A Little Java, A Few Patterns
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
To Helga, Christopher, and Sebastian.
To Mary, Rob, Rachel, Sara,
and to the memory of Brian.
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

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

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

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

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

Enjoy the pizzas!

Ralph E. Johnson
Champaign, Illinois
Preface

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

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

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

The Intended Audience

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

What this Book is Not About

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

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

Acknowledgments

We are indebted to many people for their contributions and assistance throughout the development of this book. Several extensive discussions with Shriram Krishnamurthi, Jon Rossie,
and Mitch Wand kept us on track; their detailed comments deeply influenced our thinking at critical junctures. Michael Ashley, Sundar Balasubramaniam, Cynthia Brown, Peter Drake, Bob Filman, Robby Findler, Steve Ganz, Paul Graunke, John Greiner, Erik Hilsdale, Matthew Kudzin, Julia Lawall, Shinn-Der Lee, Michael Levin, Gary McGraw, Benjamin Pierce, Amr Sabry, Jonathan Sobel, and George Springer read the book at various stages of development and their comments helped produce the final result. We also wish to thank Robert Prior at MIT Press who loyally supported us for many years and fostered the idea of a “Little Java.” The book greatly benefited from Dorai Sitaram’s incredibly clever Scheme typesetting program \SjTEX. 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. Create a file that contains a complete hierarchy of classes.
2. To each class whose name does not end with a superscript \( D, V, I, \) or \( M \), add a `toString` method according to these rules:
   a) if the class does not contain any fields, use
   ```java
   public String toString() {
       return "new " + getClass().getName() + "()";
   }
   ```
   b) if the class has one field, say \( x \), use
   ```java
   public String toString() {
       return "new " + getClass().getName() + "(" + x + ")";
   }
   ```
   c) if the class has two fields, say \( x \) and \( y \), use
   ```java
   public String toString() {
       return "new " + getClass().getName() + "(" + x + ", " + y + ")";
   }
   ```
3. Add the following class at the bottom of the file:
   ```java
   class Main {
       public static void main(String args[]) {
           DataType_or_Interface y = new ...;
           System.out.println( ... );
       }
   }
   ```

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

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

---

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.,

```java
y.­
y.­
y.-
```

replace ....... with

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

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

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

4. Finally, compile the file and interpret the class Main.
A Little Java, A Few Patterns
1. Modern Toys
Is 5 an integer?  
Yes, it is.

Is this a number: –23?  
Yes, but we don’t use negative integers.

Is this an integer: 5.32?  
No, and we don’t use this type of number.

What type of number is 5?  
int.¹  
¹ In Java, int stands for “integer.”

Quick, think of another integer!  
How about 19?

What type of value is true?  
boolean.

What type of value is false?  
boolean.

Can you think of another boolean?  
No, that’s all there is to boolean.

What is int?  
A type.

What is boolean?  
Another type.

What is a type?  
A type is a name for a collection of values.

What is a type?  
Sometimes we use it as if it were the collection.

Can we make new types?  
We don’t know how yet.
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\(^D\) {}

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

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

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

Yes, in a way. Now, is

- new Salt() a Seasoning\(^D\)?

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

And

- new Pepper()?

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?

Easy:

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

Is there any other Seasoning\(^D\)?

No, because only Salt and Pepper extend Seasoning\(^D\).

---

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 \textit{Seasoning}^D. Have we seen a datatype like \textit{Seasoning}^D before? No, but \texttt{boolean} is a type that also has just two values.

Let's define more \textit{Seasoning}^D's.

\begin{tabular}{|l|}
\hline
\texttt{class Thyme extends Seasoning^D} \\
\texttt{class Sage extends Seasoning^D} \\
\hline
\end{tabular}

And then there were four.

We can have lots of \textit{Seasoning}^D's.

\begin{tabular}{|l|}
\hline
\texttt{class Thyme extends Seasoning^D} \\
\texttt{class Sage extends Seasoning^D} \\
\hline
\end{tabular}

What is a Cartesian point?

Yes.

What is a point in Manhattan?

It is basically a pair of numbers.

How do \texttt{CartesianPt} and \texttt{ManhattanPt} differ from Salt and Pepper?

Each of them contains three things between \{ and \}. The \textit{x} and the \textit{y} are obviously the coordinates of the points. But what is the remaining thing above the bold bar?\footnote{This bar indicates the end of the constructor definition. It is used as an eye-catching separator. We recommend that you use \\
\texttt{// -------------------------------} when you enter it in a file.}

\begin{tabular}{|l|}
\hline
\texttt{abstract class Point^D} \\
\hline
\end{tabular}

\begin{tabular}{|l|}
\hline
\texttt{class CartesianPt extends Point^D} \\
\begin{tabular}{|l|}
\hline
\texttt{int x; int y;} \\
\texttt{CartesianPt(int \_x, int \_y) \{} \\
\texttt{\hspace{1em} x = \_x; \} \\
\texttt{\hspace{1em} y = -\_y; \}} \\
\end{tabular} \\
\hline
\end{tabular}

\begin{tabular}{|l|}
\hline
\texttt{class ManhattanPt extends Point^D} \\
\begin{tabular}{|l|}
\hline
\texttt{int x; int y;} \\
\texttt{ManhattanPt(int \_x, int \_y) \{} \\
\texttt{\hspace{1em} x = \_x; \} \\
\texttt{\hspace{1em} y = -\_y; \}} \\
\end{tabular} \\
\hline
\end{tabular}
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.
Do the following classes define another datatype with variants?

```java
abstract class NumD {}

class Zero extends NumD {}

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

Draw the picture, too.

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

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

How does OneMoreThan do that?  

And what does it mean to construct this new instance?

Does `predecessor` always stand for an instance of `Zero`?

---

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

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

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

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

No, its type says that it stands for a `Num\(^D\)`, which, at the moment, may be either a `Zero` or a `OneMoreThan`.
What is `new OneMoreThan(
    new OneMoreThan(
    new Zero()))`?

`new OneMoreThan( O)`?

Is `new ZeroO` the same as `0`?

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

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

Are there more `Num^D`s than `booleans`?

Are there more `Num^D`s than `ints`?

---

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

41 That is nonsense, because `0` is not a `Num^D`.

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

42 No, `0` is similar to, but not the same as, `new ZeroO`.

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

44 `4`.

45 Lots.

46 No. This answer is only conceptually correct. Java limits the number of `ints` to approximately $2^{32}$.
What is the difference between \texttt{new Zero()} and 0?

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

The primitive types (\texttt{int} and \texttt{boolean}) are distinct; others may overlap.

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

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

Almost. We will soon see what that means.

\textbf{The First Bit of Advice}

\textit{When specifying a collection of data, use abstract classes for datatypes and extended classes for variants.}
What do the following define?

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

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

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

What is

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

And what is

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

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

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

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

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

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

and

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

are instances of the same variant?

Hence, we can use both

```java
new Zero()
```

and

```java
new Salt()
```

for the construction of a `Base`, which requires an `Object`. 
Is anything else an Object?  

Correct. Is this a Layer$^D$:  

    new Base( 
      5)?  

Is this a Layer$^D$:  

    new Base( 
      false)?  

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

    new Base( 
      new Integer(5))?  

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

Easy now:  

    new Base( 
      new Boolean(false)).  

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

Too much coffee does that.  

Ready for more?  

Can't wait.
2.
Methods to Our Madness

Copyrighted Material
Remember points?

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

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

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

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

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;
    }
}
```

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

How far is

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

from the Empire State Building?

It must be for drawing the picture of the classes.

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

And how far is

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

from the origin?

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

**Methods to Our Madness**
Write the methods *distanceToO* using {, }, (, ), ;, return, int, +, \([\sqrt{\cdot}]\), and \(\cdot^2\), which determine how far a point is from the origin.

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

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

Here they are.

```java
abstract int distanceToO();

Point
```

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

CartesianPt
```

```java
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.

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

Three times, but the first one differs from the other two. It is labeled abstract, while the others are not preceded by a special word.
Do abstract methods belong to the abstract class? Yes, they always do.

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

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

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

7.

How do we arrive at that value?

We determine the value of 

\[ x + y, \]

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

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

5, because that is the value of

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

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

What does \( \lfloor \sqrt{x} \rfloor \) compute?

The largest \texttt{int} that does not exceed the square root of \( x \).

Time for a short break?

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

abstract class Shish$^D$ {
    ── Shish
}

class Skewer extends Shish$^D$ {
    ── Skewer
}

class Onion extends Shish$^D$ {
    Shish$^D$ s;
    Onion(Shish$^D$.s) {
        s = .s;
    }
    ── Onion
}

class Lamb extends Shish$^D$ {
    Shish$^D$ s;
    Lamb(Shish$^D$.s) {
        s = .s;
    }
    ── Lamb
}

class Tomato extends Shish$^D$ {
    Shish$^D$ s;
    Tomato(Shish$^D$.s) {
        s = .s;
    }
    ── Tomato
}

Did you notice the big space on the right? It is like Num$^D$ but has more variants.

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

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

And a third?

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

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

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

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

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

How about new Skewer()?

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

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

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

true.

false.

true.

false.
Write the methods *onlyOnions* \(^1\) 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.

```java
abstract boolean onlyOnions();

Shish
```

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

Skewer
```

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

Onion
```

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

Lamb
```

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

Tomato
```

Did you notice the labels in the boxes?

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

28 booleans.

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

29

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**?

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

Is this always the case?

What do these concrete methods consume?

What do these concrete methods produce?

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

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

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

---

31 Yes, we said so.

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

33 Always.

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

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

36 true.

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

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?

What is the field s of the object

\[
\text{new Onion(}
\text{new Onion(}
\text{new Skewer()))}
\]

Does s always stand for an Onion?

Then what is s.onlyOnions()?

Why do we need to know the meaning of

\[
\text{new Onion(}
\text{new Skewer()})
\]

\[
.s\text{.onlyOnions()}
\]

How do we determine the answer for

\[
\text{new Onion(}
\text{new Skewer()})
\]

\[
.s\text{.onlyOnions()}
\]
Which definition of *onlyOnions* must we use to determine the value of
\[
\text{new Onion(}
\text{new Skewer()})
\]
\[
\text{.onlyOnions()}
\]?

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

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

\[
s\text{.onlyOnions()}
\]

What is the field *s* of the object
\[
\text{new Onion(}
\text{new Skewer()})
\]

\[
\text{new Skewer()}
\]

Then what is *s*.\text{onlyOnions()}?

\[
\text{new Skewer()}
\]
\[
\text{.onlyOnions()}
\]
just as we would have expected.

Why do we need to know the meaning of
\[
\text{new Skewer()}
\]
\[
\text{.onlyOnions()}
\]?

Because the answer for
\[
\text{new Skewer()}
\]
\[
\text{.onlyOnions()}
\]
is also the answer for
\[
\text{new Onion(}
\text{new Skewer()})
\]
\[
\text{.onlyOnions()}
\]
which in turn is the answer for
\[
\text{new Onion(}
\text{new Onion(}
\text{new Skewer()))}
\]
\[
\text{.onlyOnions()}
\]

How do we determine the answer for
\[
\text{new Skewer()}
\]
\[
\text{.onlyOnions()}
\]?

We need to determine one more time which version of *onlyOnions* we must use.
Is  
\texttt{new Skewer()}

a 
\texttt{Skewer}?

Then what is the answer? 

Why? 

Are we done? 

What is the value of 
\texttt{new Onion(new Skewer()).onlyOnions()}?

Which definition of \texttt{onlyOnions} must we use to determine the value of 
\texttt{new Onion(new Lamb(new Skewer())).onlyOnions()}

51 Obviously.

52 \texttt{true}.

53 Because \texttt{true} is what the \texttt{onlyOnions} method in \texttt{Skewer} always returns.

54 Yes! The answer for 
\texttt{new Onion(new Onion(new Skewer()).onlyOnions())}

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

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

which is 
\texttt{true}.

55 false, isn't it?

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

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

Then what is `s.onlyOnions()`?

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

How do we determine the answer for `new Lamb(
  new Skewer())
  .onlyOnions()`, because the answer for `new Lamb(
  new Skewer())
  .onlyOnions()` is also the answer for `new Onion(
  new Lamb(
    new Skewer()))
  .onlyOnions()`.

And?

And now what is the answer?

false, because `false` follows the word `return` in the corresponding method definition in `Lamb`. 
Describe the methods (i.e., the function) onlyOnions in your own words.

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

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

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

And how about another Tomato?

Are we done?

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.

Here are our words: “The methods determine for a ShishD whether its contents are edible by an onion lover.”

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.”

Yes.

The object new Tomato(
    new Skewer())
is an instance of ShishD, so we can also wrap an Onion around it.

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

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

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

Of course, there is no Lamb on it.

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

That’s no big deal now.

abstract boolean isVegetarian();

Shish

boolean isVegetarian() {
    return true; }

Skewer

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

Onion

boolean isVegetarian() {
    return false; }

Lamb

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

Tomato
How many methods have we defined?  
\[ 73 \text{ Five: one } \textbf{abstract}, \text{ the others concrete.} \]

Do \textbf{abstract} methods belong to the \textbf{abstract} class?  
\[ 74 \text{ Yes, they always do.} \]

Does each variant of Shish\textsuperscript{D} contain a method called \textit{is Vegetarian}?  
\[ 75 \text{ Yes, because Shish}\textsuperscript{D} \text{ contains an } \textbf{abstract} \text{ method called } \textit{is Vegetarian}. \]

Is this always the case?  
\[ 76 \text{ Always.} \]

What do these concrete methods consume?  
\[ 77 \text{ Nothing, just as the } \textbf{abstract} \text{ method says.} \]

What do these concrete methods produce?  
\[ 78 \text{ booleans, just as the } \textbf{abstract} \text{ method says.} \]

\textbf{The Second Bit of Advice}

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

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

There are two methods per variant.

```java
class Skewer extends Shish$^D$ {
    boolean onlyOnions() {
        return true;
    }
    boolean isVegetarian() {
        return true;
    }
}
class Onion extends Shish$^D$ {
    Shish$^D$ s;
    Onion(Shish$^D$ _s) {
        s = _s;
    }
    boolean onlyOnions() {
        return s.onlyOnions();
    }
    boolean isVegetarian() {
        return s.isVegetarian();
    }
}
class Lamb extends Shish$^D$ {
    Shish$^D$ s;
    Lamb(Shish$^D$ _s) {
        s = _s;
    }
    boolean onlyOnions() {
        return false;
    }
    boolean isVegetarian() {
        return false;
    }
}
class Tomato extends Shish$^D$ {
    Shish$^D$ s;
    Tomato(Shish$^D$ _s) {
        s = _s;
    }
    boolean onlyOnions() {
        return false;
    }
    boolean isVegetarian() {
        return s.isVegetarian();
    }
}
```
They define a datatype and four variants that are similar in shape to Shish$^D$.
What is different about them?

Here are some holders.

Abstract class RodD {}

class Dagger extends RodD {}

class Sabre extends RodD {}

class Sword extends RodD {}

Are they good ones?

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

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?

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

Abstract class PlateD {}

class Gold extends PlateD {}

class Silver extends PlateD {}

class Brass extends PlateD {}

class Copper extends PlateD {}

class Wood extends PlateD {}

What is Shallot(
    new Radish(
        new Holder(
            new Dagger())))?

It’s a KebabD.
Is 
new Shallot(  
new Radish(  
new Holder(  
new Dagger()))))  
a vegetarian Kebab\textsuperscript{D}?  

Sure it is. It only contains radishes and shallots.

Is  
new Shallot(  
new Radish(  
new Holder(  
new Gold()))))  
a Kebab\textsuperscript{D}?  

Sure, because Gold is a \textsuperscript{D}Plate, \textsuperscript{D}Plate is used as a Holder, and radishes and shallots can be put on any Holder.

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

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 \textit{i.e.}, the function \textit{isVeggie},  
which check whether a kebab contains only  
vegetarian foods, regardless of what Holder it is on.

Write the abstract method \textit{isVeggie}.  

That’s possible now.

\begin{verbatim}
abstract boolean isVeggie()
\end{verbatim}  

Of course, \textit{isVeggie} belongs to Kebab\textsuperscript{D} and \textit{isVegetarian} to Shish\textsuperscript{D}
The concrete methods are similar to those called is Vegetarian. Here are two more; define the remaining two.

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

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

What is the value of

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

What is

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

What is the value of

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

And what is

```java
new Shallot(
    new Radish(
        new Holder(
            new Gold())));
```
What type of value is 
new Shallot( 
new Radish( 
new Holder( 
new Integer(52))))
.isVeggie()?

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

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

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

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

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

All the food is on a Dagger.

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

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

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

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

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

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

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

Here is the abstract method whatHolder.

```
abstract Object whatHolder()
```

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()?

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

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

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

With these kinds of hints, it's easy.

Object whatHolder() {
    return a;
}

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

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

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

Here is the method for Shallot.

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

They are all the same.

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

Write the methods of whatHolder for Shrimp and Radish.

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

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

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

Methods to Our Madness

Copyrighted Material
Here is the datatype and one of its variants.

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

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

Collect the remaining variants.

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

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

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

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

There are only three left.

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

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

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

Are there any other KebabD foods besides Shallot, Shrimp, and Radish?

No, these are the only kinds of foods on a KebabD.
Can we add more foods?

Let's define another Kebab\textsuperscript{D} class Pepper extends Kebab\textsuperscript{D} {
Kebab\textsuperscript{D} k;
Pepper(Kebab\textsuperscript{D} k) {
  k = k;
}
}

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

Is it obvious how the new methods work?

And then there were six.

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)?

Sure. We did something like that when we added Thyme and Sage to Seasoning\textsuperscript{D}.

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

class Zucchini extends Kebab\textsuperscript{D} {
Kebab\textsuperscript{D} k;
Zucchini(Kebab\textsuperscript{D} k) {
  k = k;
}
Object whatHolder() {
  return k.whatHolder();
}
}

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

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

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

The first one, clearly. Its distance to the origin is 5, but the second distance is 13.
We added the method closerToO to CartesianPt. It consumes another CartesianPt and determines whether the constructed or the consumed point is closer to the origin.

```java
class CartesianPt extends PointD {
    int x;
    int y;
    CartesianPt(int _x, int _y) {
        x = _x;
        y = _y;
    }
    
    int distanceToO() {
        return (int) Math.sqrt(x*x + y*y);    
    }
    boolean closerToO(CartesianPt p) {
        return distanceToO() <= p.distanceToO();
    }
}
```

Add the corresponding method to ManhattanPt.

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

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

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

false.

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

true, obviously.

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

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

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

That's nonsense.
Why?

How can we fix that?

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

Why does it help to replace \((\text{CartesianPt} \; p)\) by \((\text{PointD} \; p)\)?

Here is the improved CartesianPt.

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

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

    boolean closerToO(PointD p) {
        return distanceToO() \leq p.\text{distanceToO}();
    }
}
```

Improve the definition of ManhattanPt.

```java
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() \leq p.\text{distanceToO}();
    }
}
```

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

Yes, they are identical.

The method closerToO can only consume CartesianPts, not ManhattanPts.

We could replace \((\text{CartesianPt} \; p)\) by \((\text{PointD} \; p)\) in the definition of closerToO for CartesianPt.

Yes, because the definition of PointD obligates every variant to provide a method named distanceToO.

Every CartesianPt is a PointD, and every ManhattanPt is a PointD, too.

Easy.

Yes, they are identical.
Correct, and therefore we can add a copy to the abstract class \( \text{Point}^D \) and delete the definitions from the variants.

```java
abstract class Point\(^D\) {
    boolean closerTo0\(^D\)(Point\(^D\) p) {
        return
distanceToO() \leq p.distanceToO();
    }
    abstract int distanceToO();
}
```

1 The method closerTo0 is a template and the method distanceTo0 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.

```java
abstract class Point\(^D\) {
    int x;
    int y;
    Point\(^D\)(int \_x,int \_y) {
        x = \_x;
        y = \_y;
    }
    boolean closerTo0\(^D\)(Point\(^D\) p) {
        return
distanceToO() \leq p.distanceToO();
    }
    abstract int distanceToO();
}
```

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.

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

Mimicking this change is easy. But what does super(\_x,\_y) mean?

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

Do we now have everything that characterizes Point\(^D\)'s in the datatype?

Is this a long chapter?

That's simple.

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

Yes, let's have a short snack break.
3.

What's New?
Do you like to eat pizza?

abstract class Pizza\textsuperscript{D} {

}\begin{center}
\begin{tikzpicture}
\node (P) [shape=rectangle, draw] {Pizza};
\end{tikzpicture}
\end{center}

} 

\begin{center}
\begin{tikzpicture}
\node (Cr) [shape=rectangle, draw] {Crust};
\end{tikzpicture}
\end{center}

class Crust extends Pizza\textsuperscript{D} {

}\begin{center}
\begin{tikzpicture}
\node (C) [shape=rectangle, draw] {Cheese};
\end{tikzpicture}
\end{center}

class Cheese extends Pizza\textsuperscript{D} {

}\begin{center}
\begin{tikzpicture}
\node (Ol) [shape=rectangle, draw] {Olive};
\end{tikzpicture}
\end{center}

class Olive extends Pizza\textsuperscript{D} {

}\begin{center}
\begin{tikzpicture}
\node (An) [shape=rectangle, draw] {Anchovy};
\end{tikzpicture}
\end{center}

class Anchovy extends Pizza\textsuperscript{D} {

}\begin{center}
\begin{tikzpicture}
\node (Sa) [shape=rectangle, draw] {Sausage};
\end{tikzpicture}
\end{center}

class Sausage extends Pizza\textsuperscript{D} {

}\begin{center}
\begin{tikzpicture}
\node (P) [shape=rectangle, draw] {Pizza\textsuperscript{D} p;}
\node (S) [shape=rectangle, draw] {Sausage(Pizza\textsuperscript{D} \_p) { p = \_p; }}
\end{tikzpicture}
\end{center}

}\begin{center}
\begin{tikzpicture}
\node (S) [shape=rectangle, draw] {Sausage};
\end{tikzpicture}
\end{center}

Looks like good toppings. Let’s add Sausage.
Here is our favorite pizza:
```java
new Anchovy(
    new Olive(
        new Anchovy(
            new Anchovy(
                new Cheese(
                    new Crust())))))
```

How about removing the anchovies?

Let's remove them. What is the value of
```java
new Anchovy(
    new Olive(
        new Anchovy(
            new Anchovy(
                new Cheese(
                    new Crust())))))
```

...remA1()?

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

...remA()?

...remA?

...remA()?

Does remA belong to the datatype Pizza and its variants?

...remA belongs to the datatype Pizza and its variants.

...remA belongs to the datatype Pizza and its variants.

...remA belongs to the datatype Pizza and its variants.

This looks too salty.

That would make it less salty.

It should be a cheese and olive pizza, like this:
```java
new Olive(
    new Cheese(
        new Crust()))
```

It should be a cheese, sausage, and olive pizza, like this:
```java
new Sausage(
    new Olive(
        new Sausage(
            new Cheese(
                new Crust())))))
```

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

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

We didn't expect you to know this one.

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

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

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

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

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

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

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

```
What's New?
```

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

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

Yes, and now the methods of \( \text{remA} \) produce pizzas without any anchovies on them.
Let's try it out on a small pizza:
\[
\text{new Anchovy(}
\text{new Crust())}
\text{.remA().}
\]

Is
\[
\text{new Crust()}
\]
like
\[
\text{new Anchovy(}
\text{new Crust())}
\]
without anchovy?

No problem. Here is an example:
\[
\text{new Anchovy(}
\text{new Anchovy(}
\text{new Crust())})
\text{.remA().}
\]

Okay, so what if we had an olive and cheese on top:
\[
\text{new Olive(}
\text{new Cheese(}
\text{new Anchovy(}
\text{new Anchovy(}
\text{new Crust())})}
\text{.remA())}.
\]

Then, what is the value of
\[
\text{new Olive}(p.\text{remA()})
\]
where \(p\) stands for
\[
\text{new Cheese(}
\text{new Anchovy(}
\text{new Anchovy(}
\text{new Anchovy(}
\text{new Crust())})})
\text{.remA()}
\]

That's easy. The object is an Anchovy. So the answer is \text{new Crust()}.

Absolutely, but what if we had more anchovies?

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

Well, this object is an Olive and its \(p\) stands for
\[
\text{new Cheese(}
\text{new Anchovy(}
\text{new Anchovy(}
\text{new Crust())})})
\]

It is the pizza that
\[
\text{new Cheese(}
\text{new Anchovy(}
\text{new Anchovy(}
\text{new Crust())})}
\text{.remA()}
\]
produces, with an olive added on top.
What is the value of
\[ \text{new Cheese( new Anchovy( new Anchovy( new Crust()))).remA()}. \]

And what is the value of
\[ \text{new Cheese( new Anchovy( new Anchovy( new Crust()))) .remA()}. \]

Do we know the value of
\[ \text{new Anchovy( new Anchovy( new Crust())) .remA()}. \]

Does that mean that \text{new Crust()} is the answer?

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

So what is the final answer?

16. It is
\[ \text{new Cheese(p.remA())}, \]
where p stands for
\[ \text{new Anchovy( new Anchovy( new Crust()))}. \]

17. It is the pizza that
\[ \text{new Anchovy( new Anchovy( new Crust())) .remA()}
\]
produces, with cheese added on top.

18. Yes, we know that it produces \text{new Crust()}.

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

20. Yes, we must first add cheese, producing
\[ \text{new Cheese( new Crust())}
\]
and then we add the olive.

21. It is
\[ \text{new Olive( new Cheese( new Crust())))}. \]
Let’s try one more example:

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

What kind of pizza should this make?

Check it out!

Doesn’t that mean that the result is

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

The object is an instance of Cheese so the value is

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

where `p` stands for

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

What about

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

Now the object is an anchovy.

And the answer is

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

Yes, but we need to add cheese on top.

Does that mean the final answer is

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

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 29 The one in Anchovy.
    to get the cheese back?

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 32 Easy, it adds cheese on top of each anchovy:
new Olive(
    new Anchovy(
        new Cheese(
            new Anchovy(
                new Cheese(
                    new Crust()))))))
  .topAwC1()? 1

1 A better name for these methods would be topAnchovywithCheese.

Did you notice the underlines? 33 Yes, they show where we added cheese.

What's New?

Copyrighted Material
And what is
new Olive(
new Cheese(
new Sausage(
new Crust()))))
.topAwC()?

Define the remaining methods.

```java
abstract PizzaD topAwC();
```

Pizza

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

Crust

We expect you to know some of the answers.

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

Cheese

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

Olive

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

Sausage

Take a look at this method.

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

Anchovy

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

```java
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.

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

Three, because topAwC first adds cheese for each anchovy. Then remA removes all the anchovies:
new Olive(
    new Cheese(
        new Cheese(
            new Cheese(
                new Cheese(
                    new Crust()))))).

We just did something like that.

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

Aha!
Here is a different description:

“The methods look at each topping of a pizza and replace each anchovy with cheese.”

Define the methods that match this description. Call them \textit{subAbC}.\footnote{A better name for these methods would be \textit{substituteAnchovybyCheese}.} Here is the abstract method.

\begin{verbatim}
abstract Pizza\textsuperscript{D} subAbC();
\end{verbatim}

\begin{verbatim}
Pizza\textsuperscript{D} subAbC() {
\hspace{1em} return new Crust();
}
\end{verbatim}

\begin{verbatim}
Pizza\textsuperscript{D} subAbC() {
\hspace{1em} return new Cheese(p.subAbC());
}
\end{verbatim}

\begin{verbatim}
Pizza\textsuperscript{D} subAbC() {
\hspace{1em} return new Olive(p.subAbC());
}
\end{verbatim}

\begin{verbatim}
Pizza\textsuperscript{D} subAbC() {
\hspace{1em} return _______;
}
\end{verbatim}

\begin{verbatim}
Pizza\textsuperscript{D} subAbC() {
\hspace{1em} return new Sausage(p.subAbC());
}
\end{verbatim}

\begin{verbatim}
1 A better name for these methods would be \textit{substituteAnchovybyCheese}.
\end{verbatim}

Does this skeleton look familiar?\footnote{Yes, this skeleton looks just like those of \textit{topAwC} and \textit{remA}.}

Define the method that belongs in Anchovy.\footnote{Here it is.}

\begin{verbatim}
Pizza\textsuperscript{D} subAbC() {
\hspace{1em} return new Cheese(p.subAbC());
}
\end{verbatim}

\begin{verbatim}
Pizza\textsuperscript{D} subAbC() {
\hspace{1em} return new Cheese(p.subAbC());
}
\end{verbatim}
Collection time. The classes are getting larger.

abstract class PizzaD {
    abstract PizzaD remAO;
    abstract PizzaD topAwC();
    abstract PizzaD subAbC();
}

class Crust extends PizzaD {
    PizzaD remAO() {
        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 remAO() {
        return new Cheese(p.remA());
    }
    PizzaD topAwC() {
        return new Cheese(p.topAwC());
    }
    PizzaD subAbC() {
        return new Cheese(p.subAbC());
    }
}

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

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

Here is one addition: Spinach.

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

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

Isn't that neat?

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

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

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

Good idea.

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

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

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

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

Great.

We will stick with anchovies.
4.

Come to Our Carousel
Wasn’t this last collection overwhelming?  

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

Could it get worse?  

It sure could. For everything we want to do with \( \text{Pizza}^D \), we must add a method definition to each class.

Why does that become overwhelming?  

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

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

```
class OnlyOnions\(^\nu\) {
    boolean forSkewer() {
        return true;
    }
    boolean forOnion(Shish\(^D\) s) {
        return s.onlyOnions();
    }
    boolean forLamb(Shish\(^D\) s) {
        return false;
    }
    boolean forTomato(Shish\(^D\) s) {
        return false;
    }
}
```

\(^\nu\) 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 \( \text{onlyOnions} \) method in one of the variants of \( \text{Shish}^D \).

That’s right. The major difference is that they are all in one class, a visitor, whose name is OnlyOnions\(^\nu\).

Is \( \text{onlyOnions} \) different from OnlyOnions\(^\nu\)?

The former is used to name methods, the latter names a class.
And that’s the whole point.  

We want all the methods in one class.  

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

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

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

Right. What is the difference?  

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

It is time to discuss the boring part.  

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

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

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

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

Explain `s.onlyOnions()`.

Explain `ooFn.forOnion(s)`.

So what is the value of

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

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

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

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

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

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

Here are our words:

“`s` is a `Shish^D`, and therefore `s.onlyOnions()` determines whether what is below the onion is also edible by an onion lover.”

You knew we wouldn’t let you down:

“`ooFn.forOnion` says that we want to use the method we just described. It consumes a `Shish^D`, and `s` is the `Shish^D` that represents what is below the onion.”

It is still true.

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

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

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

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

And then the ride is over and we know that for this example the answer is true.

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

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

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

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

¹ *The American Heritage Dictionary* defines protocol as “[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.
We must add two lines to each variant, and they are almost the same as those for ooFn.

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

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

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

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

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

Yes, that one. Define the visitor. It is like OnlyOnions except for the method forTomato.

```java
class IsVegetarian {
    boolean forSkewer() {
        return true;
    }
    boolean forOnion(Shish s) {
        return s.isVegetarian();
    }
    boolean forLamb(Shish s) {
        return false;
    }
    boolean forTomato(Shish s) {
        return s.isVegetarian();
    }
}
```

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

How about a tea break? Instead of coffee?

The Fourth Bit of Advice

*When writing several functions for the same self-referential datatype, use visitor protocols so that all methods for a function can be found in a single class.*
Is
new Anchovy(
    new Olive(
        new Anchovy(
            new Cheese(
                new Crust()))))
a shish kebab?

No, it's a pizza.

abstract class Pizza^D {}

class Crust extends Pizza^D {}

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

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

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

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

So what do we do next?

We can define the protocol for the methods that belong to Pizza^D 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.

Okay, here is RemA\textsuperscript{v}.

\begin{verbatim}
class RemA\textsuperscript{v} {
    Pizza\textsuperscript{D} forCrust() {
        return new Crust();
    }
    Pizza\textsuperscript{D} forCheese(Pizza\textsuperscript{D} p) {
        return new Cheese(p.remA());
    }
    Pizza\textsuperscript{D} forOlive(Pizza\textsuperscript{D} p) {
        return new Olive(p.remA());
    }
    Pizza\textsuperscript{D} forAnchovy(Pizza\textsuperscript{D} p) {
        return p.remA();
    }
    Pizza\textsuperscript{D} forSausage(Pizza\textsuperscript{D} p) {
        return new Sausage(p.remA());
    }
}
\end{verbatim}

Define TopAwC\textsuperscript{v}.

The last one, SubAbC\textsuperscript{v}, is a piece of cake.

\begin{verbatim}
class TopAwC\textsuperscript{v} {
    Pizza\textsuperscript{D} forCrust() {
        return new Crust();
    }
    Pizza\textsuperscript{D} forCheese(Pizza\textsuperscript{D} p) {
        return new Cheese(p.topAwC());
    }
    Pizza\textsuperscript{D} forOlive(Pizza\textsuperscript{D} p) {
        return new Olive(p.topAwC());
    }
    Pizza\textsuperscript{D} forAnchovy(Pizza\textsuperscript{D} p) {
        return new Anchovy(p.topAwC());
    }
    Pizza\textsuperscript{D} forSausage(Pizza\textsuperscript{D} p) {
        return new Sausage(p.topAwC());
    }
}
\end{verbatim}

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

By now, even this is routine.

\begin{verbatim}
class SubAbC\textsuperscript{v} {
    Pizza\textsuperscript{D} forCrust() {
        return new Crust();
    }
    Pizza\textsuperscript{D} forCheese(Pizza\textsuperscript{D} p) {
        return new Cheese(p.subAbC());
    }
    Pizza\textsuperscript{D} forOlive(Pizza\textsuperscript{D} p) {
        return new Olive(p.subAbC());
    }
    Pizza\textsuperscript{D} forAnchovy(Pizza\textsuperscript{D} p) {
        return new Anchovy(p.subAbC());
    }
    Pizza\textsuperscript{D} forSausage(Pizza\textsuperscript{D} p) {
        return new Sausage(p.subAbC());
    }
}
\end{verbatim}

And we thought we were working with a pizza pie.
5.
Objects Are People, Too
Have we seen this kind of definition before? 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. Yes, it includes only one variant for adding toppings to a pizza, and toppings are Objects.

What kind of toppings can we put on these kinds of pizza? Any kind, because Object is the class of all objects. Here are some fish toppings.

abstract class FishD {}

class Anchovy extends FishD {}  
class Salmon extends FishD {}  
class Tuna extends FishD {}
Nice datatype. Is
\[
\text{new Top(new Anchovy(),}
\text{new Top(new Tuna(),}
\text{new Top(new Anchovy(),}
\text{new Bot()))})
\]
a pizza pie?

What is the value of
\[
\text{new Top(new Salmon(),}
\text{new Top(new Anchovy(),}
\text{new Top(new Tuna(),}
\text{new Top(new Anchovy(),}
\text{new Bot()))})
\]
.remA()?

Is it true that the value of
\[
\text{new Top(new Salmon(),}
\text{new Top(new Tuna(),}
\text{new Bot()))}
\]
.remA() is
\[
\text{new Top(new Salmon(),}
\text{new Top(new Tuna(),}
\text{new Bot()))}
\]

Does remA belong to Pie\textsuperscript{D}?

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

\[
\text{RemA}\textsuperscript{V} \text{ raFn = new RemA}\textsuperscript{V}();
\text{abstract Pie}\textsuperscript{D} \text{ remA();}
\]

It is a pizza pie, and so is
\[
\text{new Top(new Tuna(),}
\text{new Top(new Integer(42),}
\text{new Top(new Anchovy(),}
\text{new Top(new Integer(5),}
\text{new Bot()))})
\]

It is this fishy pizza pie:
\[
\text{new Top(new Salmon(),}
\text{new Top(new Tuna(),}
\text{new Bot())}
\]

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

Yes, and it produces pizza pies.

This is easy by now.

\[
\text{Pie}\textsuperscript{D} \text{ remA()} \\
\text{\hspace{1cm} return raFn.forBot(); }
\]

\[
\text{Pie}\textsuperscript{D} \text{ remA()} \\
\text{\hspace{1cm} return raFn.forTop(t,r); }
\]
Great. Isn't that easy?

What part of this exercise differs from datatype to datatype?

Anything else?

Why (t,r)?

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

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

Great guesses! What does

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

mean?

And what does

  new Anchovy().equals(t)

mean?

Not yet. It depends on what equals means.

Easy and boring.

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

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

Because these are the fields of Top.

Here are some guesses.

We guess:

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

We could guess:

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

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

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

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

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

The class Anchovy contains a method called equals. This method compares one Anchovy to another, and it always returns false, because no anchovy is a tuna.

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

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

Okay. How?

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

```
abstract class Fish\(^D\) {}

class Anchovy extends Fish\(^D\) {
    public\(^1\) boolean equals(Object o) {
        return (o instanceof Anchovy);
    }
}

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

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

Assuming that 
(o instanceof Tuna)
is true when o is an instance of Tuna, these method definitions are obvious.

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

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

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

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

Now, what is the value of `new Anchovy().equals(new Anchovy())`?  

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

Yet the value of `new Anchovy().equals(new Tuna())` is still false.

Of course, because an anchovy is never a tuna.

Could we have written `RemA^V` without using `equals`?  

Absolutely, `instanceof` is enough.

```java
class RemA^V {
  Pie^D forBot() {
    return new Bot();
  }
  Pie^D forTop(Object t,Piev^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.

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?  

How about `remFish` and `RemFish^V`?
How do we use remFish?

Add the protocol for RemFish\textsuperscript{\textregistered}. We designed the abstract portion.

\texttt{RemFish\textsuperscript{\textregistered} rfFn = new RemFish\textsuperscript{\textregistered}();
abstract Pie\textsuperscript{\textregistered} \textit{remFish}(Fish\textsuperscript{\textregistered} f);

Where do \textit{(f)} and \textit{(t,r,f)} come from?

Let's define RemFish\textsuperscript{\textregistered} and its two methods.

If we add another kind of fish to our datatype, what would happen to the definition of RemFish\textsuperscript{\textregistered}?
Let's try it out with a short example:

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

**Yes. What values does** `forTop` **consume?**

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

```java
new AnchovyO,
new BotO,
new AnchovyO;
```

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.

**And now?**

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

Now we need to determine the value of

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

**So?**

```java
new BotO;
```

It produces `new BotO`, no matter what `f` is.

**What is the value of**

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

It is the same as

```java
forBot(f),
```

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

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

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

**All clear?**

Ready to move on, after snack time.
Does \[
\text{new Top(new Integer(2),}
\text{new Top(new Integer(3),}
\text{new Top(new Integer(2),}
\text{new Bot())))}
\text{.remInt(new Integer(3))}
\]
look familiar?

What does \text{remInt} do? \[42\]

It removes Integers from pizza pies just as \text{remFish} removes fish from pizza pies.

Who defined \text{equals} for \text{Integer}? \[43\]
The Machine decided \[\text{new Integer(0).equals(new Integer(0))}\]
to be true, and the rest was obvious.

Define the visitor \text{RemInt}\text{\textsuperscript{V}} \[44\]
Wonderful! We do the interesting thing first. This visitor is almost identical to \text{RemFish}\text{\textsuperscript{V}}
We just need to change the type of what the two methods consume.

```java
class RemInt\text{\textsuperscript{V}} {
    Pie\textsuperscript{P} forBot(Integer i) {
        return new Bot();
    }
    Pie\textsuperscript{P} forTop(Object t,Pie\textsuperscript{P} 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 \text{i} and not \text{f}? \[45\]
No, \text{i} is just a better name than \text{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 PieDs?  

47 Yes.

Can we remove Fish$^D$ from PieDs?  

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 Rem$^V$ {
    Pie$^D$ forBot(Object o) {
        return new Bot();
    }
    Pie$^D$ forTop(Object t, Pie$^D$ r, Object o) {
        if (o.equals(t))
            return r.rem(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.

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.

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

And here are the pieces for Bot and Top.

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

class Top extends Pie^D {
    Object t;
    Pie^D r;
    Top(Object _t, Pie^D _r) {
        t = _t;
        r = _r;
    }

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

Let's remove some things from pizza pies:

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

Works like a charm with the same result as before.

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

Ditto.

Next:

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

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

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

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

What's wrong with that?
We expected it to remove new Zero() from the pizza.

And why didn't it?
Because equals for NumV's uses Object's equals, which always produces false—as we discussed above when we introduced equals.

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

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

```java
abstract class NumV {}

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

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

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

```java
class Zero extends NumV {
    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 NumV.
No. What is the type of o?  

So what is o.predecessor?  

Correct. What do we know after if has determined that  
(o instanceof OneMoreThan)  
is true?  

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

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

Are o and ((OneMoreThan)o) interchangeable?  

Is this complicated?  

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

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

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

Time for lunch?

Did you have a good lunch break?

Now what is the value of
\[
\text{new Top(new Anchovy(),}
\text{new Top(new Integer(3),}
\text{new Top(new Zero(),}
\text{new Bot())))
\]
\[.rem(new Zero())?
\]

And why?

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

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

What is the value of
\[
\text{new Top(new Anchovy(),}
\text{new Top(new Tuna(),}
\text{new Top(new Anchovy(),}
\text{new Bot())))
\]
\[.substFish(new Salmon(),
\text{new Anchovy())?}
\]

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

---

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

\(\text{1 Check chapter 1 for "similar."}

74 That’s just in time.

75 Yes, thank you.

76 Now we get
\[
\text{new Top(new Anchovy(),}
\text{new Top(new Integer(3),}
\text{new Bot()))},
\]
which is precisely what we want.

77 Because \textit{equals} now knows how to compare \texttt{NumDs}.

78 No, only if we need it.

79 Yes, we do.

80 It is the same pizza pie with all the anchovies replaced by salmon:
\[
\text{new Top(new Salmon(),}
\text{new Top(new Tuna(),}
\text{new Top(new Salmon(),}
\text{new Bot()))).}
\]

81 It consumes two \texttt{fish} and works on \texttt{PieDs}.
And what does it produce? 82

What is the value of
\[
\text{new Top(new Integer(3),}
\text{new Top(new Integer(2),}
\text{new Top(new Integer(3),}
\text{new Bot()))})
\text{.substInt(new Integer(5),}
\text{new Integer(3))}\? 83

It always produces a Pie$^D$.

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

What kind of values does 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$^V$.

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

Define SubstInt$^V$.

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

```java
class SubstInt$^V$ {
    Pie$^D$ forBot(Integer n,Integer o) {
        return new Bot();
    }
    Pie$^D$ forTop(Object t,
        Pie$^D$ r,
        Integer n,
        Integer o) {
        if (o.equals(t))
            return new Top(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$^V$.

Chapter 5

Copyrighted Material
Yes, we call it Subst\textsuperscript{V}. Define it.

Now it is time to add the protocol for Subst\textsuperscript{V} to Pie\textsuperscript{D}. Here are the variants.

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

class Top extends Pie\textsuperscript{D} {
    Object t;
    Pie\textsuperscript{D} r;
    Top(Object _t, Pie\textsuperscript{D} _r) {
        t = _t;
        r = _r; }
    Pie\textsuperscript{D} rem(Object o) {
        return remFn.forTop(t, r, o); }
    Pie\textsuperscript{D} subst(Object n, Object o) {
        return substFn.forTop(t, r, n, o); }
}
```

The abstract part is obvious.

```java
abstract class Pie\textsuperscript{D} {
    Rem\textsuperscript{V} remFn = new Rem\textsuperscript{V}();
    Subst\textsuperscript{V} substFn = new Subst\textsuperscript{V}();
    abstract Pie\textsuperscript{D} rem(Object o);
    abstract Pie\textsuperscript{D} subst(Object n, Object o);
}
```

So? That was some heavy lifting.
6.

Boring Protocols

Copyrighted Material
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 PieV {
RemV remFn = new RemV();
SubstV substFn = new SubstV();
abstract PieV rem(Object o);
abstract PieV subst(Object n, Object o);
}

What is the difference between rem and subst in PieV?

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

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

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;
Piev r;
Top(Object _t, Piev _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);
}

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

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.

Simpler: rem asks for the forTop service from remFn and hands over the field values and the Object it consumes; subst asks for the forTop service from substFn and hands over the field values and the two Objects it consumes.
And that is all there is to the methods in the variants of a protocol. But \texttt{remFn} and \texttt{substFn} defined in the datatype are still a bit mysterious.

Let's not create \texttt{remFn} and \texttt{substFn} in the datatype.

\begin{lstlisting}[language=Java]
abstract class \texttt{PieD} {
    abstract \texttt{PieD rem}(\texttt{Rem\textsuperscript{v} remFn},
        Object \texttt{o});
    abstract \texttt{PieD subst}(\texttt{Subst\textsuperscript{v} substFn},
        Object \texttt{n},
        Object \texttt{o});
}
\end{lstlisting}

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

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

The definition of the datatype says that they are a \texttt{Rem\textsuperscript{v}} and a \texttt{Subst\textsuperscript{v}}, respectively. And every \texttt{Rem\textsuperscript{v}} defines \texttt{forBot} and \texttt{forTop}, and so does every \texttt{Subst\textsuperscript{v}}.

Here is how it changes \texttt{Top}.

\begin{lstlisting}[language=Java]
class \texttt{Top} extends \texttt{PieD} {
    Object \texttt{t};
    \texttt{PieD r};
    \texttt{Top}(Object \_t,\texttt{PieD \_r}) {
        \texttt{t} = \_t;
        \texttt{r} = \_r;
    }
    \texttt{PieD rem}(\texttt{Rem\textsuperscript{v} remFn},
        Object \texttt{o}) {
        \texttt{return remFn.}\texttt{forTop}(\texttt{t},\texttt{r},\texttt{o});
    }
    \texttt{PieD subst}(\texttt{Subst\textsuperscript{v} substFn},
        Object \texttt{n},
        Object \texttt{o}) {
        \texttt{return substFn.}\texttt{forTop}(\texttt{t},\texttt{r},\texttt{n},\texttt{o});
    }
}
\end{lstlisting}

How does it affect \texttt{Bot}?

\begin{lstlisting}[language=Java]
class \texttt{Bot} extends \texttt{PieD} {
    \texttt{PieD rem}(\texttt{Rem\textsuperscript{v} remFn},
        Object \texttt{o}) {
        \texttt{return remFn.}\texttt{forBot}(\texttt{o});
    }
    \texttt{PieD subst}(\texttt{Subst\textsuperscript{v} substFn},
        Object \texttt{n},
        Object \texttt{o}) {
        \texttt{return substFn.}\texttt{forBot}(\texttt{n},\texttt{o});
    }
}
\end{lstlisting}
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\(^v\).

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

Modify Subst\(^v\) accordingly.

Consuming an extra value here also affects how the methods \textit{rem} and \textit{subst} are used.

In Rem\(^v\) and Subst\(^v\), the interesting parts, for example.

That takes all the fun out of it.

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

Yes, what about it. Copying is easy.

Which object?

The protocol is that \textit{rem} in \textit{Bot} and \textit{Top} asks for the \textit{forBot} and \textit{forTop} methods of \textit{remFn}, respectively.
How does that happen?  

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

That's it. Tricky? 

Why? 

What is the value of 

\[
\begin{align*}
&\text{new Top(new Anchovy(),} \\
&\quad \text{new Top(new Integer(3),} \\
&\quad \quad \text{new Top(new Zero(),} \\
&\quad \quad \quad \text{new Bot())}) \\
&\quad \quad \quad .rem(new Rem^V(),} \\
&\quad \quad \quad \quad \text{new Zero())?}
\end{align*}
\]

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

What is the value of 

\[
\begin{align*}
&\text{new Top(new Integer(3),} \\
&\quad \text{new Top(new Integer(2),} \\
&\quad \quad \text{new Top(new Integer(3),} \\
&\quad \quad \quad \text{new Bot())}) \\
&\quad \quad \quad .subst(new Subst^V(),} \\
&\quad \quad \quad \quad \text{new Integer(5),} \\
&\quad \quad \quad \quad \quad \text{new Integer(3))?}
\end{align*}
\]

17. It happens with 
\[
\begin{align*}
&\text{remFn.forBot(\ldots)} \\
&\quad \text{and} \\
&\text{remFn.forTop(\ldots),}
\end{align*}
\]
respectively.

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 \textit{r.rem(this,\ldots)} in forTop, it tells \textit{rem} to use the same instance over again.


20. Because this is a Rem^V, and it is exactly what we need to complete the job.

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

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

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

So what is the underlined part about? We changed the methods in Pie\(^D\), which means that we must also change how it is used.

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

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

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

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

Here is a way to define Subst\(^V\) that avoids the Wow. This visitor has two fields.\(^1\)

\begin{verbatim}
class Subst\(^V\) {
    Object \(n\);
    Object \(o\);
    Subst\(^V\)(Object \(n\),Object \(o\)) {
        \(n = n\);
        \(o = o\);
    }

    Pie\(^D\) forBot() {
        return new Bot();
    }
    Pie\(^D\) forTop(Object \(t\),Pie\(^D\) \(r\)) {
        if (\(o.equals(t)\))
            return new Top(n,r.subst(this));
        else
            return new Top(t,r.subst(this));
    }
}
\end{verbatim}

\(^1\) In functional programming, a visitor with fields is called a closure (or a higher-order function), which would be the result of applying a curried version of \textit{subst}.
How do we create a \texttt{SubstV}?

We use
\begin{verbatim}
new SubstV(new Integer(5),
           new Integer(3)).
\end{verbatim}

What does that do?

It creates a \texttt{SubstV} whose methods know how to substitute \texttt{new Integer(5)} for all occurrences of \texttt{new Integer(3)} in \texttt{Pie^D}.

How do the methods know that without consuming more values?

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

Okay, so how would we substitute all \texttt{new Integer(3)} with \texttt{new Integer(5)} in \begin{verbatim}
new Top(new Integer(3),
       new Top(new Integer(2),
              new Top(new Integer(3),
                     new Bot())))?
\end{verbatim}

We write
\begin{verbatim}
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))).
\end{verbatim}

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

We write
\begin{verbatim}
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))).
\end{verbatim}

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

Of course, because the methods \texttt{subst} in the \texttt{Bot} and \texttt{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?

```
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);
    }
}
```

37 In the Top variant, we still need to hand over both \( t \) and \( r \).

38 We haven't defined \( \text{Rem}^V \) 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 \( \text{rem} \) is defined as it is or as

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

42 Both \( \text{Rem}^V \) and \( \text{Subst}^V \) 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 \textbf{abstract class}.  

Boring Protocols

Copyrighted Material  

91
Yes! Do it! Great job, except that we will use interface for specifying visitors like these.

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

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

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

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

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

Fine.

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

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

What is a good name for this method?

The method accepts a visitor and asks for its services, so we call it accept.
And what is a better name for \textit{pvFn}? 

Now we can simplify the protocol. Here is the new \textit{RemV}.

\texttt{class RemV implements PieVisitor\textsuperscript{I} \{ 
\quad Object \_o; 
\quad RemV(\text{Object } \_o) \{ 
\quad o = \_o; \} 
\} 

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

Supply the protocol.

Did you notice the two underlined occurrences of \texttt{public}? 

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

It's a way to say that these are the methods that satisfy the obligations imposed by the \texttt{interface}.

Correct. They are just icing.

Easy: \textit{ask}, because we ask for services.

Here we go.

\texttt{abstract class PieD \{ 
\quad abstract PieD accept(PieVisitor\textsuperscript{I} ask); 
\} 

class Bot extends PieD \{ 
\quad PieD accept(PieVisitor\textsuperscript{I} ask) \{ 
\quad \quad return ask.forBot(); \} 
\} 

class Top extends PieD \{ 
\quad Object \_t; 
\quad PieD \_r; 
\quad Top(\text{Object } \_t,\text{PieD } \_r) \{ 
\quad \quad t = \_t; 
\quad \quad r = \_r; \} 
\quad PieD accept(PieVisitor\textsuperscript{I} ask) \{ 
\quad \quad return ask.forTop(t,r); \} 
\} 

Yes, what about them?

Why?

Looks weird, but let's move on.

Okay, we still won't forget them.
Now define the new \( \text{Subst}^Y \).

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

Why is there a line, not an arrow, from \( \text{Subst}^Y \) to \( \text{PieVisitor}^I \)?

And the dashed line?

Here it is.

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

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

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

Here is our picture.

The \( \text{Subst}^Y \) visitor **implements** \( \text{PieVisitor}^I \), it doesn’t **extend** it. Arrows mean “extends,” lines mean “implements.”

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

Explain what LtdSubstV produces.\textsuperscript{1}

\textsuperscript{1} A better name is LimitedSubstitutionV, and that is how we pronounce it.

Good. Define LtdSubstV

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

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

class LtdSubstV implements PieVisitorL {
    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()))?

How come?

Because c, the counting field, never changes.

Why doesn’t c ever change?

Because this, the LtdSubstV that performs the substitutions, never changes.

Can we fix this?

We can’t change this, but we can replace this with a new LtdSubstV that reflects the change.

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

Simple, we use
new LtdSubstV(c - 1, n, 0)
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 \( \mathrm{LtdSubst}^\nu \).

\[
\text{class } \text{LtdSubst}^\nu \text{ implements } \text{PieVisitor}^I \{ \\
\text{int } c; \\
\text{Object } n; \\
\text{Object } o; \\
\text{LtdSubst}^\nu(\text{int } \_c, \text{Object } \_n, \text{Object } \_o) \{ \\
c = \_c; \\
n = \_n; \\
o = \_o; \\
\} \\
\text{public } \text{Pie}^D \text{ forBot}() \{ \\
\text{return new } \text{Bot}(); \\
\} \\
\text{public } \text{Pie}^D \text{ forTop(Object } t, \text{Pie}^D r) \{ \\
\text{if } (c == 0) \\
\text{return new } \text{Top}(t, r); \\
\text{else} \\
\text{if } (o.equals(t)) \\
\text{return} \\
\text{new } \text{Top}(n, \\
r.\text{accept(} \\
\text{new } \text{LtdSubst}^\nu(c - 1, n, o))); \\
\text{else} \\
\text{return} \\
\text{new } \text{Top}(t, \\
r.\text{accept(} \\
\text{this)}); \\
\} \\
\}
\]

How does \textbf{this} differ from \textbf{new } \text{LtdSubst}^\nu(c - 1, n, o)\?  

They are two different \( \text{LtdSubst}^\nu \)s. One replaces \( c \) occurrences of \( o \) by \( n \) in a pizza pie, and the other one replaces only \( c - 1 \) of them.

How do you feel about protocols now?  

They are exciting. Let’s do more.
7.
Oh My!
Is  
new Flat(new Apple(),  
new Flat(new Peach(),  
new Bud())))  
a flat Tree^D?  

1. Yes.

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

2. Yes, it is also a flat Tree^D.

And how about  
new Split(  
new Bud(),  
new Flat(new Fig(),  
new Split(  
new Bud(),  
new Bud())))?  

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

Here is one more example:  
new Split(  
new Split(  
new Bud(),  
new Flat(new Lemon(),  
new Bud())),  
new Flat(new Fig(),  
new Split(  
new Bud(),  
new Bud()))).  

4. No, it isn't flat either.

Is it flat?  

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

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

6. Okay, let's.

Good. Then let's move on.

Oh My!

Copyrighted Material
Here are some fruits.

abstract class Fruit\textsuperscript{D} {}

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

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

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

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

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

Let's say all Tree\textsuperscript{D}s are either flat, split, or bud. Formulate a rigorous description for Tree\textsuperscript{D}s.

Did you notice that we have redefined the method \textit{equals} in the variants of Fruit\textsuperscript{D}? \textsuperscript{7} That probably means that we will need to compare fruits and other things.

Do Tree\textsuperscript{D}'s variants contain \textit{equals}? \textsuperscript{8} No, which means we won't compare them, but we could.
How does the datatype TreeD differ from all the other datatypes we have seen before?

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

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

Here is the new datatype definition.

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

Revise the variants.

```java
class Bud extends TreeD {
    boolean accept(bTreeVisitorT ask) {
        return ask.forBud();
    }
}
```

```java
class Flat extends TreeD {
    FruitD f;
    TreeD t;
    Flat(FruitD _f, TreeD _t) {
        f = _f;
        t = _t;
    }
    boolean accept(bTreeVisitorT ask) {
        return ask.forFlat(f, t);
    }
}
```

```java
class Split extends TreeD {
    TreeD l;
    TreeD r;
    Split(TreeD _l, TreeD _r) {
        l = _l;
        r = _r;
    }
    boolean accept(bTreeVisitorT ask) {
        return ask.forSplit(l, r);
    }
}
```

But isn’t bTreeVisitorT a pretty unusual name?

Yes, it is. Hang in there, we need unusual names for unusual interfaces. Here b reminds us that the visitor’s methods produce booleans.

Okay.
How many methods does the definition of `blsFlatV` contain, assuming it implements `bTreeVisitor^T`?

Three, because it works with `Tree^D`'s, and the datatype definition for `Tree^D`'s has three variants.

What type of values do the methods of `blsFlatV` produce?

`booleans`.

What visitor does `blsFlatV` remind us of?

`OnlyOnionsV`.

Here is a skeleton for `blsFlatV`.

```java
class blsFlatV implements bTreeVisitor^T {
    public
    boolean forBud() {
        return _________ ;
    }
    public
    boolean forFlat(Fruit^D f, Tree^D t) {
        return _________ ;
    }
    public
    boolean forSplit(Tree^D l, Tree^D r) {
        return _________ ;
    }
}
```

Fill in the blanks.

Define the `blsSplitV` visitor, whose methods check whether a `Tree^D` is constructed with `Split` and `Bud` only.

Here is the easy part.

```java
class blsSplitV implements bTreeVisitor^T {
    public
    boolean forBud() {
        return true; }
    public
    boolean forFlat(Fruit^D f, Tree^D t) {
        return t.accept(this); }
    public
    boolean forSplit(Tree^D l, Tree^D r) {
        return false; }
}
```
What is difficult about the last line?

Isn't that easy?

And then?

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

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

Finish the definition of \texttt{blsSplitV} using
\begin{verbatim}
   if (...) 
      return ...
   else 
      return ...
\end{verbatim}

We need to check whether both \(l\) and \(r\) are split trees.

Yes, we just use the methods of \texttt{blsSplitV} on \(l\) and \(r\).

Then we need to know that both are true.

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

No, then the answer is false.

Now we can do it.

\begin{verbatim}
class blsSplitV implements bTreeVisitorI {
   public 
      boolean forBud() { 
         return true; }
   public 
      boolean forFlat(FruitD f, TreeD t) { 
         return false; }
   public 
      boolean forSplit(TreeD l, TreeD r) {
         if (l.accept(this)) 
            return r.accept(this);
         else 
            return false; }
}
\end{verbatim}

\begin{footnote}
  \textsuperscript{1} We could have written the if... as 
  \texttt{return l.accept(this) && r.accept(this).}
\end{footnote}
Give an example of a $\text{Tree}^D$ for which the methods of $\text{blsSplit}^V$ respond with true.

There is a trivial one:
new Bud().

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

Here is one:
new Split(
new Split(
new Bud(),
new Split(
new Bud(),
new Bud())),
new Split(
new Bud(),
new Split(
new Bud(),
new Bud()))).

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

No.

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

Here it is.

```java
class bHasFruit$^V$
  implements bTreeVisitor$^T$ {
public
  boolean forBud() {
    return false; }
public
  boolean forFlat(Fruit$^D$ f, Tree$^D$ t) {
    return true; }
public
  boolean forSplit(Tree$^D$ l, Tree$^D$ r) {
    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
new Split(
  new Split(
    new Bud(),
    new Flat(new Lemon(),
      new Bud())),
  new Flat(new Fig(),
    new Split(
      new Bud(),
      new Bud())))?

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

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

What is the height of
new Bud()?

So what is the height of a Tree\(^D\)?

Do the methods of iHeight\(^V\) work on a Tree\(^D\)?

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

Why is the height 4?
Because the value of
new Split(
    new Bud(),
    new Bud())
    .accept(new iHeightV())
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?
We need to add one to the larger of the
two numbers so that we don't forget that the
original TreeD was constructed with Split and
those two TreeD's.

□ picks the larger of two numbers, x and y. 1

1 When you enter this in a file, use
Math.max(x,y).
Math is a class that contains max as a (static) method.

Oh, that’s nice. What kind of methods does
iHeightV define?

iHeightV's methods measure the heights of
the TreeD's to which they correspond.

Now that's a problem.
Why?

So what?

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

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

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 TreeD’s variants. Start with the easy one.

---

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

41 The methods of iHeight\(^V\) produce ints, which are not booleans.

42 It’s almost the same as bTreeVisitor\(^I\).

```java
interface iTreeVisitor\(^I\) {
    int forBud();
    int forFlat(Fruit\(^D\) f, Tree\(^D\) t);
    int forSplit(Tree\(^D\) l, Tree\(^D\) r);
}
```

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 bTreeVisitor\(^I\) is indeed different from iTreeVisitor\(^I\), so we can have two versions of accept in Tree\(^D\).

45 It is easy.

```java
class Bud extends Tree\(^D\) {
    boolean accept(bTreeVisitor\(^I\) ask) {
        return ask.forBud();
    }
    int accept(iTreeVisitor\(^I\) ask) {
        return ask.forBud();
    }
}
```
The others are easy, too. We duplicate accept.

```java
class Flat extends TreeD {
    FruitD f;
    TreeD t;
    Flat(FruitD _f, TreeD _t) {
        f = _f;
        t = _t;
    }
    boolean accept(bTreeVisitorI ask) {
        return ask.forFlat(f, t);
    }
}
```

Here is iHeightV

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

Complete these methods.

What is the value of
```
new Split(
    new Bud(),
    new Bud())
.accept(new iHeightV())?
```

And why is it 1?
```
Because
    new Bud().accept(new iHeightV())
is 0, the larger of 0 and 0 is 0, and one more is 1.
```

We must also change the type of what the new accept method consumes and produces.

```java
class Split extends TreeD {
    TreeD l;
    TreeD r;
    Split(TreeD _l, TreeD _r) {
        l = _l;
        r = _r;
    }
    boolean accept(bTreeVisitorI ask) {
        return ask.forSplit(l, r);
    }
}
```

That's easy now.

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

1, of course.
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 SubstY(
            new Apple(),
            new Fig()))?

Correct. Define the SubstY visitor.

If the visitor SubstY 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())))).

What's the problem?

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

Its methods produce TreeDs, 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);
}

Good job. How about the datatype TreeD

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 TreeD.

No problem.

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

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

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

Here is a Tree$^D$ that has three Figs:

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

Now define $i_{\text{Occurs}}$, whose methods count how often some Fruit$^D$ occurs in a tree.

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

Even the visitors are no longer interesting.

```
class $i_{\text{Occurs}}$ implements $i_{\text{TreeVisitor}}$
{
    Fruit$^D$ a;
    $i_{\text{Occurs}}$(Fruit$^D$ a) {
        a = _a;
    }
    
    public int forBud() {
        return 0;
    }
    public int forFlat(Fruit$^D$ f, Tree$^D$ t) {
        if (f.equals(a))
            return t.accept(this) + 1;
        else
            return t.accept(this);
    }
    public int forSplit(Tree$^D$ l, Tree$^D$ r) {
        return l.accept(this) + r.accept(this);
    }
}
```
Do you like your fruit with yogurt?  

Is it disturbing that we have three nearly identical versions of accept in TreeDs and its variants?  

Can we avoid it?  

Remember Integer and Boolean? They make it possible.

Here is the interface for a protocol that produces Object in place of boolean, int, and TreeD.

```java
interface TreeVisitor<T> {
    Object forBud();
    Object forFlat(FruitD f, TreeD t);
    Object forSplit(TreeD l, TreeD r);
}
```

Here is the datatype and the Bud variant.

```java
abstract class TreeD {
    abstract
    Object accept(TreeVisitor<T> ask);
}
```

class Bud extends TreeD {
    Object accept(TreeVisitor<T> ask) {
        return ask.forBud();
    }
}

Define the remaining variants of TreeD.

class Flat extends TreeD {
    FruitD f;
    TreeD t;
    Flat(FruitD _f, TreeD _t) {
        f = _f;
        t = _t;
    }
    Object accept(TreeVisitor<T> 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(TreeVisitor<T> ask) {
        return ask.forSplit(l, r);
    }
}

We prefer coconut sorbet.

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.

If boolean and int were classes, we could use Object for boolean, int, and TreeD. Unfortunately, they are not.

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

Here they are.

```java
class Flat extends TreeD {
    FruitD f;
    TreeD t;
    Flat(FruitD _f, TreeD _t) {
        f = _f;
        t = _t;
    }
    Object accept(TreeVisitor<T> 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(TreeVisitor<T> ask) {
        return ask.forSplit(l, r);
    }
}
```
Good. Now define IsFlat\textsuperscript{V}, an Object producing version of blsFlat\textsuperscript{V}.

```java
class IsFlat\textsuperscript{V} implements TreeVisitor\textsuperscript{D} {
    public Object forBud() {
        return new Boolean(true);
    }
    public Object forFlat(Fruit\textsuperscript{D} f, Tree\textsuperscript{D} t) {
        return t.accept(this);
    }
    public Object forSplit(Tree\textsuperscript{D} l, Tree\textsuperscript{D} r) {
        return new Boolean(false);
    }
}
```

That’s no big deal.

And how about IsSplit\textsuperscript{V}?

```java
class IsSplit\textsuperscript{V} implements TreeVisitor\textsuperscript{D} {
    public Object forBud() {
        return new Boolean(true);
    }
    public Object forFlat(Fruit\textsuperscript{D} f, Tree\textsuperscript{D} t) {
        return t.accept(this);
    }
    public Object forSplit(Tree\textsuperscript{D} l, Tree\textsuperscript{D} r) {
        return new Boolean(false);
    }
}
```

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

```java
class IsSplit\textsuperscript{V} implements TreeVisitor\textsuperscript{D} {
    public Object forBud() {
        return new Boolean(true);
    }
    public Object forFlat(Fruit\textsuperscript{D} f, Tree\textsuperscript{D} t) {
        return t.accept(this);
    }
    public Object forSplit(Tree\textsuperscript{D} l, Tree\textsuperscript{D} r) {
        if (((Boolean) (l.accept(this)))) .booleanValue())
            return r.accept(this);
        else
            return new Boolean(false);
    }
}
```

That’s no big deal.

Okay, here it is.

```java
class IsSplit\textsuperscript{V} implements TreeVisitor\textsuperscript{D} {
    public Object forBud() {
        return new Boolean(true);
    }
    public Object forFlat(Fruit\textsuperscript{D} f, Tree\textsuperscript{D} t) {
        return new Boolean(false);
    }
    public Object forSplit(Tree\textsuperscript{D} l, Tree\textsuperscript{D} r) {
        if (((Boolean) (l.accept(this)))) .booleanValue())
            return r.accept(this);
        else
            return new Boolean(false);
    }
}
```

That’s no big deal.

Oh, because \texttt{l.accept(this)} produces an Object, we must first convert\textsuperscript{1} it to a \texttt{Boolean}. Then we can determine the underlying \texttt{boolean} with the \texttt{boolean Value} method. We have seen this in chapter 5 when we converted an Object to a OneMoreThan.

\textsuperscript{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, \textit{Conference Record on Principles of Programming Languages}, 146–159. Paris, 1997).

Will the conversion always work?

```java
class IsSplit\textsuperscript{V} implements TreeVisitor\textsuperscript{D} {
    public Object forBud() {
        return new Boolean(true);
    }
    public Object forFlat(Fruit\textsuperscript{D} f, Tree\textsuperscript{D} t) {
        return new Boolean(false);
    }
    public Object forSplit(Tree\textsuperscript{D} l, Tree\textsuperscript{D} r) {
        if (((Boolean) (l.accept(this)))) .booleanValue())
            return r.accept(this);
        else
            return new Boolean(false);
    }
}
```

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

The Seventh Bit of Advice

\textit{When designing visitor protocols for many different types, create a unifying protocol using Object.}

\textit{Copyrighted Material}
Did you think that was bad? Then study this definition during your next break.

```java

class OccursV implements TreeVisitorI {
    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()));
    }

}  
```

Oh my!
Like Father, Like Son
What is the value of  
\((7 + ((4 - 3) \times 5))\)\?  
\(^1\) 12.

What is the value of  
\((+ 7 (\times (- 4 3) 5))\)?  
\(^2\) 12,  
because we have just rewritten the  
previous expression with prefix operators.

What is the value of  
\(\text{new Plus(}\)  
\(\text{new Const(new Integer(7)),}\)  
\(\text{new Prod(}\)  
\(\text{new Diff(}\)  
\(\text{new Const(new Integer(4)),}\)  
\(\text{new Const(new Integer(3)),}\)  
\(\text{new Const(new Integer(5)))))?\)  
\(^3\) \text{new Integer(12),}  
because we have just rewritten the  
previous expression using \text{Integer} and  
constructors.

Where do the constructors come from?  
\(^4\) A datatype and its variants that represent  
arithmetic expressions.

Did you like that?  
\(^5\) So far, so good.

What is the value of  
\(\{7,5\} \cup (((\{4\} \setminus \{3\}) \cap \{5\}))\)?  
\(^6\) \{7,5\}.

What is the value of  
\((\cup \{7,5\} (\cap (\setminus \{4\} \{3\}) \{5\}))\)?  
\(^7\) \{7,5\},  
we just went from infix to prefix notation.

What is the value of  
\((+ \{7,5\} (\times (\setminus \{4\} \{3\}) \{5\}))\)?  
\(^8\) \{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)))?

Where do the constructors come from? A datatype and its variants that represent set expressions.

Do you still like it? Sure, why not.

Does the arithmetic expression look like the set expression? Yes, they look the same except for the constants:
new Plus(
  new Const(●),
  new Prod(
    new Diff(
      new Const(●),
      new Const(●)))),
new Const(●)).

Let's say that an expression is either
  a Plus(expr₁, expr₂),
  a Diff(expr₁, expr₂),
  a Prod(expr₁, expr₂), or
  a constant,
where expr₁ and expr₂ stand for arbitrary expressions. What should be the visitor interface?

That's a tricky question.

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

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

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

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

class Diff extends ExprD {
    ExprD l;
    ExprD r;
    Diff(ExprD _l, ExprD _r) {
        l = _l;
        r = _r;
    }
    Object accept(ExprVisitor ask) {
        return ask.forDiff(l, r);
    }
}

class Prod extends ExprD {
    ExprD l;
    ExprD r;
    Prod(ExprD _l, ExprD _r) {
        l = _l;
        r = _r;
    }
    Object accept(ExprVisitor ask) {
        return ask.forProd(l, r);
    }
}

class Const extends ExprD {
    Object c;
    Const(Object _c) {
        c = _c;
    }
    Object accept(ExprVisitor ask) {
        return ask.forConst(c);
    }
}
```
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).intValueO
+
new Integer(2).intValueO?

How do we turn that into an Integer?

Okay, so here is a skeleton of IntEvalV

```java
class IntEvalV implements ExprVisitorT {
    public Object forPlus(Expr l, Expr r) {
        return plus(l.accept(this),
                   r.accept(this));
    }
    public Object forDiff(Expr l, Expr r) {
        return diff(l.accept(this),
                    r.accept(this));
    }
    public Object forProd(Expr l, Expr r) {
        return prod(l.accept(this),
                    r.accept(this));
    }
    public Object forConst(Object c) {
        return c;
    }
    Object plus(_1 l, _2 r) {
        return ___3;
    }
    Object diff(_1 l, _2 r) {
        return ___4;
    }
    Object prod(_1 l, _2 r) {
        return ___5;
    }
}
```

Yes, we can. It must have four methods, one per variant, and it is like OccursV from the previous chapter.

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

An int, what else?

We use new Integer(…).

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

It consumes two Expr^D_s, determines their respective values, and pluses them.

How are the values represented?  

As Objects, because we are using our most general kind of (and most recent) visitor.

So what kind of values must plus consume?  

Objects, because that's what 

l.accept(this)  

and  

r.accept(this)  

produce.

What must we put in the first and second blanks?  

Object.

Can we add Objects?  

No, we must convert them to Integers first and extract their underlying ints.

Can we convert all Objects to Integers?  

No, but all Objects produced by IntEval^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())?

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.

What should we do?

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

```java
new Integer(
    ((Integer)l).intValue() +
    ((Integer)r).intValue());
```

Complete the definition now.

And their set expressions, too.

Now it's easy. Here we go.

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

Copyrighted Material

Chapter 8
That one was pretty easy, wasn't it?  

Yes. Let’s implement an ExprVisitor\textsuperscript{I} for sets.

What do we need to implement one for sets?  

We certainly need methods for \textit{plusing}, \textit{diffing}, and \textit{prod}ing sets.

That's correct, and here is everything.

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

We use our words:  

"As its name says, \textit{add} adds an element to a set. If the element is a \textit{member} of the set, the set remains the same; otherwise, a \textit{new} set is constructed with \textit{Add}.”

Why is this so tricky?  

Constructors always construct, and \textit{add} does not always construct.

Do we need to understand that?  

Not now, but feel free to absorb it when you have the time.
Define the variants Empty and Add for SetD. Here we go.

```java
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?  

Not now, but feel free to think about them when you have the time. We haven’t even used visitors to define operations for union, set-difference, and intersection, but we trust you can.

What do we have to change in IntEval^V to obtain SetEval^V, an evaluator for set expressions?

Not much, just plus, diff, and prod.

How should we do that?

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

That’s the worst way of doing that.

What?

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

That’s true. If we copied the definition and changed it, we would have identical copies of 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?

That part is easy:

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

and

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

and

Object prod(Object l, Object r) {
    return ((Set^D)l).prod((Set^D)r); }.
Very good, and if we define $\text{SetEval}^V$ as an extension of $\text{IntEval}^V$, that's all we have to put inside of $\text{SetEval}^V$.

```
class SetEval^V extends IntEval^V {
   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 $\text{equals}$?

Now that's much easier than copying and modifying.

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

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

Three.

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

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

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

It doesn't say so.

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

It says so.

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

It says so.

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

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

Interesting question. How does this work now?

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

It is a Prod and therefore an ExprD

And what does accept consume?

An instance of SetEvalV, but its type is ExprVisitorT

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

That's what we need to determine the value of next, because it is
\[
\text{ask.forProd(l,r),}
\]

with ask, l, and r replaced by what they stand for.

Where is the definition of SetEvalV's method forProd?

It is in IntEvalV

Suppose we had the values of
\[
\text{new Const(new Empty())}
\text{.add(new Integer(7)))}
\text{.accept(this)}
\]

and
\[
\text{new Const(new Empty())}
\text{.add(new Integer(3)))}
\text{.accept(this)}.
\]

What would we have to evaluate next?

If their values were A and B, we would have to determine the value of
\[
\text{prod}(A,B).
\]
Isn't that strange?

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

It is this object.

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

Good. And now what?

The values are obviously

\[
\text{new Empty().add(new Integer(7))}
\]

and

\[
\text{new Empty().add(new Integer(3)).}
\]

Where is the definition of forConst that determines these values?

Here is the next expression in our sequence:

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

Where does prod come from?

---

57 Why?

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

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

60 That clarifies things.

61 Now we still need to determine the values of

\[
\text{new Const(new Empty().add(new Integer(7))).accept(this)}
\]

and

\[
\text{new Const(new Empty().add(new Integer(3))).accept(this)}.\]

62 It, too, is in \( \text{IntEval^V} \).

63 The object is an instance of \( \text{SetEval^V} \), which overrides the \( \text{prod} \) method in \( \text{IntEval^V} \) with its own.
What next?

Is `new EmptyO.add(new Integer(7))` an instance of `SetV`?

And how about `new EmptyO.add(new Integer(3))`?

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

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

And what does `prod` do?

We overrode that, too.

Is it natural that `SetEval^V` extends `IntEval^V`?

---

64 Next we need to determine the value of

\[ ((\text{Set}^D) (\text{new Empty}(\text{add(new Integer(7))}))) \]

\[ .\text{prod}((\text{Set}^D) \text{new Empty}(\text{add(new Integer(3))})) , \]

because it is

\[ ((\text{Set}^D) (l.\text{accept}(\text{this}))) \]

\[ .\text{prod}((\text{Set}^D) r.\text{accept}(\text{this})) \]

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

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

66 It's the same.

67 This example makes the need for conversions obvious again.

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

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?

70 Thanks, guys.

71 No, not at all.
Why did we do that?

But just because something works, it doesn’t mean it’s rational.

What distinguishes IntEvalV from SetEvalV?

What are the pieces that they have in common?

Good. Here is how we express that.

```java
abstract class EvalV implements ExprVisitor {
  public Object forPlus(Expr l, Expr r) {
    return plus(l.accept(this), r.accept(this));
  }
  public Object forDiff(Expr l, Expr r) {
    return diff(l.accept(this), r.accept(this));
  }
  public Object forProd(Expr l, Expr r) {
    return prod(l.accept(this), r.accept(this));
  }
  public Object forConst(Object c) {
    return c;
  }
  abstract Object plus(Object l, Object r);
  abstract Object diff(Object l, Object r);
  abstract Object prod(Object l, Object r);
}
```

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.

Yes, let’s do better. We have defined all these classes ourselves, so we are free to rearrange them any way we want.

The methods plus, diff, and prod.

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

Isn’t this abstract class like PointP?
Yes, we can think of it as a datatype for \texttt{EvarD} 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 \texttt{IntEvalV} extending \texttt{EvalD}.

```java
class IntEvalV extends EvalD {
    Object plus(Object l, Object r) {
        return new Integer(((Integer)l).intValue() + ((Integer)r).intValue());
    }
    Object diff(Object l, Object r) {
        return new Integer(((Integer)l).intValue() - ((Integer)r).intValue());
    }
    Object prod(Object l, Object r) {
        return new Integer(((Integer)l).intValue() * ((Integer)r).intValue());
    }
}
```

Define \texttt{SetEvalV}.

Is it natural for two evaluators to be on the same footing?  Much more so than one extending the other.

Time for supper.  If you are neither hungry nor tired, you may continue.
Remember Subst\textsuperscript{V} from chapter 6?

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

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

What do the two visitors have in common? Many things: \(n\), \(o\), and \textit{forBot}.

Where do they differ? They differ in \textit{forTop}, but \textit{LtdSubst} also has an extra field.

And where do we put the pieces that two classes have in common? We put them into an abstract class.

What else does the abstract class contain? It specifies the pieces that are different if they are needed for all extensions.
Define the abstract class Subst\(^D\), which contains all the common pieces and specifies what a concrete pie substituter must contain in addition.

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

```
abstract class Subst\(^D\)
    implements PieVisitor\(^D\) {
    Object n;
    Object o;
    public Pie\(^D\) forBot() {
        return new Bot();
    }
    public
        abstract Pie\(^D\) forTop(Object t,Pie\(^D\) r);
}
```

We can define Subst\(^V\) by extending Subst\(^D\).

```
class Subst\(^V\) extends Subst\(^D\) {
    Subst\(^V\)(Object \_n,Object \_o) {
        n = \_n;
        o = \_o;
    }

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

Define LtdSubst\(^V\).

```
class LtdSubst\(^V\) extends Subst\(^D\) {
    int c;
    LtdSubst\(^V\)(int \_c,Object \_n,Object \_o) {
        n = \_n;
        o = \_o;
        c = \_c;
    }

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

Do the two remaining classes still have things in common?

It also extends Subst\(^D\).

```
class LtdSubst\(^V\) extends Subst\(^D\) {
    int c;
    LtdSubst\(^V\)(int \_c,Object \_n,Object \_o) {
        n = \_n;
        o = \_o;
        c = \_c;
    }

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

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:

```java
abstract class SubstD implements PieVisitorD {
    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:

```java
public PieV forBot() {
    return new Bot();
}
```

We must use `super` in the constructors:

```java
class SubstV extends SubstD {
    SubstV(Object _n, Object _o) {
        super(_n, _o);
    }

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

```java
class LtdSubstV extends SubstD {
    int C;
    LtdSubstV(int _c, Object _n, Object _o) {
        super(_n, _o);
        C = _c;
    }

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

Was that first part easy? As pie.
That's neat. How about some art work?  

No, but the picture captures the important relationships.  

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

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

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

As we just said, $c$ is an addition and for $\text{Top}$ is different.

The Eighth Bit of Advice

*When extending a class, use overriding to enrich its functionality.*
Here is the good old definition of Subst\textsuperscript{V} from chapter 6 one more time.

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

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

Define LtdSubst\textsuperscript{V} as an extension of Subst\textsuperscript{V}.

```java
class LtdSubst\textsuperscript{V} extends Subst\textsuperscript{V} {
    int c;
    LtdSubst\textsuperscript{V}(int \_c, Object \_n, Object \_o) {
        super(\_n, \_o);
        c = \_c;
    }

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

Let's draw a picture.

![Diagram](image)

The rest follows naturally, just as with the evaluators and the previous version of these two classes.

```
subtext
```

Define LtdSubst\textsuperscript{V} as an extension of Subst\textsuperscript{V}.

Let's draw a picture.

![Diagram](image)

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

```java
abstract class Point\textsuperscript{D} {
    int x;
    int y;
    Point\textsuperscript{D}(int \_x, int \_y) {
        x = \_x;
        y = \_y;
    }

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

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

    abstract int distanceToO();
}
```

It has been a long time since we discussed the datatype Point\textsuperscript{D} and its variants, but they are not that easy to forget.

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

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

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

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

Good. Take a look at this extension of ManhattanPt.

```java
class ShadowedManhattanPt
    extends ManhattanPt {
    int \_\Delta_x;
    int \_\Delta_y;
    ShadowedManhattanPt(int \_x,
        int \_y,
        int \_\Delta_x,
        int \_\Delta_y) {
        super(\_x, \_y);
        \_\Delta_x = \_\Delta_x;
        \_\Delta_y = \_\Delta_y;
    }

    int distanceToO() {
        return super.distanceToO() + \_\Delta_x + \_\Delta_y;
    }
}
```

What is unusual about the constructor?

It uses
\[
\Delta_x = \_\Delta_x;
\Delta_y = \_\Delta_y;
\]
in addition to \texttt{super(\_x, \_y)}.

Be a Good Visitor

\hspace{10000} Copyrighted Material 139
And what does that mean?  
Okay. So what is a ShadowedManhattanPt?  
Is this a ShadowedManhattanPt:  
new ShadowedManhattanPt(2,3,1,O)?  
What is unusual about distanceToO?  
Here, super.distanceToO refers to the method definition of distanceToO that is relevant in the class that ShadowedManhattanPt extends.  
Correct. But what would we have done if ManhattanPt had not defined distanceToO?  
Yes, and so on. What is the value of new ShadowedManhattanPt(2,3,1,0) .distanceToO()?

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.

4 It is a ManhattanPt with two additional fields: $\Delta_x$ and $\Delta_y$. These two represent the information that determines how far the shadow is from the point with the fields $x$ and $y$.

5 Yes.

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?

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

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

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.

```java
class ShadowedCartesianPt
    extends CartesianPt {
    int Δx;
    int Δy;
    ShadowedCartesianPt(int _x, int _y, int Δx, int Δy) {
        super(_x, _y);
        Δx = _Δx;
        Δy = _Δy;
    }

    int distanceToO() {
        return super.distanceToO() + \sqrt{Δ_x^2 + Δ_y^2};
    }
}
```

What is unusual about the constructor?

Is this a ShadowedCartesianPt:

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

And what is the value of

```java
new ShadowedCartesianPt(12,5,3,4).distanceToO()?
```

What do we expect?

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{Δ_x^2 + Δ_y^2}

13 17, obviously.
Why 17? Because we need to think of this point as if it were
new CartesianPt(15,9).

We need to add $\Delta_x$ to $x$ and $\Delta_y$ to $y$ when
we think of a ShadowedCartesianPt.

And indeed, the value of
new CartesianPt(15,9)
.distanceToO()
is 17.

Does this explain how $\text{distanceToO}$ should
measure the distance of a
ShadowedCartesianPt to the origin?

Completely. It should make a new
CartesianPt by adding the corresponding
fields and should then measure the distance
of that new point to the origin.

Revise the definition of ShadowedCartesianPt
accordingly.

```
class ShadowedCartesianPt
    extends CartesianPt {
    int $\Delta_x$;
    int $\Delta_y$;
    ShadowedCartesianPt(int $\_x$,
                        int $\_y$,
                        int $\Delta_x$,
                        int $\Delta_y$) {
        super($\_x$, $\_y$);
        $\Delta_x$ = $\Delta_x$;
        $\Delta_y$ = $\Delta_y$; }

    int distanceToO() {
        return 
            new CartesianPt($x + \Delta_x$, $y + \Delta_y$)
                .distanceToO(); }
    }
```

Okay.

Do we still need the new CartesianPt after
$\text{distanceToO}$ has determined the distance?

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

¹ And neither does Java. Object-oriented languages manage
memory so that programmers can focus on the difficult parts
of design and implementation.
Correct. What is the value of 
new CartesianPt(3,4) 
.closerToO( 
new ShadowedCartesianPt(1,5,1,2))?

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

How did we determine that value?

That’s obvious.

Is the rest of this chapter obvious, too?

What?

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

Oh. Well, that makes the hint obvious.

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

Fine.

Are sandwiches square meals for you?

They can be well-rounded.

Here are circles and squares.

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

class Square extends ShapeD {
    int s;
    Square(int _s) {
        s = _s;
    }
    boolean accept(ShapeVisitorI ask) {
        return ask.forSquare(s);
    }
}
```

Then this must be the datatype that goes with it.

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

Be a Good Visitor

Copyrighted Material 143
Very good. We also need an interface, and here it is.

```java
interface ShapeVisitor<T> {
    boolean forCircle(int r);
    boolean forSquare(int s);
    boolean forTrans(PointD q, ShapeD s);
}
```

Yes and we will need this third variant.

```java
class Trans1 extends ShapeD {
    PointD q;
    ShapeD s;
    Trans(PointD _q, ShapeD _s) {
        q = -q;
        s = -s;
    }

    boolean accept(ShapeVisitor<T> ask) {
        return ask.forTrans(q, s);
    }
}
```

1 A better name is Translation.

Let's create a circle.

26 It suggests that there is another variant: Trans.

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

28 No problem:
   ```java
   new Circle(10).
   ```

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

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

31 What does that mean?
<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suppose we wish to determine whether some CartesianPt is inside of the circle?</td>
<td>And how about the square?</td>
</tr>
<tr>
<td>Pick one.</td>
<td>That will do. Is the CartesianPt with ( x ) coordinate 10 and ( y ) coordinate 10 inside the square?</td>
</tr>
<tr>
<td>And how about the circle?</td>
<td>And how about the circle?</td>
</tr>
<tr>
<td>Are all circles and squares located at the origin?</td>
<td>This is where Trans comes in. What is new Trans(</td>
</tr>
<tr>
<td></td>
<td>new CartesianPt(5,6),</td>
</tr>
<tr>
<td></td>
<td>new Circle(10))?</td>
</tr>
<tr>
<td>How do we place a square's southwest corner at new CartesianPt(5,6)?</td>
<td>How do we place a square's southwest corner at new CartesianPt(5,6)?</td>
</tr>
<tr>
<td></td>
<td>New CartesianPt(10),</td>
</tr>
<tr>
<td></td>
<td>new Square(10)).</td>
</tr>
<tr>
<td>Is new CartesianPt(10,10) inside either the circle or the square that we just referred to?</td>
<td>Is new CartesianPt(10,10) inside either the circle or the square that we just referred to?</td>
</tr>
<tr>
<td></td>
<td>It is inside both of them.</td>
</tr>
</tbody>
</table>

In that case, we must think of the circle as being drawn around the origin.

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

Let's say the square's southwest corner sits on the origin.

Yes, it is. but barely.

Certainly not, because the circle's radius is 10, but the distance of the point to the origin is 14.

We have no choice so far, because Circle and Square only contain one field each: the radius and the length of a side, respectively.

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

Also with Trans: new Trans( new CartesianPt(5,6), new Square(10)).
How do we determine whether some point is inside a circle? If the circle is located at the origin, it is simple. We determine the distance of the point to the origin and whether it is smaller than the radius.

How do we determine whether some point is inside a square? If the square is located at the origin, it is simple. We check whether the point's $x$ coordinate is between 0 and $s$, the length of the side of the square.

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

Aren't we on a roll? We have only done the easy stuff so far. It is not clear how to check these things when the circle or the square are not located at the origin.

Let's take a look at our circle around \texttt{new CartesianPt(5,6)} again. Can we think of this point as the origin? We can if we translate all other points by an appropriate amount.

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

How could we translate the points by an appropriate amount? We could subtract the appropriate amount from each point.

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

```java
class HasPt\(^V\) implements ShapeVisitor\(^I\) {
    Point\(^P\) p;
    HasPt\(^V\)(Point\(^P\) _p) {
        p = _p;
    }

    public boolean forCircle(int r) {
        return p.distanceToO() \leq r; }
    public boolean forSquare(int s) {
        if (p.x \leq s)
            return (p.y \leq s);
        else
            return false;
    }
    public boolean forTrans(Point\(^P\) q,Shape\(^P\) s) {
        return s.accept(  
            new HasPt\(^V\)(p.minus(q))); }
}
```

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

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:
\[
\text{new Trans(}
\quad \text{new CartesianPt(5,4),}
\quad \text{new Trans(}
\quad \quad \text{new CartesianPt(5,6),}
\quad \quad \text{new Circle(10))})
\quad .\text{accept(}
\quad \quad \quad \text{new HasPtV(new CartesianPt(10,10)))?}
\]

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

But what is the value?

First, we have to find out whether
\[
\text{new Trans(}
\quad \text{new CartesianPt(5,6),}
\quad \text{new Circle(10))}
\quad .\text{accept(}
\quad \quad \text{new HasPtV(new CartesianPt(5,6)))}
\]

is true or false.

And then?

Second, we need to look at
\[
\text{new Circle(10)}
\quad .\text{accept(}
\quad \quad \text{new HasPtV(new CartesianPt(0,0))),}
\]

but the value of this is obviously true.

Very good. Can we nest Trans three times?

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

Ready to begin?

What? Wasn’t that it?

No. The exciting part is about to start.

We are all eyes.

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

It becomes a square, obviously.

And the orange on top?

A circle, Transed appropriately.
Can we think of the two objects as one?

Here is our way.

class Union extends Shapev {
    Shapev s;
    Shapev t;
    Union(Shapev _s,Shapev _t) {
        s = _s;
        t = _t;
    }
    boolean accept(ShapeVisitor ask) {
        return _______; }
}

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

So what should we do now?

Correct, except that we won't allow ourselves to change ShapeVisitor

Just to make the problem more interesting.
We would be stuck, but fortunately we can extend interfaces. Take a look at this.

```java
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 ShapeVisitorI and the additional one named forUnion.

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

Does that mean accept in Union should receive a UnionVisitorI, so that it can use the forUnion method?

Yes it should, but because UnionVisitorI extends ShapeVisitorI, it is also a ShapeVisitorI.

We have been here before. Our accept method must consume a ShapeVisitorI and fortunately every UnionVisitorI implements a ShapeVisitorI, too. But if we know that accept consumes a UnionVisitorI, we can convert the ShapeVisitorI to a UnionVisitorI and invoke the forUnion method.

Perfect reasoning. Here is the completed definition of Union.

```java
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);
    }
}
```

And it makes complete sense.
Let’s create a Union shape. 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 whether
new CartesianPt(12,16)
is inside?

Could it be a UnionVisitor? No. It does not provide the method forUnion.

Define UnionHasPt\(^V\), which extends HasPt\(^V\) with an appropriate method forUnion.

```java
class UnionHasPt\(^V\) extends HasPt\(^V\) {
    UnionHasPt\(^V\)(Point\(^D\) \_p) {
        super(\_p);
    }

    boolean forUnion(Shape\(^D\) s,Shape\(^D\) t) {
        if\(^1\) (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? Of course, we just put it in.
Is UnionHasPt\textsuperscript{V} a UnionVisitor\textsuperscript{T}?

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

```java
class UnionHasPt\textsuperscript{V}
    extends HasPt\textsuperscript{V}
    implements UnionVisitor\textsuperscript{T} {
        UnionHasPt\textsuperscript{V}(Point\textsuperscript{D} _p) {
            super(_p);
        }
        public boolean forUnion(Shape\textsuperscript{D} s, Shape\textsuperscript{D} t) { 
            if (s.accept(this))
                return true;
            else
                return t.accept(this);
        }
    }
```

76 It provides the required methods: \textit{forCircle}, \textit{forSquare}, \textit{forTrans}, and \textit{forUnion}.

77 The first two additional words have an obvious meaning. They explicitly say that this visitor provides the services of UnionVisitor\textsuperscript{T}. And, as we have said before, the addition of public is necessary, because this visitor \textit{implements} an interface.

Good try. Let's see whether it works. What should be the value of

```java
new Trans(
    new CartesianPt(3,7),
    new Union(
        new Square(10),
        new Circle(10)))
    .accept(
        new UnionHasPt\textsuperscript{V}(
            new CartesianPt(13,17)))?
```

We know how \textit{forTrans} works, so we're really asking whether

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

is inside the Union shape.

78

So?

Which means that we're asking whether

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

is inside of

```java
new Square(10)
```

or inside of

```java
new Circle(10).
```
Okay. And what should be the answer? Let’s see whether the value of
new Trans(
  new CartesianPt(3,7),
  new Union(
    new Square(10),
    new Circle(10)))
.accept(
  new UnionHasPt^v(
    new CartesianPt(13,17)))
is true?

And? Usually we start by determining what kind of object we are working with.

How did we construct this shape? With Trans.

Which method should we use on it? forTrans, of course.

Where is forTrans defined? It is defined in HasPt^v

So what should we do now? 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))? It’s a Shape^D.
How did we construct this ShapeD?

With Union.

So which method should we use on it?

forUnion, of course.

How do we find the appropriate forUnion method?

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

\[
\text{new HasPtV(}
\text{new CartesianPt(10,10))}
\]

is a UnionVisitor and then invoke its forUnion.

Is an instance of HasPtV a UnionVisitor?

No!

Does it contain a method forUnion?

No!

Then what is the value of

\[
\text{new Union(}
\text{new Square(10),}
\text{new Circle(10))}
\text{.accept(}
\text{new HasPtV(}
\text{new CartesianPt(10,10))))?}
\]

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

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

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

What do we do next?

Relax. Read a novel. Take a nap.

Which of those is best?

You guessed it: whatever you did is best.

We should have prepared this extension in a better way.

How could we have done that?
Here is the definition of \texttt{HasPtV} that we should have provided if we wanted to extend it without making changes.

```java
class HasPtV implements ShapeVisitor \{ 
PointD p;
HasPtV(PointD -p) 
{ 
p = -p;
ShapeVisitor newHasPt(PointD p) 
{ 
return new HasPtV(p);
}

public boolean forCircle(int r) 
{ 
return p.distanceToO() \leq r;
}
public boolean forSquare(int s) 
{ 
if (p.x \leq s) 
return (p.y \leq 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 \texttt{newHasPt} produce?

A new \texttt{ShapeVisitor}, as its interface implies.

And how does it produce that?

By constructing a \texttt{new} instance of \texttt{HasPtV}

Is \texttt{newHasPt} like a constructor?

It is virtually indistinguishable from a constructor, which is why it is above the line that separates constructors from methods.
Does that mean the new definition of $\text{HasPt}^\vee$ and the previous one are really the same?\footnote{A functional programmer would say that $\text{newHasPt}$ and $\text{HasPtV}$ are \(\eta\)-equivalent.}

They are mostly indistinguishable. Both for $\text{Trans}$, the one in the previous and the one in the new definition of $\text{HasPt}^\vee$, produce the same values when they consume the same values.

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

That's not obvious.

Can we override $\text{newHasPt}$ when we extend $\text{HasPt}^\vee$?

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

Let's override $\text{newHasPt}$ in $\text{UnionHasPt}^\vee$.

When we override it, we need to make sure it produces a $\text{ShapeVisitor}^I$.

That's true. Should it produce a $\text{HasPt}^\vee$ or a $\text{UnionHasPt}^\vee$?

The latter. Then for $\text{Trans}$ in $\text{HasPt}^\vee$ keeps producing a $\text{UnionHasPt}^\vee$, if we start with a $\text{UnionHasPt}^\vee$.

Good answer. Should we repeat it?

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 (overrideable) method so that visitors can be extended, too.*
And that’s exactly what we need. Revise the definition of UnionHasPtV.\textsuperscript{1}

\begin{verbatim}
class UnionHasPtV
    extends HasPtV
    implements UnionVisitorT {
    UnionHasPtV(PointD _p) {
        super(_p);
    }
    ShapeVisitorT 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);
        }
}
\end{verbatim}

\textsuperscript{1} The is an instance of the factory method pattern [4].

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

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

What does the box mean?

Everything outside of the box is what we designed originally and considered to be unchangeable; everything inside is our extension.
Does the picture convey the key idea of this chapter?

No. It does not show the addition of a constructor-like method to HasPt\(^V\) and how it is overridden in UnionHasPt\(^V\).

Is anything missing?

Square, but that’s okay.

Let’s see whether this definition works.

What is the value of

\[
\text{new Trans(}
\text{new CartesianPt(3,7),}
\text{new Union(}
\text{new Square(10),}
\text{new Circle(10))})
\text{.accept(}
\text{new UnionHasPt\(^V\)(}
\text{new CartesianPt(13,17))))?}
\]

We remember that the shape was built with Trans.

Which method should we use on it?

forTrans, of course.

Where is forTrans defined?

It is defined in HasPt\(^V\).

So what should we do now?

We should determine the value of

\[
\text{new Union(}
\text{new Square(10),}
\text{new Circle(10))}
\text{.accept(}
\text{this.newHasPt(}
\text{new CartesianPt(10,10))}).
\]

What is this?

The current visitor, of course.

And how does that work?

We determine the value of

\[
\text{this.newHasPt(}
\text{new CartesianPt(10,10))}
\text{and then use accept for the rest.}
\]
And what do we create?

What is the value of
\[
\text{new Union(}
  \text{new Square}(10),
  \text{new Circle}(10))
\]
\[.\text{accept(}
  \text{new UnionHasPtV(}
    \text{new CartesianPt}(10,10)))\]?

How do we do that?

We first determine the value of
\[
\text{new Square}(10)
\]
\[.\text{accept(}
  \text{new UnionHasPtV(}
    \text{new CartesianPt}(10,10)))\].

If it is true, we’re done.

Is it true?

It is. So we’re done and we got the value we expected.

Are we happy now?

Ecstatic.

Is it good to have extensible definitions?

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?

Yes, we can and should always do so.

And why is that?

Because no program is ever finished.

Are you hungry yet?

Are our meals ever finished?
10.
The State of Things to Come

Copyrighted Material
Have you ever wondered where the pizza pies come from?

You should have, because someone needs to make the pie.

Here is our pizza pieman.

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

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 before?

We have seen them.

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

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

Let's not worry about them for a while.

Fine, but they are weird.
Here is the interface for Pieman

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

Isn’t it missing p?

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

Here are PieVisitorI and PieD

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

abstract class PieD {
    abstract Object accept(PieVisitorI ask);
}
```

Define Bot and Top.

They are very familiar.

```java
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 Occurs\(^V\). It counts how often some topping occurs on a pie.

```java
class Occurs\(^V\) implements PieVisitor\(^I\) {
    Object a;
    Occurs\(^V\)(Object \_a) {
        a = \_a;
    }

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

    public Object forTop(Object t, Pie\(^D\) r) {
        if (t.equals(a))
            return new Integer(((Integer) r.accept(this)).intValue() + 1);
        else
            return r.accept(this);
    }
}
```

Great! Now we have almost all the visitors for our pieman. Define Rem\(^V\), which removes a topping from a pie.

```java
class Rem\(^V\) implements PieVisitor\(^I\) {
    Object o;
    Rem\(^V\)(Object \_o) {
        o = \_o;
    }

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

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

We remember that one, too.

```java
class Subst\(^V\) implements PieVisitor\(^I\) {
    Object n;
    Object o;
    Subst\(^V\)(Object \_n, Object \_o) {
        n = \_n;
        o = \_o;
    }

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

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

And this little visitor substitutes one good topping for another.
Now we are ready to talk. What is the value of

\[ \text{new Pieman}^M().\text{occTop(new Anchovy())}? \]

Which pie?

The pie named \( p \) in the new \( \text{Pieman}^M \).

And how many anchovies are on that pie?

None.

And what is the value of

\[ \text{new Pieman}^M().\text{addTop(new Anchovy())}? \]

That's where those stand-alone semicolons come in again. They were never explained.

True. If we wish to determine the value of

\[ \text{new Pieman}^M().\text{addTop(new Anchovy())}, \]

we must understand what

\[
\begin{align*}
p &= \text{new Top(new Anchovy(),p)} \\
\text{return occTop(new Anchovy())}
\end{align*}
\]

means?

That's right. But that's what happens when you have one too many double espressos.

Here it means that \( p \) changes and that future references to \( p \) reflect the change.

And the change is that \( p \) has a new topping, right?

When does the future begin?

Does it begin below the stand-alone semicolon?

That's precisely what a stand-alone semicolon means. Now do we know what

\[ \text{return occTop(new Anchovy())} \]

produces?

It produces the number of anchovies on \( p \).
And how many are there?  

And now what is the value of  
\texttt{new Pieman}^{\mathcal{M}}() \texttt{.addTop(new Anchovy())}?  

No, it's not. Take a close look. We created a \texttt{new} pieman, and that pieman added only one anchovy to his \texttt{p}.

Yes, there is. Take a look at this:  
\texttt{Pieman}^{\mathcal{T}} \ y = \texttt{new Pieman}^{\mathcal{M}}().

What is the value of  
\texttt{y.addTop(new Anchovy())}?  

And now what is the value of  
\texttt{y.substTop(new Tuna(),new Anchovy())}?  

Correct. So what is the value of  
\texttt{y.occTop(new Anchovy())}?  

Very good. And now take a look at this:  
\texttt{Pieman}^{\mathcal{T}} \ yy = \texttt{new Pieman}^{\mathcal{M}}().

What is the value of  
\texttt{yy.addTop(new Anchovy())}  
\texttt{;}  
\texttt{yy.addTop(new Anchovy())}  
\texttt{;}  
\texttt{yy.addTop(new Salmon())}  
\texttt{;}  

We added one, so the value is 1.

It's 2, isn't it?

Oh, isn't there a way to place several requests with the same pieman?

Okay, \texttt{y} stands for some pieman.

1. We know that.

Still 1. According to the rules of semicolon and =, this replaces all anchovies on \texttt{p} with tunas, changes \texttt{p}, and then counts how many tunas are on \texttt{p}.

0, because \texttt{y}'s pie no longer contains any anchovies.

What are the doing at the end?
Because this is only half of what we want to look at. Here is the other half:

```java
yy.addTop(new Tuna())
;
yy.addTop(new Tuna())
;
yy.substTop(new Tuna(), new Anchovy())?
```

And what is the value of

```java
yy.remTop(new Tuna())
```

after we are through with all that?

Does that mean `remTop` always produces 0?

Now what is the value of

```java
yy.occTop(new Salmon())
```

And how about

```java
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

```java
yy.p = new Bot()
```

and thus change a pieman like `yy`?

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.

It's 0, because `remTop` first removes all tunas and then counts how many there are left.

Yes, it always does.

1.

0, because `y` and `yy` are two different piemen.

No, it changed.

When we eat a pizza pie, we change, but we are still the same.

The `p` in `yy` changed, nothing else.

No, because `yy`'s type is `PiemanI`, `p` isn't available. Only `addTop`, `remTop`, `substTop`, and `occTop` are visible.
Isn't it good that we didn't include \( p \) in \( \Pieman^I \)?

Yes, with this trick we can prevent others from changing \( p \) (or parts of \( p \)) in strange ways. Everything is clear now.

Clear like soup?

Just like chicken soup.

Can we define a different version of \( \text{Subst}^V \) so that it changes toppings the way a pieman changes his pies?

We can't do that yet.

And that's what we discuss next. Do you need a break?

No, a cup of coffee will do.

Compare this new \( \text{PieVisitor}^I \) with the first one in this chapter.

It isn't all that different. A \( \text{PieVisitor}^I \) must still provide two methods: \( \text{forBot} \) and \( \text{forTop} \), except that the former now consumes a Bot and the latter a Top.

True. Here is the unchanged datatype.

The definition is straightforward.

Define the Bot variant.

We only have one instance of \( \text{Bot} \) when we use \( \text{forBot} \), namely \text{this}, so \( \text{forBot} \) is clearly supposed to consume \text{this}.
That’s progress. And that’s what happens in Interesting. Top, too.

class Top extends PieD {
    Object t;
    PieD r;
    Top(Object _t, PieD _r) {
        t = _t;
        r = _r;
    }
    Object accept(PieVisitor^ ask) {
        return ask.forTop(this);
    }
}

Modify this version of Occurs^V so that it implements the new PieVisitor^T

class Occurs^V implements PieVisitor^T {
    Object a;
    Occurs^V(Object _a) {
        a = _a;
    }
    public Object forBot() {
        return new Integer(0);
    }
    public Object forTop(Object t, PieD r) {
        if (t.equals(a))
            return new Integer(((Integer) (r.accept(this))).intValue() + 1);
        else
            return r.accept(this);
    }
}

How does forBot change?

The forBot method basically stays the same, but forTop changes somewhat.

class Occurs^V implements PieVisitor^T {
    Object a;
    Occurs^V(Object _a) {
        a = _a;
    }
    public Object forBot(Bot that) {
        return new Integer(0);
    }
    public Object forTop(Top that) {
        if (that.t.equals(a))
            return new Integer(((Integer) (that.r.accept(this))).intValue() + 1);
        else
            return that.r.accept(this);
    }
}

It now consumes a Bot, which is why we had to add (Bot that) behind its name.
How does \textit{forTop} change? 

It no longer receives the field values of the corresponding \textit{Top}. Instead it consumes the entire object, which makes the two fields available as \textit{that.t} and \textit{that.r}.

And? 

With that, we can replace the fields \textit{t} and \textit{r} with \textit{that.t} and \textit{that.r}.

Isn't that easy? 

This modification of \textit{Occurs$^V$} certainly is.

Then try \textit{Rem$^V$} 

It's easy; we use the same trick.

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

Do we need to do \textit{Subst$^V$}? 

Not really. It should be just like \textit{Rem$^V$}.

And indeed, it is. Happy now? 

So far, so good. But what's the point of this exercise?

Oh, Point$^D$s? They will show up later. 

Seriously.
Here is the point. What is new about this version of Subst\textsuperscript{V}?

```java
class Subst\textsuperscript{V} implements PieVisitor\textsuperscript{X} {
    Object n;
    Object o;
    Subst\textsuperscript{V}(Object \_n, Object \_o) {
        n = \_n;
        o = \_o;
    }
    public Object forBot(Bot that) {
        return that;
    }
    public Object forTop(Top that) {
        if (o.equals(that.t)) {
            that.t = n
            ;
            that.r.accept(this)
            ;
            return that;
        }
        else {
            that.r.accept(this)
            ;
            return that;
        }
    }
}
```

Don't they say “no news is good news?”

Yes, because we want to define a version of Subst\textsuperscript{V} that modifies toppings without constructing a new pie.

What do the methods of Subst\textsuperscript{V} always return?

They always return that, which is the object that they consume.

So how do they substitute toppings?

By changing the that before they return it. Specifically, they change the \textit{t} field of that to \textit{n} when it equals \textit{o}.
Correct. And from here on, that.t holds the new topping. What is that.r.accept(this) about?

Is there anything else to say about the new SubstV?

Do we have to change PiemanM?

Is it truly safe to modify the toppings of a pie?

Can we do LtdSubstV now without creating new instances of LtdSubstV or Top?

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 PointV {
    int x;
    int y;
    PointV(int _x, int _y) {
        x = _x;
        y = _y;
    }
}

boolean closerToO(PointV p) {
    return distanceToO() ≤ p.distanceToO();
}

PointV minus(PointV p) {
    return new CartesianPt(x - p.x, y - p.y);
}

abstract int distanceToO();

The datatype has three extensions.

class CartesianPt extends PointD {
    CartesianPt(int _x, int _y) {
        super(_x, _y);
    }
    int distanceToO() {
        return \sqrt{x^2 + y^2};
    }
}

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

class ShadowedManhattanPt extends ManhattanPt {
    int Δx;
    int Δy;
    ShadowedManhattanPt(int _x,
        int _y,
        int Δx,
        int Δy) {
        super(_x, _y);
        Δx = _Δx;
        Δy = _Δy;
    }
    int distanceToO() {
        return super.distanceToO() + Δx + Δy;
    }
}

 Aren’t we missing a variant?  

Yes, we are missing ShadowedCartesianPt.

Copyrighted Material  Chapter 10
Good enough. We won’t need it. Here is one 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.

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

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

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

    int moveBy(int \_x,int \_y) {
        x = x + \_x;
        y = y + \_y;
        return distanceToO();
    }

    abstract int distanceToO();
}
```

Let `ptChild` stand for

```
new ManhattanPt(1,4).
```

What is the value of `ptChild.distanceToO()`?
What is the value of 
\( ptChild.moveBy(2,8) \)?

Good. Now let’s watch a child with a helium-filled balloon that casts a shadow. Let \( ptChildBalloon \) be
\[
\text{new ShadowedManhattanPt(1,4,1,1)}.
\]
What is the value of 
\( ptChildBalloon.distanceToO() \)?

What is the value of 
\( ptChildBalloon.moveBy(2,8) \)?

Did the balloon move, too?

Isn’t that powerful?

The more things change, the cheaper our desserts get.

Correct but now we are through and it is time to go out and to celebrate with a grand dinner.

Yes, but to get to the dessert, we had to work quite hard.

Don’t forget to leave a tip.
Commencement

Copyrighted Material
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


Index

Add, 124
addTop, 161
Anchovy, 43, 53, 64, 65, 69, 72
Apple, 100

Base, 10
bHasFruitV, 104
blsFlatV, 102
blsSplitV, 102, 103
Bot, 69, 78, 83, 85, 86, 91, 93, 162, 167
Brass, 29
Bud, 100, 101, 107, 110, 112
bTreeVisitorI, 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
EvalD, 130
ExprD, 119
ExprVisitorI, 118

Fig, 100
FishD, 69, 72
Flat, 100, 101, 108, 110, 112
FruitD, 100

Gold, 29

HasPtV, 147, 155
Holder, 28, 36

iHeightV, 108
IntEvalv, 120, 122, 131

KebabD, 28, 36
Lamb, 16, 27, 28, 36, 59, 62
LayerD, 10
Lemon, 100
LtdSubstV, 95, 97, 132-136

ManhattanPt, 5, 13, 38, 39, 139, 172
mem, 124
minus, 139
moveBy, 173

newHasPt, 155, 157
NumD, 7, 79

occoTop, 161
OccursV, 114, 163, 168
Olive, 43, 53, 64, 65
OneMoreThan, 7, 79
Onion, 16, 27, 28, 36, 59, 62
onlyOnions, 18, 27
OnlyOnionsV, 57, 59

Peach, 100
Pear, 100
Pepper, 4, 37
PieD, 69, 78, 83, 85, 86, 91, 93, 162, 167
PieVisitorI, 92, 162, 167
PiemanI, 162
PiemanM, 161
PizzaD, 43, 53, 63-65
PlateD, 29
plus, 124
Plus, 119
PointD, 5, 13, 40, 139, 172, 173
prod, 124
Prod, 119

Copyrighted Material
Radish, 28, 36
remA, 45, 49, 70
RemAv, 66, 71, 73
RemFishv, 74
Remv, 77, 87, 93, 163, 169
RemlntV, 76
remTop, 161
RodD, 29
Sabre, 29
Sage, 5
Salmon, 69, 72
Salt, 4
Sausage, 43, 53, 63–65
SeasoningD, 4
SetD, 123
SetEvalv, 126, 131
ShadowedCartesianPt, 141, 142
ShadowedManhattanPt, 139, 172
Shallot, 28, 36
ShapeD, 143
ShapeVisitorZ, 144
ShishD, 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
SubAbCv, 66
substD, 133, 134
SubstFishv, 82
Substv, 83, 87, 89, 94, 132–136, 163, 170
SubstIntv, 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
TopAwCv, 66
Trans, 144
TreeD, 100, 101, 107, 109, 112
TreeVisitorZ, 112
tSubstv, 111
tTreeVisitorZ, 109
Tuna, 69, 72
Union, 149, 150
UnionHasPtv, 151, 152, 157
UnionVisitorZ, 150
whatHolder, 33–36
Wood, 29
Zero, 7, 79
Zucchini, 37,
This is for the loyal Schemers and MLers.

```java
interface TI { 
  o→oI apply(TI x); 
}

interface O→oI { 
  Object apply(Object x); 
}

interface oo→ooI { 
  o→oI apply(oo→ooI x); 
}

interface oo→oo→ooI { 
  o→oI apply(oo→oo→ooI x); 
}

class Y implements oo→oo→ooI { 
  public o→oI apply(oo→oo→ooI f) { 
    return new H(f).apply(new H(f)); } 
}

class H implements T→I { 
  oo→ooI f; 
  H(oo→ooI f) { 
    f = f; } 
  public o→oI apply(T→I x) { 
    return f.apply(new G(x)); } 
}

class G implements o→oI { 
  T→I x; 
  G(T→I x) { 
    x = x; } 
  public Object apply(Object y) { 
    return (x.apply(x)).apply(y); } 
}
```

No, we wouldn't forget factorial.

```java
class MkFact implements oo→ooI { 
  public o→oI apply(o→oI fact) { 
    return new Fact(fact); } 
}

class Fact implements o→oI { 
  o→oI fact; 
  Fact(o→oI fact) { 
    fact = fact; } 
  public Object apply(Object i) { 
    int inti = ((Integer)i).intValue(); 
    if (inti == 0) 
      return new Integer(1); 
    else 
      return new Integer( 
        inti 
        * 
        ((Integer) 
          fact.apply(new Integer(inti - 1))) 
        .intValue()); } 
}
```

Copyrighted Material