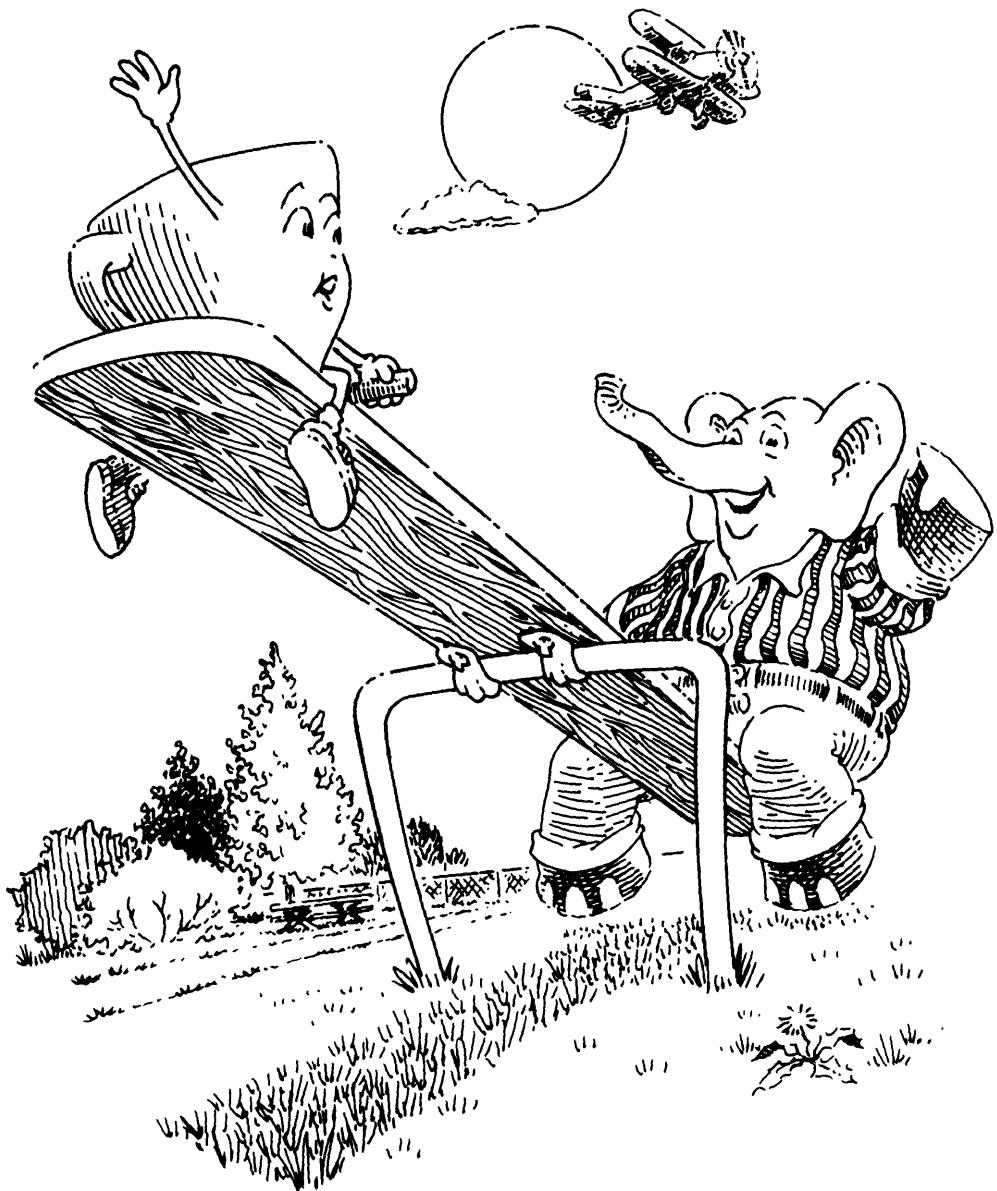
```
class OccursV implements TreeVisitorT {
    FruitD a;
    OccursV(FruitD _a) {
        a = _a; }

    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()): }

}
```

This page intentionally left blank

86
Like Father
Like Son



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

¹ 12.

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

² 12,

because we have just rewritten the previous expression with prefix operators.

What is the value of
`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))))?`

³ `new Integer(12)`,

because we have just rewritten the previous expression using `Integer` and constructors.

Where do the constructors come from?

⁴ A datatype and its variants that represent arithmetic expressions.

Did you like that?

⁵ So far, so good.

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

⁶ $\{7, 5\}$.

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

⁷ $\{7, 5\}$,

we just went from infix to prefix notation.

What is the value of
 $(+ \{7, 5\} (\times (- \{4\} \{3\}) \{5\}))?$

⁸ $\{7, 5\}$,

we just renamed the operators.

What is the value of

```
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))))?
```

⁹ **new Empty()**
.add(new Integer(7))
.add(new Integer(5)),
because we have just rewritten the
previous expression using the constructors.

Where do the constructors come from?

¹⁰ A datatype and its variants that represent
set expressions.

Do you still like it?

¹¹ Sure, why not.

Does the arithmetic expression look like the
set expression?

¹² Yes, they look the same except for the
constants:
new Plus(
 new Const(•),
 new Prod(
 new Diff(
 new Const(•),
 new Const(•)),
 new Const(•))).

Let's say that an expression is either

a Plus($expr_1, expr_2$),
a Diff($expr_1, expr_2$),
a Prod($expr_1, expr_2$), or
a constant,

where $expr_1$ and $expr_2$ stand for arbitrary
expressions. What should be the visitor
interface?

¹³ 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(ExprVisitorI 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(ExprVisitorI 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(ExprVisitorI 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(ExprVisitorI ask) {  
    return ask.forProd(l,r); }  
}
```

```
class Const extends ExprD {  
    Object c;  
    Const(Object _c) {  
        c = _c; }  
}
```

```
Object accept(ExprVisitorI ask) {  
    return ask.forConst(c); }  
}
```

Can we now define a visitor whose methods determine the value of an arithmetic expression?

How do we add

new Integer(3)
and
new Integer(2)?

But what is the result of
new Integer(3).intValue()
+
new Integer(2).intValue()?

How do we turn that into an **Integer**?

Okay, so here is a skeleton of **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; }
}
```

¹⁵ Yes, we can. It must have four methods, one per variant, and it is like **Occurs^V** from the previous chapter.

¹⁶ We have done this before. We use the method *intValue* to determine the **ints** that correspond to the **Integers**, and then add them together.

¹⁷ An **int**, what else?

¹⁸ We use **new Integer(...)**.

¹⁹ That's an interesting skeleton. It contains five different kinds of blanks and two of them occur three times each. But we can see the bones only. Where is the beef?

How does <i>forPlus</i> work?	²⁰ It consumes two Expr^D s, determines their respective values, and <i>pluses</i> them.
How are the values represented?	²¹ As Objects , because we are using our most general kind of (and most recent) visitor.
So what kind of values must <i>plus</i> consume?	²² Objects , because that's what <i>l.accept(this)</i> and <i>r.accept(this)</i> produce.
What must we put in the first and second blanks?	²³ Object .
Can we add Objects ?	²⁴ No, we must convert them to Integers first and extract their underlying ints .
Can we convert all Objects to Integers ?	²⁵ No, but all Objects produced by IntEval^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^V())?	²⁶ Wow. At some level, this is nonsense.
Correct, so sometimes the conversion may fail, because we use an instance of IntEval^V on nonsensical arithmetic expressions.	²⁷ What should we do?

We agree to avoid such arithmetic expressions.¹

¹ In other words, we have *unsafe* evaluators for our expressions. One way to make them safe is to add a method that checks whether constants are instances of the proper class and that raises an exception [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.

²⁸ And their set expressions, too.

²⁹ 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?

³⁰ Yes. Let's implement an ExprVisitor^T for sets.

What do we need to implement one for sets? ³¹ We certainly need methods for *plusing*, *diffing*, and *prodig* sets.

That's correct, and here is everything.

³² 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.

³³ 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?

³⁴ Constructors always construct, and *add* does not always construct.

Do we need to understand that?

³⁵ Not now, but feel free to absorb it when you have the time.

Define the variants `Empty` and `Add` for `SetD`.³⁶ 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? ³⁷ Not now, but feel free to think about them when you have the time. We haven't even used visitors to define operations for union, set-difference, and intersection, but we trust you can.

What do we have to change in `IntEvalV` to obtain `SetEvalV`, an evaluator for set expressions?

How should we do that? ³⁹ Oh, that's a piece of pie. We just copy the definition of `IntEvalV` and replace its `plus`, `diff`, and `prod` methods.

That's the worst way of doing that. ⁴⁰ What?

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

⁴¹ That's true. If we copied the definition and changed it, we would have identical copies of `forPlus`, `forDiff`, `forProd`, and `forConst`. We should reuse this definition.¹

¹ Sometimes we do not have license to see the definitions, so copying might not even be an option.

Yes, and we are about to show you better ways. How do we have to change `plus`, `diff`, and `prod`?

⁴² 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); }  
}
```

Is it like `equals`?

⁴³ 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`?

⁴⁵ Three.

How many methods from `IntEvalV` are not overridden in `SetEvalV`?

⁴⁶ Four: `forPlus`, `forDiff`, `forProd`, and `forConst`.

Does `SetEvalV` implement `ExprVisitorI`?

⁴⁷ It doesn't say so.

Does `SetEvalV` extend `IntEvalV`?

⁴⁸ It says so.

Does `IntEvalV` implement `ExprVisitorI`?

⁴⁹ It says so.

Does `SetEvalV` implement `ExprVisitorI`?

⁵⁰ 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())?
```

⁵¹ 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))))?
```

⁵² It is a Prod and therefore an Expr^D.

And what does *accept* consume?

⁵³ An instance of SetEval^V, but its type is ExprVisitor^I.

What is

```
new SetEvalV().forProd(
  new Const(new Empty())
    .add(new Integer(7)),
  new Const(new Empty())
    .add(new Integer(3))))?
```

⁵⁴ 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^V's method
forProd? ⁵⁵ It is in IntEval^V.

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?

⁵⁶ If their values were A and B, we would have to determine the value of
prod(A,B).

Isn't that strange?

⁵⁷ Why?

So far, we have always used a method on a particular object.

⁵⁸ That's true. What is the object with which we use *prod(A,B)*?

It is **this** object.

⁵⁹ Oh, does that mean we should evaluate
new SetEval^V().*prod(A,B)*?

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

Good. And now what?

⁶¹ Now we still need to determine the values of
new Const(new Empty())
.add(**new Integer(7)**)
.accept(**this**)
and
new Const(new Empty())
.add(**new Integer(3)**)
.accept(**this**).

The values are obviously

new Empty()
.add(**new Integer(7)**)

and

new Empty()
.add(**new Integer(3)**).

Where is the definition of *forConst* that determines these values?

Here is the next expression in our sequence:

new SetEval^V()
.prod(**new Empty()**
.add(**new Integer(7)**),
new Empty()
.add(**new Integer(3)**)).

Where does *prod* come from?

⁶² It, too, is in **IntEval^V**.

⁶³ The object is an instance of **SetEval^V**, which overrides the *prod* method in **IntEval^V** with its own.

What next?

⁶⁴ Next we need to determine the value of
((Set^D)(new Empty()
.add(new Integer(7))))
.prod((Set^D)new Empty()
.add(new Integer(3))),

because it is

((Set^D)(l.accept(this)))
.prod((Set^D)r.accept(this))

with *l.accept(this)* and *r.accept(this)* replaced by their respective values.

Is

new Empty().add(new Integer(7))
an instance of Set^D?

⁶⁵ Of course it is, but the type of *l.accept(this)*, which is where it comes from, is Object.

And how about

new Empty().add(new Integer(3))?

⁶⁶ It's the same.

And that is why the method must contain a conversion from Object to Set^Ds.

⁶⁷ This example makes the need for conversions obvious again.

Time for the last question. Where does this *prod* come from now?

⁶⁸ This one belongs to Set^D or more precisely its Empty and Add variants.

And what does *prod* do?

⁶⁹ It determines the intersection of one Set^D with another Set^D, but didn't we agree that the previous question was the last question on that topic?

We overrode that, too.

⁷⁰ Thanks, guys.

Is it natural that SetEval^V extends IntEval^V? ⁷¹ No, not at all.

Why did we do that?

⁷² Because we defined `IntEvalV` first.¹

¹ 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.

⁷³ 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`?

⁷⁴ The methods `plus`, `diff`, and `prod`.

What are the pieces that they have in common?

⁷⁵ They share the methods `forPlus`, `forDiff`, `forProd`, and `forConst`.

Good. Here is how we express that.

⁷⁶ 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 $\text{Eval}^{\mathcal{D}}$ 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 $\text{IntEval}^{\mathcal{V}}$ extending $\text{Eval}^{\mathcal{D}}$.

```
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()); }  
}
```

Define $\text{SetEval}^{\mathcal{V}}$.

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

Time for supper.

⁷⁷ What do we do now?

⁷⁸ It is basically like the original but extends $\text{Eval}^{\mathcal{D}}$, not $\text{IntEval}^{\mathcal{V}}$.

```
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); }  
}
```

⁷⁹ Much more so than one extending the other.

⁸⁰ If you are neither hungry nor tired, you may continue.

Remember `SubstV` from chapter 6?

```
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)); }
```

⁸¹ Yes, and `LtdSubstV`, too.

```
class LdtSubstV implements PieVisitorT {
    int c;
    Object n;
    Object o;
    LdtSubstV(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 LdtSubstV(c - 1, n, o)));
            else
                return
                    new Top(t, r.accept(this)); }
```

What do the two visitors have in common?

⁸² Many things: `n`, `o`, and `forBot`.

Where do they differ?

⁸³ They differ in `forTop`, but `LtdSubstV` also has an extra field.

And where do we put the pieces that two classes have in common?

⁸⁴ We put them into an abstract class.

What else does the abstract class contain?

⁸⁵ It specifies the pieces that are different if they are needed for all extensions.

Define the **abstract class** Subst^D , which contains all the common pieces and specifies what a concrete pie substituter must contain in addition.

⁸⁶ It's not a big deal, except for the fields.

```
abstract class SubstD
    implements PieVisitorT {
    Object n;
    Object o;
    public PieD forBot() {
        return new Bot();
    }
    public
        abstract PieD forTop(Object t,PieD r);
}
```

We can define Subst^V by extending Subst^D .

```
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 LtdSubst^V .

⁸⁷ It also extends Subst^D .

```
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?

⁸⁸ No, but the constructors have some overlap. Shouldn't we lift the Subst^V constructor into Subst^D , because it holds the common elements?

That's a great idea. Here is the new version of `SubstD`.

```
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 `SubstV` and `LtdSubstV`.

⁸⁹ 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 LdtSubstV extends SubstD {
    int c;
    LdtSubstV(int _c, Object _n, Object _o) {
        super(_n, _o);
        c = _c; }

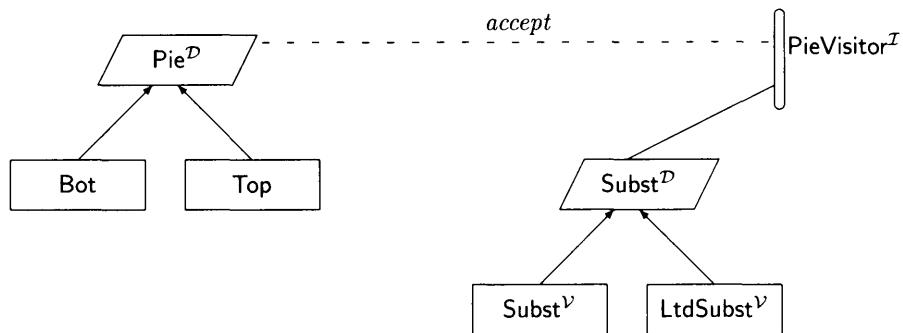
    public PieD forTop(Object t, PieD r) {
        if (c == 0)
            return new Top(t, r);
        else
            if (o.equals(t))
                return
                    new Top(n,
                        r.accept(
                            new LdtSubstV(c - 1, n, o)));
            else
  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

Was that first part easy?

⁹⁰ As pie.

That's neat. How about some art work?

⁹¹ Is this called a pie chart?



No, but the picture captures the important relationships.

⁹² Fine.

Is it also possible to define **LtdSubst^V** as an extension of **Subst^V**?

⁹³ It may even be better. In some sense, **LtdSubst^V** just adds a service to **Subst^V**: It counts as it substitutes.

If **LtdSubst^V** is defined as an extension of **Subst^V**, what has to be added and what has to be changed?

⁹⁴ As we just said, *c* is an addition and *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 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)); }  
}
```

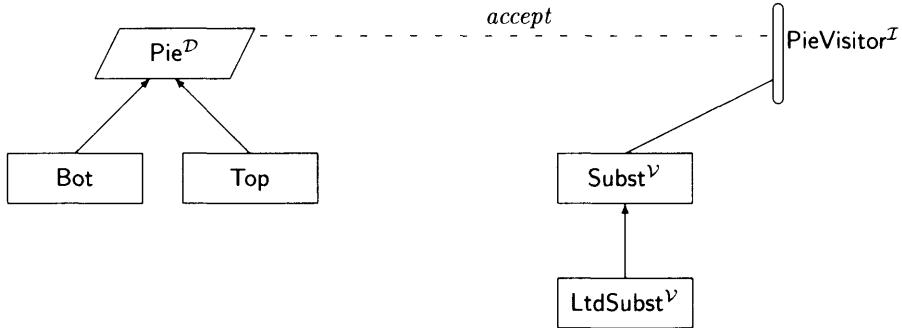
Define LtdSubst^V as an extension of Subst^V .

⁹⁵ 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.

⁹⁶ Fine, and don't forget to use lines, rather than arrows, for **implements**.



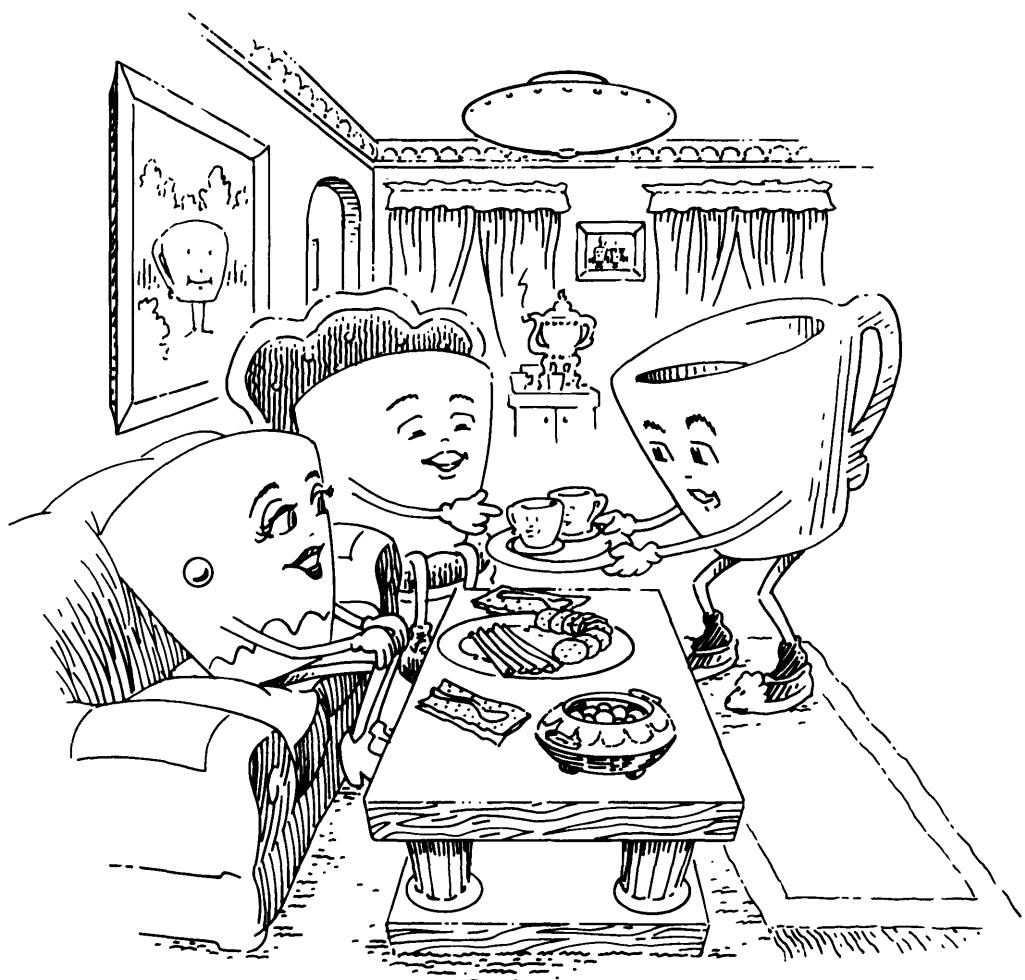
You deserve a super-deluxe pizza now.

⁹⁷ It's already on its way.

This page intentionally left blank

9.

Be a Good Visitor



Remember Point^D? If not, here is the datatype with one additional method, *minus*. We will talk about *minus* when we need it, but for now, just recall Point^D'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();  
}
```

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; }  
}
```

What is unusual about the constructor?

¹ It has been a long time since we discussed the datatype Point^D 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 ⌊√x2 + y2⌋; }  
}
```

```
class ManhattanPt extends PointD {  
    ManhattanPt(int _x,int _y) {  
        super(_x,_y); }  
  
    int distanceToO() {  
        return x + y; }  
}
```

² It uses
 $\Delta_x = _x$;
 $\Delta_y = _y$;
in addition to **super**($_x,_y$).

And what does that mean?

- ³ By using **super** on the first two values consumed, the constructor creates a **ShadowedManhattanPt** with proper *x* and *y* fields. The rest guarantees that this newly created point also contains values for the two additional fields.
-

Okay. So what is a **ShadowedManhattanPt**?

- ⁴ It is a **ManhattanPt** with two additional fields: Δ_x and Δ_y . These two represent the information that determines how far the shadow is from the point with the fields *x* and *y*.
-

Is this a **ShadowedManhattanPt**:

new ShadowedManhattanPt(2,3,1,0)?

- ⁵ Yes.

What is unusual about *distanceToO*?

- ⁶ Unlike any other method we have seen before, it contains the word **super**. So far, we have only seen it used in constructors. What does it mean?
-

Here, **super.distanceToO** refers to the method definition of *distanceToO* that is relevant in the class that **ShadowedManhattanPt** extends.

- ⁷ Okay. That means we just add *x* and *y* when we evaluate **super.distanceToO()**.
-

Correct. But what would we have done if **ManhattanPt** had not defined *distanceToO*?

- ⁸ Then we would refer to the definition in the class that **ManhattanPt** extends, right?
-

Yes, and so on. What is the value of
new ShadowedManhattanPt(2,3,1,0)
.distanceToO()?

- ⁹ It is 6, because $2 + 3$ is 5, and then we have to add 1 and 0.
-

Precisely. Now take a look at this extension of `CartesianPt`.

```
class ShadowedCartesianPt  
    extends CartesianPt {  
    int Δx;  
    int Δy;  
    ShadowedCartesianPt(int _x,  
                        int _y,  
                        int _Δx,  
                        int _Δy) {  
        super(_x,_y);  
        Δx = _Δx;  
        Δy = _Δy; }  
  
    int distanceToO() {  
        return  
            super.distanceToO()  
            +  
            ⌊√(Δx² + Δy²)⌋; }  
}
```

What is unusual about the constructor?

Is this a `ShadowedCartesianPt`:

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

And what is the value of
`new ShadowedCartesianPt(12,5,3,4)`.
`distanceToO()`?

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

¹¹ Yes.

¹² 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 Δ_x replaced by 3 and Δ_y replaced by 4.

What do we expect?

¹³ 17, obviously.

Why 17?

We need to add Δ_x to x and Δ_y to y when we think of a `ShadowedCartesianPt`.

Does this explain how `distanceToO` should measure the distance of a `ShadowedCartesianPt` to the origin?

Revise the definition of `ShadowedCartesianPt` accordingly.

- ¹⁴ Because we need to think of this point as if it were
`new CartesianPt(15,9).`
-

- ¹⁵ And indeed, the value of
`new CartesianPt(15,9)`
`.distanceToO()`

is 17.

- ¹⁶ 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.
-

- ¹⁷ 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?

- ¹⁸ No, once we have the distance, we have no need for this point.¹
-

¹ And neither does Java. Object-oriented languages manage memory so that programmers can focus on the difficult parts of design and implementation.

Correct. What is the value of

**new CartesianPt(3,4)
.closerToO(
new ShadowedCartesianPt(1,5,1,2))?**

¹⁹ true,

because the distance of the `CartesianPt` to
the origin is 5, while that of the
`ShadowedCartesianPt` is 7.

How did we determine that value?

²⁰ That's obvious.

Is the rest of this chapter obvious, too?

²¹ What?

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

²² Oh. Well, that makes the hint obvious.

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

²³ Fine.

Are sandwiches square meals for you?

²⁴ They can be well-rounded.

Here are circles and squares.

²⁵ Then this must be the datatype that goes
with it.

```
class Circle extends ShapeD {  
    int r;  
    Circle(int _r) {  
        r = _r; }  
  
    boolean accept(ShapeVisitorT ask) {  
        return ask.forCircle(r); }  
}
```

```
abstract class ShapeD {  
    abstract  
    boolean accept(ShapeVisitorT ask);  
}
```

```
class Square extends ShapeD {  
    int s;  
    Square(int _s) {  
        s = _s; }  
  
    boolean accept(ShapeVisitorT 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);  
}
```

²⁶ 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); }  
}
```

²⁷ Okay, now this looks pretty straightforward, but what's the point?

¹ A better name is **Translation**.

Let's create a circle.

²⁸ No problem:
new Circle(10).

How should we think about that circle?

²⁹ We should think about it as a circle with radius 10.

Good. So how should we think about
new Square(10)?

³⁰ Well, that's a square whose sides are 10 units long.

Where are our circle and square located?

³¹ What does that mean?

Suppose we wish to determine whether some `CartesianPt` is inside of the circle? ³² In that case, we must think of the circle as being drawn around the origin.

And how about the square?

³³ There are many ways to think about the location of the square.

Pick one.

³⁴ 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?

³⁵ Yes, it is, but barely.

And how about the circle?

³⁶ 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?

³⁷ 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))`?

³⁸ Aha. With `Trans` we can place a circle of radius 10 at a point like
 `new CartesianPt(5,6).`

How do we place a square's southwest corner at `new CartesianPt(5,6)`?

³⁹ Also with `Trans`:
 `new Trans(`
 `new CartesianPt(5,6),`
 `new Square(10)).`

Is `new CartesianPt(10,10)` inside either the circle or the square that we just referred to?

⁴⁰ It is inside both of them.

-
- How do we determine whether some point is inside a circle?
- 41 If the circle is located at the origin, it is simple. We determine the distance of the point to the origin and whether it is smaller than the radius.
-
- How do we determine whether some point is inside a square?
- 42 If the square is located at the origin, it is simple. We check whether the point's x coordinate is between 0 and s , the length of the side of the square.
-
- Is that all?
- 43 No, we also need to do that for the y coordinate.
-
- Aren't we on a roll?
- 44 We have only done the easy stuff so far. It is not clear how to check these things when the circle or the square are not located at the origin.
-
- Let's take a look at our circle around
new CartesianPt(5,6)
again. Can we think of this point as the origin?
- 45 We can if we translate all other points by an appropriate amount.
-
- By how much?
- 46 By 5 in the x direction and 6 in the y direction, respectively.
-
- How could we translate the points by an appropriate amount?
- 47 We could subtract the appropriate amount from each point.
-
- Is there a method in Point^D that accomplishes that?
- 48 Yes. Is that why we included *minus* in the new definition of Point^D?

Indeed. And now we can define the visitor **HasPt^V**, whose methods determine whether some **Shape^D** has a **Point^D** inside of it.

```
class HasPtV implements ShapeVisitorI {
    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)));
    }
}
```

What is the value of
new Circle(10)
.accept()
new HasPt^V(new CartesianPt(10,10)))?

Good. And what is the value of
new Square(10)
.accept()
new HasPt^V(new CartesianPt(10,10)))?

Let's consider something a bit more interesting. What is the value of
new Trans(
 new CartesianPt(5,6),
 new Circle(10))
.accept()
new HasPt^V(new CartesianPt(10,10)))?

⁴⁹ The three methods put into algebra what we just discussed.

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

⁵⁰ We said that this point wasn't inside of that circle, so the answer is **false**.

⁵¹ **true**.

⁵² 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)))?
```

But what is the value?

⁵³ Now that is tricky. We used **Trans** twice, which we should have expected given **Trans**'s definition.

And then?

⁵⁴ 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?

⁵⁶ Ten times, if we wish, because a **Trans** contains a **Shape**^D, and that allows us to nest things as often as needed.

Ready to begin?

⁵⁷ What? Wasn't that it?

No. The exciting part is about to start.

⁵⁸ We are all eyes.

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

⁵⁹ It becomes a square, obviously.

And the orange on top?

⁶⁰ A circle, **Transed** appropriately.

Can we think of the two objects as one?

⁶¹ 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(ShapeVisitorT ask) {  
        return _____; }  
}
```

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

⁶³ We know that a ShapeVisitor^T 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?

⁶⁴ We need to change ShapeVisitor^T 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 ourselves⁶⁵ Why can't we change it? to change ShapeVisitor^T.

Just to make the problem more interesting.

⁶⁶ 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.¹ This extension produces an interface that contains all the obligations (*i.e.*, names of methods and what they consume and produce) of **ShapeVisitor^I** and the additional one named *forUnion*.

¹ Unlike a class, an interface can actually extend several other interfaces. A class can implement several different interfaces.

Yes it should, but because **UnionVisitor^I** extends **ShapeVisitor^I**, it is also a **ShapeVisitor^I**.

⁶⁷ Which means that we extend **interfaces** the way we extend **classes**.

⁶⁸ Does that mean *accept* in **Union** should receive a **UnionVisitor^I**, so that it can use the *forUnion* method?

⁶⁹ We have been here before. Our *accept* method must consume a **ShapeVisitor^I** and fortunately every **UnionVisitor^I** implements a **ShapeVisitor^I**, too. But if we know that *accept* consumes a **UnionVisitor^I**, we can convert the **ShapeVisitor^I** to a **UnionVisitor^I** 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); }
}
```

⁷⁰ And it makes complete sense.

Let's create a Union shape.

⁷¹ 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 ⁷² We can't. HasPt^V is only a ShapeVisitor^T , it whether

`new CartesianPt(12,16)`
is inside?

Could it be a UnionVisitor^T ?

⁷³ No. It does not provide the method *forUnion*.

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

⁷⁴ Here it is. Its method checks whether the point is in one or the other part of a union. The other methods come from 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); }  
}
```

¹ We could have written the if... as
`return s.accept(this) || t.accept(this);`.

Does UnionHasPt^V contain *forUnion*?

⁷⁵ Of course, we just put it in.

Is `UnionHasPtV` a `UnionVisitorI`?

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); }
}
```

⁷⁶ It provides the required methods: `forCircle`, `forSquare`, `forTrans`, and `forUnion`.

⁷⁷ 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)))?
```

So?

⁷⁸ We know how `forTrans` works, so we're really asking whether

`new CartesianPt(10,10)`

is inside the `Union` shape.

⁷⁹ 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?

⁸⁰ It should be true.

Let's see whether the value of

```
new Trans(  
    new CartesianPt(3,7),  
    new Union(  
        new Square(10),  
        new Circle(10)))  
.accept(  
    new UnionHasPtV(  
        new CartesianPt(13,17)))
```

is true?

And?

⁸² It's a Shape^D.

How did we construct this shape?

⁸³ With Trans.

Which method should we use on it?

⁸⁴ forTrans, of course.

Where is forTrans defined?

⁸⁵ It is defined in HasPt^V.

So what should we do now?

⁸⁶ We should determine the value of
new Union(
 new Square(10),
 new Circle(10))
.accept(
 new HasPt^V(
 new CartesianPt(10,10))).

What type of object is

```
new Union(  
    new Square(10),  
    new Circle(10))?
```

⁸⁷ It's a Shape^D.

How did we construct this `ShapeD`?

⁸⁸ With Union.

So which method should we use on it?

⁸⁹ `forUnion`, of course.

How do we find the appropriate `forUnion` method?

⁹⁰ In `accept`, which is defined in `Union`, we confirm that

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

is a `UnionVisitorI` and then invoke its `forUnion`.

Is an instance of `HasPtV` a `UnionVisitorI`?

⁹¹ No!

Does it contain a method `forUnion`?

⁹² No!

Then what is the value of

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

⁹³ It doesn't have a value. We are stuck.¹

¹ A Java program raises a `RuntimeException`, indicating that the attempt to confirm the `UnionVisitorIness` of the object failed. More specifically, we would see the following when running the program:

```
java.lang.ClassCastException: UnionHasPtV  
    at Union.accept(...java:...)  
    at UnionHasPtV.forTrans(...java:...)  
    at Trans.accept(...java:...).
```

What do we do next?

⁹⁴ Relax. Read a novel. Take a nap.

Which of those is best?

⁹⁵ You guessed it: whatever you did is best.

We should have prepared this extension in a better way.

⁹⁶ 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 ShapeVisitorI {  
    PointD p;  
    HasPtV(PointD _p) {  
        p = _p; }  
    ShapeVisitorI newHasPt(PointD p) {  
        return new HasPtV(p); }  
  
    public boolean forCircle(int r) {  
        return p.distanceToO() ≤ r; }  
    public boolean forSquare(int s) {  
        ifI (p.x ≤ s)  
            return (p.y ≤ s);  
        else  
            return false; }  
    public  
    boolean forTrans(PointD q,ShapeD s) {  
        return  
            s.accept(newHasPt(p.minus(q))); }  
}
```

How does this definition differ from the previous one?

Good. What does `newHasPt` produce?

⁹⁸ A new `ShapeVisitorI`, as its interface implies.

And how does it produce that?

⁹⁹ By constructing a `new` instance of `HasPtV`.

Is `newHasPt` like a constructor?

¹⁰⁰ 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`¹⁰¹ and the previous one are really the same?¹

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.

¹ A functional programmer would say that `newHasPt` and `HasPtV` are η -equivalent.

Very well. But how does that help us with our problem?

¹⁰² That's not obvious.

Can we override `newHasPt` when we extend `HasPtV`?

¹⁰³ Yes, we can override any method that we wish to override.

Let's override `newHasPt` in `UnionHasPtV`.

¹⁰⁴ When we override it, we need to make sure it produces a `ShapeVisitorT`.

That's true. Should it produce a `HasPtV` or a `UnionHasPtV`?

¹⁰⁵ The latter. Then *forTrans* in `HasPtV` keeps producing a `UnionHasPtV`, if we start with a `UnionHasPtV`.

Good answer. Should we repeat it?

¹⁰⁶ 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 ¹⁰⁷ Here it is.
definition of UnionHasPt^V .¹

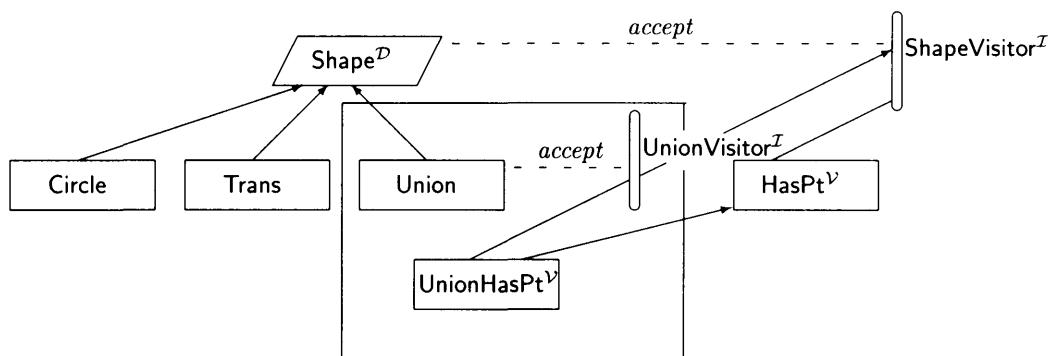
```
class UnionHasPtV
    extends HasPtV
    implements UnionVisitorI {
    UnionHasPtV(PointD -p) {
        super(-p); }
    ShapeVisitorI newHasPt(PointD p) {
        return new UnionHasPtV(p); }

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

¹ This is an instance of the *factory method* pattern [4].

If we assemble all this into one picture, what ¹⁰⁸ do we get?

A drawing that helps our understanding of the relationships among the classes and interfaces.



What does the box mean?

¹⁰⁹ 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? ¹¹⁰ No. It does not show the addition of a constructor-like method to `HasPtV` and how it is overridden in `UnionHasPtV`.

Is anything missing?

¹¹¹ `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)))?
```

¹¹² We remember that the shape was built with `Trans`.

Which method should we use on it?

¹¹³ *forTrans*, of course.

Where is *forTrans* defined?

¹¹⁴ It is defined in `HasPtV`.

So what should we do now?

¹¹⁵ We should determine the value of
`new Union(
 new Square(10),
 new Circle(10))
.accept(
 this.newHasPt(
 new CartesianPt(10,10))).`

What is `this`?

¹¹⁶ The current visitor, of course.

And how does that work?

¹¹⁷ We determine the value of
`this.newHasPt(
 new CartesianPt(10,10))`
and then use `accept` for the rest.

And what do we create?

¹¹⁸ The new UnionVisitor^T:
 new UnionHasPt^V(
 new CartesianPt(10,10)).

What is the value of
 new Union(
 new Square(10),
 new Circle(10))
 .accept(
 new UnionHasPt^V(
 new CartesianPt(10,10)))?

How do we do that?

¹¹⁹ UnionHasPt^V also satisfies the interface ShapeVisitor^T, so now we can invoke the *forUnion* method.

Is it true?

¹²⁰ We first determine the value of
 new Square(10)
 .accept(
 new UnionHasPt^V(
 new CartesianPt(10,10))).

If it is true, we're done.

Are we happy now?

¹²² Ecstatic.

Is it good to have extensible definitions?

¹²³ 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?

And why is that?

¹²⁴ Yes, we can and should always do so.

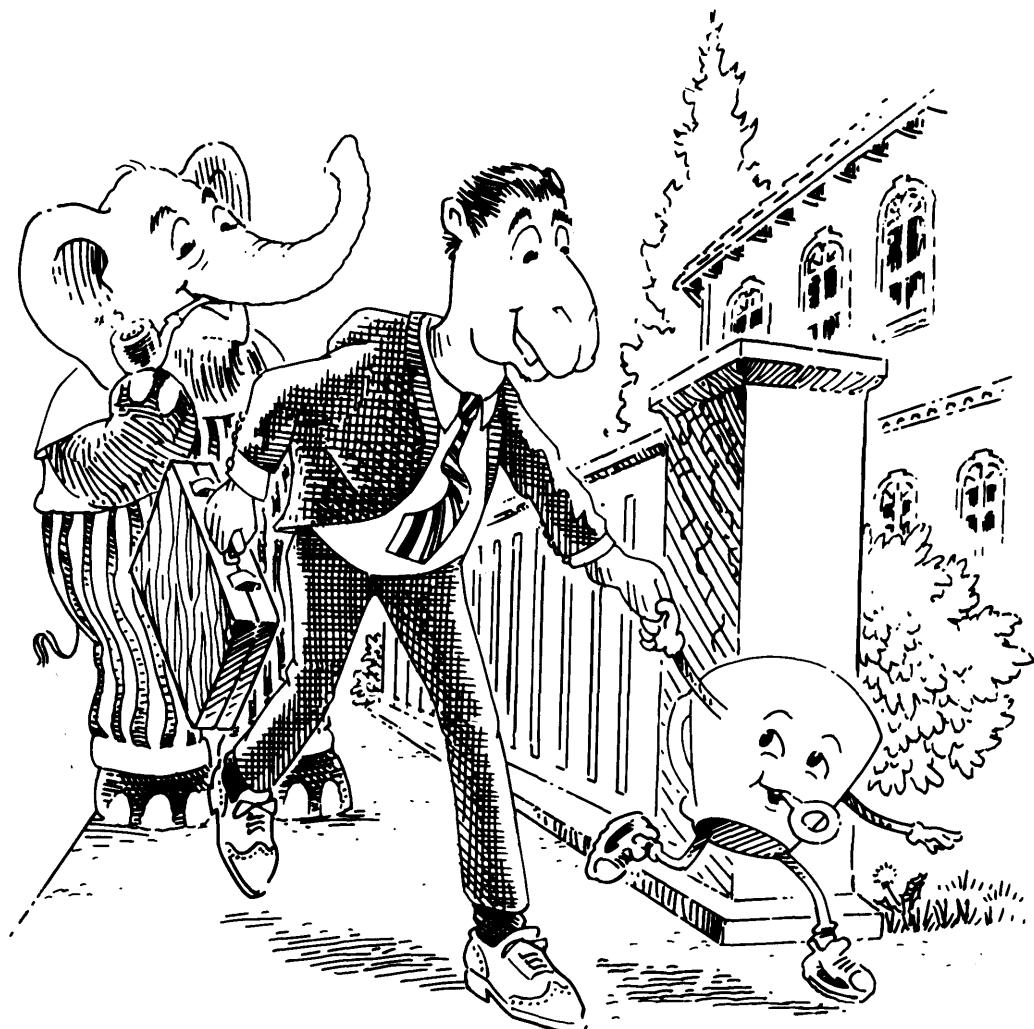
¹²⁵ Because no program is ever finished.

Are you hungry yet?

¹²⁶ Are our meals ever finished?

10.

The Shape of Things to Come



Have you ever wondered where the pizza pies come from? ¹ You should have, because someone needs to make the pie.

Here is our pizza pieman.

² This is beyond anything we have seen before.

```
class PiemanM implements PiemanI {
    PieD p = new Bot();
    public int addTop(Object t) {
        p = new Top(t,p)
        ;
        return occTop(t); }
    public int remTop(Object t) {
        p = (PieD)p.accept(new RemV(t))
        ;
        return occTop(t); }
    public int substTop(Object n,Object o) {
        p = (PieD)p.accept(new SubstV(n,o))
        ;
        return occTop(n); }
    public int occTop(Object o) {
        return
            ((Integer)p.accept(new OccursV(o)))
            .intValue(); }
}
```

^M This superscript is a reminder that the class manages a data structure. Lower superscripts when you enter this kind of definition in a file: `PiemanM`.

How so? Haven't we seen `PieD`, `Top`, and `Bot` ³ We have seen them. before?

And haven't we seen visitors like `RemV`, `SubstV`, and `OccursV` for various datatypes?

⁴ Yes, yes. But what are the stand-alone semicolons about?

Let's not worry about them for a while.

⁵ Fine, but they are weird.

Here is the **interface** for $\text{Pieman}^{\mathcal{M}}$.

```
interface PiemanI {
    int addTop(Object t);
    int remTop(Object t);
    int substTop(Object n, Object o);
    int occTop(Object o);
}
```

⁶ Isn't it missing p ?

We don't specify fields in interfaces. And in ⁷ Whatever.
any case, we don't want anybody else to
see p .

Here are $\text{PieVisitor}^{\mathcal{I}}$ and $\text{Pie}^{\mathcal{D}}$.

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

```
abstract class PieD {
    abstract
        Object accept(PieVisitorI ask);
}
```

Define **Bot** and **Top**.

⁸ They are very familiar.

```
class Bot extends PieD {
    Object accept(PieVisitorI ask) {
        return ask.forBot();
    }
}
```

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

```
Object accept(PieVisitorI ask) {
    return ask.forTop(t, r);
}
```

Here is `OccursV`. It counts how often some topping occurs on a pie.

```
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); }  
}
```

⁹ And this little visitor substitutes one good topping for another.

```
class SubstV implements PieVisitorT {  
    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 `RemV`, which removes a topping from a pie.

¹⁰ We remember that one, too.

```
class RemV implements PieVisitorT {  
    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

new Pieman^M().occTop(new Anchovy())?

¹¹ We first create a **Pieman^M** and then ask how many anchovies occur on the pie.

Which pie?

¹² The pie named *p* in the new **Pieman^M**.

And how many anchovies are on that pie?

¹³ None.

And what is the value of

new Pieman^M().addTop(new Anchovy())?

¹⁴ 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^M().addTop(new Anchovy()),

we must understand what

p = new Top(new Anchovy(),p)

;

return occTop(new Anchovy())

means?

¹⁵ 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 = new Top(new Anchovy(),p)?

That's right. But that's what happens when you have one too many double espressos.

¹⁶ So what does it mean?

Here it means that *p* changes and that future references to *p* reflect the change.

¹⁷ And the change is that *p* has a new topping, right?

When does the future begin?

¹⁸ Does it begin below the stand-alone semicolon?

That's precisely what a stand-alone semicolon means. Now do we know what

¹⁹ It produces the number of anchovies on *p*.

return occTop(new Anchovy())

produces?

And how many are there?

²⁰ We added one, so the value is 1.

And now what is the value of

new Pieman^M().addTop(new Anchovy())?

²¹ 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*.

²² Oh, isn't there a way to place several
requests with the same pieman?

Yes, there is. Take a look at this:

Pieman^I y = new Pieman^M().

²³ Okay, *y* stands for some pieman.

What is the value of

y.addTop(new Anchovy())?

²⁴ 1. We know that.

And now what is the value of

y.substTop(new Tuna(),new Anchovy())?

²⁵ 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())?

²⁶ 0, because *y*'s pie no longer contains any
anchovies.

Very good. And now take a look at this:

Pieman^I yy = new Pieman^M().

What is the value of

yy.addTop(new Anchovy())

;

yy.addTop(new Anchovy())

;

yy.addTop(new Salmon())

;

..

²⁷ 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())?
```

And what is the value of

```
yy.remTop(new Tuna())
```

after we are through with all that?

Does that mean *remTop* always produces 0? ³⁰ Yes, it always does.

Now what is the value of

```
yy.occTop(new Salmon())?
```

And how about

```
y.occTop(new Salmon())?
```

Is *yy* the same pieman as before?

So is it the same one?

When we asked *yy* to substitute all anchovies by tunas, did the pie change?

Does that mean that anybody can write

```
yy.p = new Bot()
```

and thus change a pieman like *yy*?

²⁸ 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.

²⁹ It's 0, because *remTop* first removes all tunas and then counts how many there are left.

³⁰ Yes, it always does.

³¹ 1.

³² 0, because *y* and *yy* are two different piemen.

³³ No, it changed.

³⁴ When we eat a pizza pie, we change, but we are still the same.

³⁵ The *p* in *yy* changed, nothing else.

³⁶ No, because *yy*'s type is *Pieman^T*, *p* isn't available. Only *addTop*, *remTop*, *substTop*, and *occTop* are visible.

Isn't it good that we didn't include p in Pieman^T ?

³⁷ Yes, with this trick we can prevent others from changing p (or parts of p) in strange ways. Everything is clear now.

Clear like soup?

³⁸ Just like chicken soup.

Can we define a different version of Subst^V so ³⁹ 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?

⁴⁰ No, a cup of coffee will do.

Compare this new PieVisitor^T with the first one in this chapter.

```
interface PieVisitorT {  
    Object forBot(Bot that);  
    Object forTop(Top that);  
}
```

⁴¹ It isn't all that different. A PieVisitor^T 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.

```
abstract class PieD {  
    abstract  
    Object accept(PieVisitorT ask);  
}
```

⁴² The definition is straightforward.

```
class Bot extends PieD {  
    Object accept(PieVisitorT ask) {  
        return ask.forBot(this); }  
}
```

Define the *Bot* variant.

Is it? Why does it use **this**?

⁴³ 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 ⁴⁴ Interesting.
Top, too.

```
class Top extends PieD {  
    Object t;  
    PieD r;  
    Top(Object _t,PieD _r) {  
        t = _t;  
        r = _r; }  
  
    Object accept(PieVisitorI ask) {  
        return ask.forTop(this); }  
}
```

Modify this version of Occurs^V so that it implements the new PieVisitor^I.

```
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); }  
}
```

⁴⁵ The *forBot* method basically stays the same, but *forTop* changes somewhat.

```
class OccursV implements PieVisitorI {  
    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?

⁴⁶ It now consumes a Bot, which is why we had to add (Bot *that*) behind its name.

How does *forTop* change?

⁴⁷ 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?

⁴⁸ With that, we can replace the fields *t* and *r* with *that.t* and *that.r*.

Isn't that easy?

⁴⁹ This modification of *Occurs^V* certainly is.

Then try *Rem^V*.

⁵⁰ It's easy; we use the same trick.

```
class RemV implements PieVisitorI {  
    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^V*?

⁵¹ Not really. It should be just like *Rem^V*.

And indeed, it is. Happy now?

⁵² So far, so good. But what's the point of this exercise?

Oh, *Point^Ds*? They will show up later.

⁵³ Seriously.

Here is the point. What is new about this version of 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; }  
    }  
}
```

⁵⁴ There are no **news**.

Don't they say "no **news** is good news?"

⁵⁵ Does this saying apply here, too?

Yes, because we want to define a version of Subst^V that modifies toppings without constructing a **new** pie.

⁵⁶ That's a way of putting it.

What do the methods of Subst^V always **return**?

⁵⁷ They always return *that*, which is the object that they consume.

So how do they substitute toppings?

⁵⁸ By changing the *that* before they return it. Specifically, they change the *t* field of *that* to *n* when it equals *o*.

What?

⁵⁹ The
that.t = n
does it.

Correct. And from here on, *that.t* holds the new topping. What is

that.r.accept(this)
about?

⁶⁰ In the previous *Subst^V*, *r.accept(this)* created a **new** pie from *r* with all toppings appropriately substituted. In our new version, *that.r.accept(this)* modifies the pie *r* so that below the following semicolon it contains the appropriate toppings.

Is there anything else to say about the new *Subst^V*?

⁶¹ Not really. It does what it does, which is what we wanted.¹

¹ This is a true instance of the *visitor* pattern [4]. What we previously called “visitor” pattern instances, were simple variations on the theme.

Do we have to change *Pieman^M*?

⁶² 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?

⁶³ Yes, because the *Pieman^M* manages the toppings of *p*, and nobody else sees *p*.

Can we do *LtdSubst^V* now without creating new instances of *LtdSubst^V* or *Top*?

⁶⁴ 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();  
}
```

⁶⁵ 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?

⁶⁶ Yes, we are missing ShadowedCartesianPt.

Good enough. We won't need it. Here is one⁶⁷ 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 **Point^D** the method *moveBy*, which consumes two **ints** and changes the fields of a point appropriately.

The method should return the new distance to the origin.

⁶⁸ First we must know what the method is supposed to produce.

⁶⁹ Now we know how to do this.

```
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

new ManhattanPt(1,4).

What is the value of

ptChild.distanceToO()?

⁷⁰ 5.

What is the value of
ptChild.moveBy(2,8)?

⁷¹ 15.

Good. Now let's watch a child with a
helium-filled balloon that casts a shadow.
Let *ptChildBalloon* be

⁷² 7.

new ShadowedManhattanPt(1,4,1,1).
What is the value of
ptChildBalloon.distanceToO()?

What is the value of
ptChildBalloon.moveBy(2,8)?

⁷³ 17, of course.

Did the balloon move, too?
⁷⁴ Yes, it just moved along as we moved the
point.

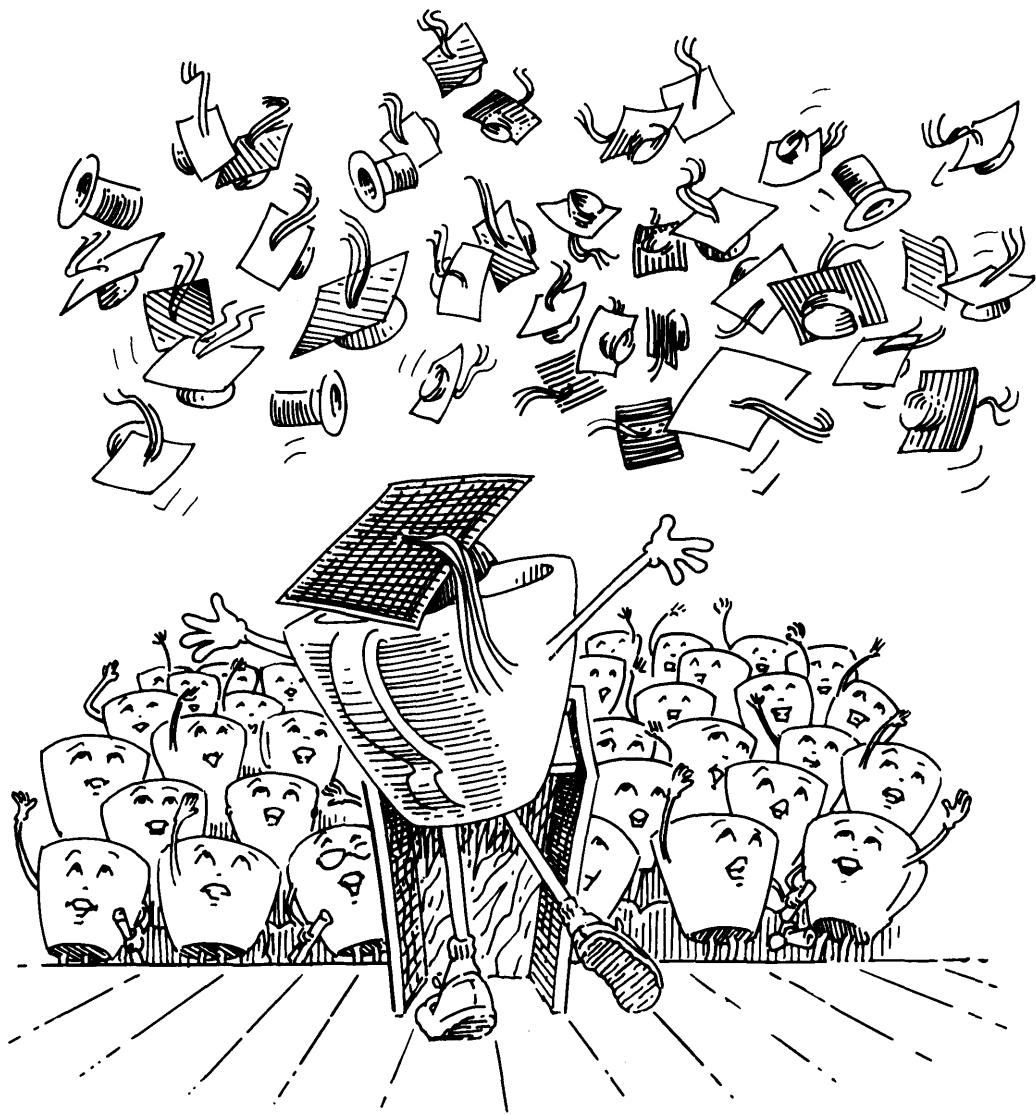
Isn't that powerful?
⁷⁵ It sure is. We added one method, used it,
and everything moved.

The more things change, the cheaper our
desserts get.
⁷⁶ 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.
⁷⁷ Don't forget to leave a tip.

This page intentionally left blank

Comienzo



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.

Index

- Add, 124
addTop, 161
Anchovy, 43, 53, 64, 65, 69, 72
Apple, 100
- Base, 10
 $bHasFruit^V$, 104
 $bIsFlat^V$, 102
 $bIsSplit^V$, 102, 103
Bot, 69, 78, 83, 85, 86, 91, 93, 162, 167
Brass, 29
Bud, 100, 101, 107, 110, 112
 $bTreeVisitor^T$, 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
 $Eval^D$, 130
 $Expr^D$, 119
 $ExprVisitor^T$, 118
- Fig, 100
 $Fish^D$, 69, 72
Flat, 100, 101, 108, 110, 112
 $Fruit^D$, 100
- Gold, 29
- $HasPt^V$, 147, 155
Holder, 28, 36
- $iHeight^V$, 108
 $IntEval^V$, 120, 122, 131
- $iOccurs^V$, 111
 $IsFlat^V$, 113
 $IsSplit^V$, 113
is Vegetarian, 25, 27
 $IsVegetarian^V$, 63
is Veggie, 30, 31, 36
 $iTreeVisitor^T$, 107
- Kebab D , 28, 36
- Lamb, 16, 27, 28, 36, 59, 62
 $Layer^D$, 10
Lemon, 100
 $LtdSubst^V$, 95, 97, 132–136
- ManhattanPt, 5, 13, 38, 39, 139, 172
mem, 124
minus, 139
moveBy, 173
- newHasPt*, 155, 157
 Num^D , 7, 79
- occTop*, 161
 $Occurs^V$, 114, 163, 168
Olive, 43, 53, 64, 65
 $OneMoreThan$, 7, 79
Onion, 16, 27, 28, 36, 59, 62
onlyOnions, 18, 27
 $OnlyOnions^V$, 57, 59
- Peach, 100
Pear, 100
Pepper, 4, 37
 Pie^D , 69, 78, 83, 85, 86, 91, 93, 162, 167
 $PieVisitor^T$, 92, 162, 167
 $Pieman^T$, 162
 $Pieman^M$, 161
 $Pizza^D$, 43, 53, 63–65
 $Plate^D$, 29
plus, 124
Plus, 119
 $Point^D$, 5, 13, 40, 139, 172, 173
prod, 124
Prod, 119

- Radish, 28, 36
 $remA$, 45, 49, 70
 $\text{RemA}^{\mathcal{V}}$, 66, 71, 73
 $\text{RemFish}^{\mathcal{V}}$, 74
 $\text{Rem}^{\mathcal{V}}$, 77, 87, 93, 163, 169
 $\text{RemInt}^{\mathcal{V}}$, 76
 $remTop$, 161
 $\text{Rod}^{\mathcal{D}}$, 29

 Sabre, 29
 Sage, 5
 Salmon, 69, 72
 Salt, 4
 Sausage, 43, 53, 63–65
 Seasoning $^{\mathcal{D}}$, 4
 Set $^{\mathcal{D}}$, 123
 SetEval $^{\mathcal{V}}$, 126, 131
 ShadowedCartesianPt, 141, 142
 ShadowedManhattanPt, 139, 172
 Shallot, 28, 36
 Shape $^{\mathcal{D}}$, 143
 ShapeVisitor $^{\mathcal{T}}$, 144
 Shish $^{\mathcal{D}}$, 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

 $subAbC$, 52, 53
 $\text{SubAbC}^{\mathcal{V}}$, 66
 $\text{Subst}^{\mathcal{D}}$, 133, 134
 $\text{SubstFish}^{\mathcal{V}}$, 82
 $\text{Subst}^{\mathcal{V}}$, 83, 87, 89, 94, 132–136, 163, 170
 $\text{SubstInt}^{\mathcal{V}}$, 82
 $substTop$, 161
 Sword, 29

 Thyme, 5
 Tomato, 16, 27, 28, 36, 59, 62
 Top, 69, 78, 83, 85, 86, 91, 93, 162 168
 $topAwC$, 50, 53
 $\text{TopAwC}^{\mathcal{V}}$, 66
 Trans, 144
 $\text{Tree}^{\mathcal{D}}$, 100, 101, 107, 109, 112
 $\text{TreeVisitor}^{\mathcal{T}}$, 112
 $t\text{Subst}^{\mathcal{V}}$, 111
 $t\text{TreeVisitor}^{\mathcal{T}}$, 109
 Tuna, 69, 72

 Union, 149, 150
 $\text{UnionHasPt}^{\mathcal{V}}$, 151, 152, 157
 $\text{UnionVisitor}^{\mathcal{T}}$, 150

 $whatHolder$, 33–36
 Wood, 29

 Zero, 7, 79
 Zucchini, 37,

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());  
    }  
}
```