CSE 331
Software Design & Implementation

Kevin Zatloukal

Fall 2017

Design Patterns, Part 2

(Based on slides by Mike Ernst, Dan Grossman, David Notkin, Hal Perkins, Zach Tatlock)

Reminder

- Course evaluations are available:
  - https://uw.iasystem.org/survey/183496

- Section tomorrow will review for the final

Review: Factories

Goal: want more flexible abstractions for what class to instantiate

- instantiation is ubiquitous in Java...
  yet Java constructors have many limitations

Factory method

- call a method to create the object
- method can do computation, return subtype, reuse objects

Factory object (also Builder)

- Factory has factory methods for some type(s)
- Builder has methods to describe object and then create it

Prototype

- every object is a factory, can create more objects like itself
- call **`clone`** to get a new object of same subtype as receiver

Review: Factory Method

Factory method: call a method to create the object

- can return any subtype or an existing object
- can give it a better name

```
new Matrix(double[] vals) { ... }
new Matrix(double[] vals, int rowSize) { ... }
```

versus `Matrix.fromX`

```
Matrix fromVector(double[] vals)
Matrix fromRowMajorEntries(double[] vals, int rowSize)
Matrix fromColMajorEntries(double[] vals, int colSize)
```

- Has two methods with same signature — impossible w/ constructors
- This approach can be used for *any* Java class.

Review: Builder

Builder: object with methods to describe object and then create it

- fits especially well with immutable classes when clients want to add data one bit at a time

Example 1: **`StringBuilder`**

```
StringBuilder buf = new StringBuilder();
buf.append("Total distance: ");
buf.append(dist);
buf.append(" meters");
return buf.toString();
```

Example 2: **`Graph.Builder`**

- `addNode`, `addEdge`, and `createGraph` methods
- (static inner class `Builder` can use **private** constructors)

Sharing

Second weakness of constructors: they always return a *new object*

Singleton: only one object exists at runtime

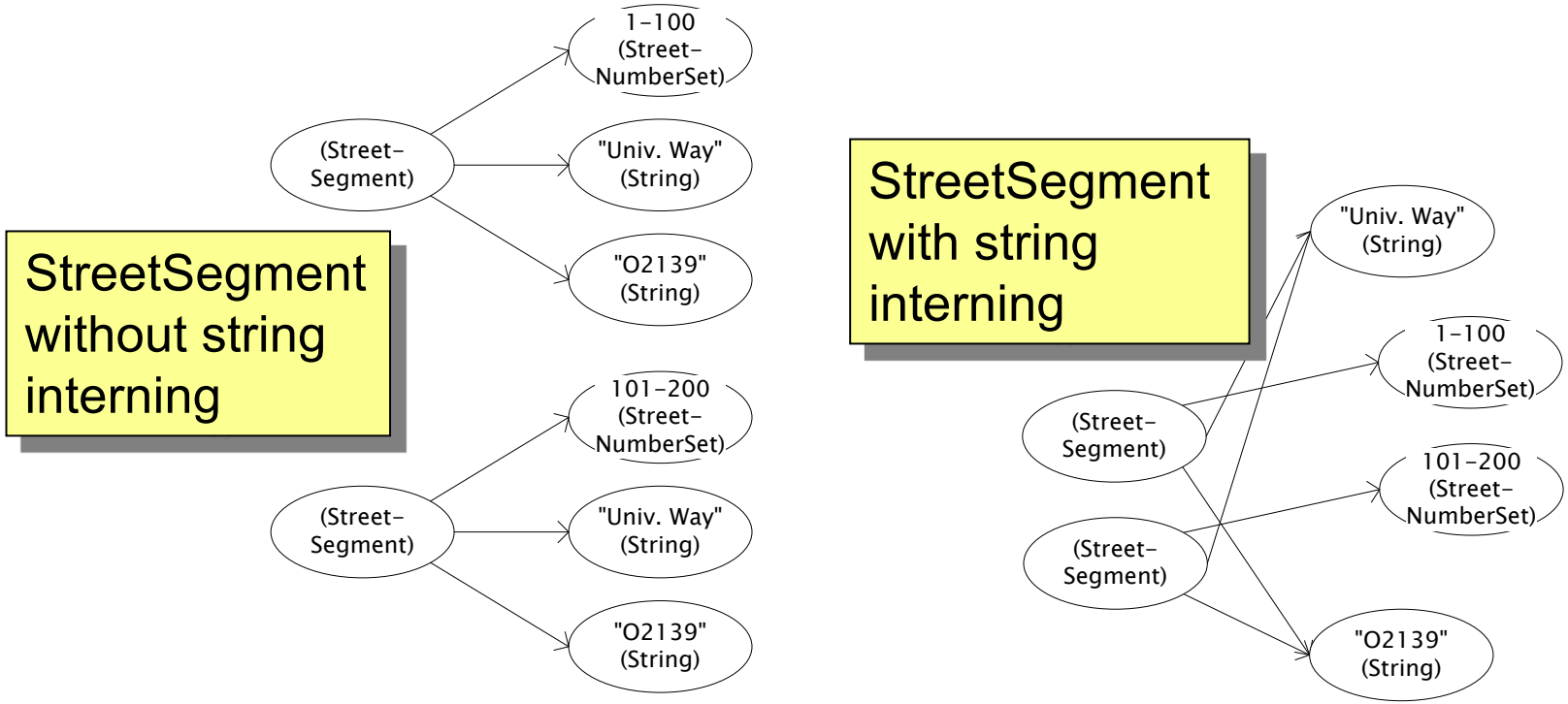
- factory method returns the same object every time
- (we've seen this already)

Interning: only one object with a particular value exists at runtime

- (with a particular *abstract* value)
- factory method can return an existing object (not a new one)
- interning can be used without factory methods
  - see `String.intern`

Interning pattern

Reuse existing objects instead of creating new ones:



Interning mechanism

- Maintain a collection of all objects in use
- If an object already appears, return that instead
  - (be careful in multi-threaded contexts)

```
HashMap<String, String> segNames;
String canonicalName(String n) {
  if (segNames.containsKey(n)) {
    return segNames.get(n);
  } else {
    segNames.put(n, n);
    return n;
  }
}
```

Why not `Set<String>` ?

`Set` supports `contains` but not `get`

- Java builds this in for strings: `String.intern()`

Interning pattern

- Benefits of interning:

  1. May compare with `==` instead of `equals()`
     - eliminates a source of common bugs!!
       - (my brain still freaks out when it sees == between objects)

  2. May save space by creating fewer objects
     - (space is less and less likely to be a problem nowadays)
     - also, interning can actually waste space if objects are not cleaned up when *no longer needed*
       - there are additional techniques to fix that ("weak references")

- Sensible only for immutable objects

`java.lang.Boolean`
does not use the Interning pattern

```
public class Boolean {
  private final boolean value;

  // construct a new Boolean value
  public Boolean(boolean value) {
    this.value = value;
  }

  public static Boolean FALSE = new Boolean(false);
  public static Boolean TRUE = new Boolean(true);

  // factory method that uses interning
  public static Boolean valueOf(boolean value) {
    if (value) {
      return TRUE;
    } else {
      return FALSE;
    }
  }
}
```

Recognition of the problem

Javadoc for **`Boolean`** constructor:

> Allocates a **`Boolean`** object representing the value argument.
>
> **Note:** It is rarely appropriate to use this constructor. Unless a new instance is required, the static factory `valueOf(boolean)` is generally a better choice. It is likely to yield significantly better space and time performance.

Josh Bloch (JavaWorld, January 4, 2004):

> The **`Boolean`** type should not have had public constructors. There's really no great advantage to allow multiple **`true`**s or multiple **`false`**s, and I've seen programs that produce millions of **`true`**s and millions of **`false`**s, creating needless work for the garbage collector.
>
> So, in the case of immutables, I think factory methods are great.

GoF patterns: three categories

Creational Patterns are about the object-creation process

Factory Method, Abstract Factory, Singleton, Builder, Prototype, …

Structural Patterns are about how objects/classes can be combined

Adapter, Bridge, Composite, Decorator, Façade, Proxy, …

Behavioral Patterns are about communication among objects

Command, Interpreter, Iterator, Mediator, Observer, State, Strategy, Chain of Responsibility, Visitor, Template Method, …

Green = ones we've seen already

Structural patterns: Wrappers

Wrappers are a thin veneer over an encapsulated class

- modify the interface
- extend behavior
- restrict access

The encapsulated class does most of the work

Pattern	Functionality	Interface
Adapter	**same**	**different**
Decorator	**different**	**same**
Proxy	**same**	**same**

Some wrappers have qualities of more than one of adapter, decorator, and proxy

Adapter

Real life example: adapter to go from US to UK power plugs

- both do the same thing
- but they have slightly interface expectations

Change an interface without changing functionality

- rename a method
- convert units
- implement a method in terms of another

Example: angles passed in radians vs. degrees

Example: use “old” method names for legacy code

Adapter example: rectangles

Our code is using this **`Rectangle`** interface:

```
interface Rectangle {
  // grow or shrink this by the given factor
  void scale(float factor);
  // move to the left or right
  void translate(float x, float y);
}
```

But we want to use a library that has this class:

```
class JRectangle {
  void scaleWidth(float factor) { ... }
  void scaleHeight(float factor) { ... }
  void shift(float x, float y) { ... }
}
```

Adapter example: rectangles

Create an adapter that delegates to `Rectangle`:

```
class RectangleAdapter implements Rectangle {
  JRectangle rect;
  RectangleAdapter(JRectangle rect) {
    this.rect = rect;
  }
  void scale(float factor) {
    rect.scaleWidth(factor);
    rect.scaleHeight(factor);
  }

  void translate(float x, float y) {
    rect.shift(x, y);
  }
  ...
}
```

Adapters

- This sort of thing happens **a lot**
  - unless two libraries were designed to work together, they won't work together without an adapter
- The example code uses **delegation**:
  - special case of composition where the outer object just forwards calls on to one other object
- Adapters can also **remove** methods
- Adapters can also be written by subclassing
  - but then all the usual warnings about subclassing apply ***if*** you override any methods of the superclass
  - your subclass could easily break when superclass changes

Decorator

- Add functionality without breaking the interface:
  1. Add to existing methods to do something extra
     - satisfying a stronger specification
  2. Provide extra methods

- Subclasses are often decorators
  - but not always: Java subtypes are not always true subtypes

Decorator example: Bordered windows

```
interface Window {
  // rectangle bounding the window
  Rectangle bounds();
  // draw this on the specified screen
  void draw(Screen s);
  ...
}

class WindowImpl implements Window {
  ...
}
```

Bordered window implementations

Via subclasssing:

```
class BorderedWindow1 extends WindowImpl {
  void draw(Screen s) {
    super.draw(s);
    bounds().draw(s);
  }
}
```

Delegation permits multiple borders on a window, or a window that is both bordered and shaded

Via delegation:

```
class BorderedWindow2 implements Window {
  Window innerWindow;
  BorderedWindow2(Window innerWindow) {
    this.innerWindow = innerWindow;
  }
  void draw(Screen s) {
    innerWindow.draw(s);
    innerWindow.bounds().draw(s);
  }
}
```

A decorator can remove functionality

Remove functionality without changing the Java interface

- no longer a true subtype, but *sometimes* that is necessary

Example: **`UnmodifiableList`**

- What does it do about methods like **`add`** and **`put`**?
  - throws an exception
  - moves error checking from the compiler to runtime
    - like Java array subtypes are another example of this

  Problem: **`UnmodifiableList`** is not a true subtype of **`List`**

  Decoration via delegation can create a class with no Java subtyping relationship, which is often desirable
  - Java subtypes that are not true subtypes are **confusing**
  - maybe necessary for **`UnmodifiableList`** though

Proxy

- Same interface *and* functionality as the wrapped class
  - so... uh... wait, what?

- Control access to other objects
  - communication: manage network details when using a remote object
  - locking: serialize access by multiple clients
  - security: permit access only if proper credentials
  - creation: object might not yet exist (creation is expensive)
    - hide latency when creating object
    - avoid work if object is never used

Composite pattern

- Composite permits a client to manipulate either an *atomic* unit or a *collection* of units in the same way
  - no need to "always know" if an object is a collection of smaller objects or not
- Good for dealing with "part-whole" relationships
- Used by jQuery in JavaScript
- An extended example…

Composite example: Bicycle

- Bicycle
  - Wheel
    - Skewer
      - Lever
      - Body
      - Cam
      - Rod
    - Hub
    - Spokes
    - Nipples
    - Rim
    - Tape
    - Tube
    - Tire
  - Frame
  - Drivetrain
  - ...

Methods on components

```
abstract class BicycleComponent {
  int weight();
  float cost();
}
class Skewer extends BicycleComponent {
  float price;
  float cost() { return price; }
}
class Wheel extends BicycleComponent {
  float assemblyCost;
  Skewer skewer;
  Hub hub;
  ...
  float cost() {
    return assemblyCost + skewer.cost()
           + hub.cost() + ...;
  }
}
```

Composite example: Libraries

Library
- Section (for a given genre)
  - Shelf
    - Volume
      - Page
        - Column
          - Word
            - Letter

```
interface Text {
  String getText();
}
class Page implements Text {
  String getText() {
    ... return concatenation of column texts ...
  }
}
```

Composite example: jQuery (*off topic*)

- jQuery provides a function **`$`** that returns one or many objects
  - **`$("#foo")`** would return the object with ID "foo"
    - (or returns an empty collection if none exists)
  - **`$("p")`** wound return a collection of all P nodes

- Calling a method on a jQuery object calls that method on all objects in the collection:
  - if **`foo`** is a node with id "foo", then **`foo.hide()`** has the same effect as **`$("foo").hide()`**
  - **`$("p").hide()`** would hide *all* the P nodes

GoF patterns: three categories

Creational Patterns are about the object-creation process

Factory Method, Abstract Factory, Singleton, Builder, Prototype, …

Structural Patterns are about how objects/classes can be combined

Adapter, Bridge, Composite, Decorator, Façade, Proxy, …

Behavioral Patterns are about communication among objects

Command, Interpreter, Iterator, Mediator, Observer, State, Strategy, Chain of Responsibility, Visitor, Template Method, …

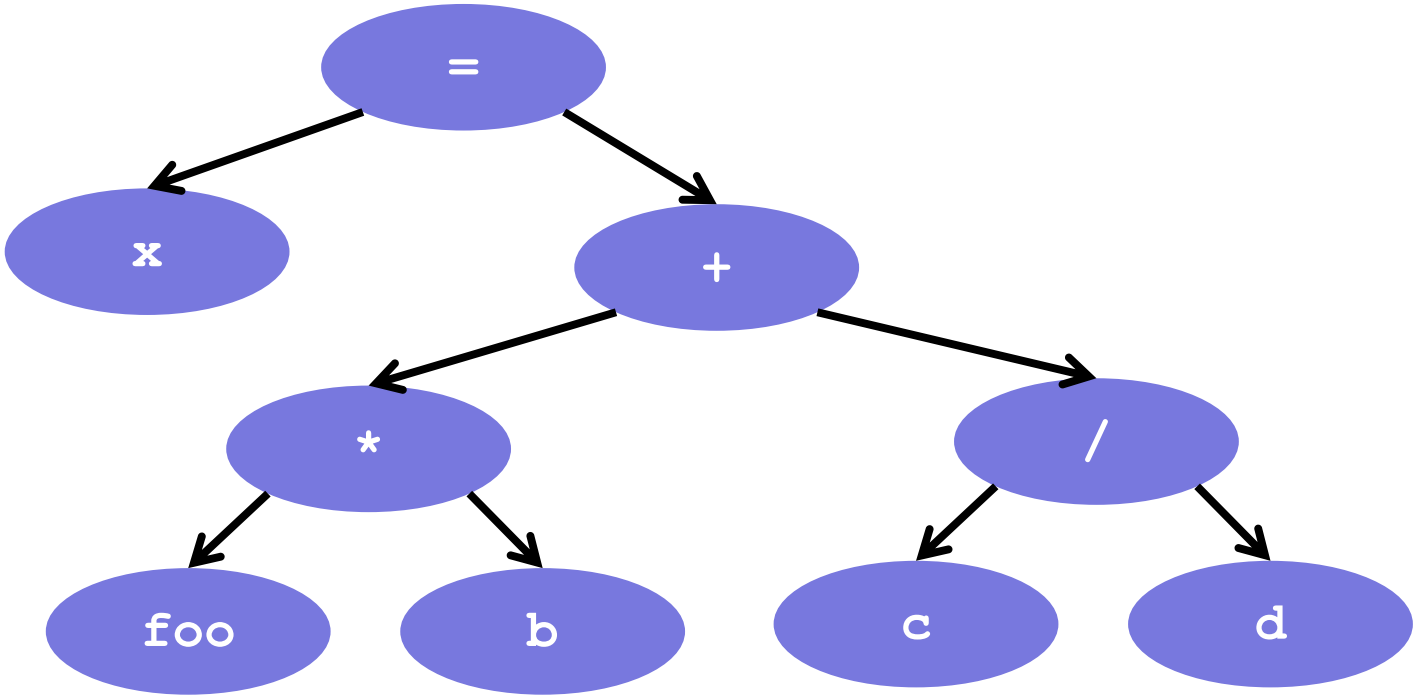
Green = ones we've seen already

Traversing composites

- Goal: perform operations on all parts of a composite
- Idea is to generalize the notion of an iterator: process the components in an order appropriate for the application
- This is really important when writing a compilers
  - (doesn't come up nearly as much elsewhere though)
- Example of patterns to work around limitations of OOP
- Example: arithmetic expressions in Java
  - how do we represent, say, `x = foo*b + c/d;`
  - how do we traverse/process these expressions?

Representing Java code

```
x = foo * b + c / d;
```

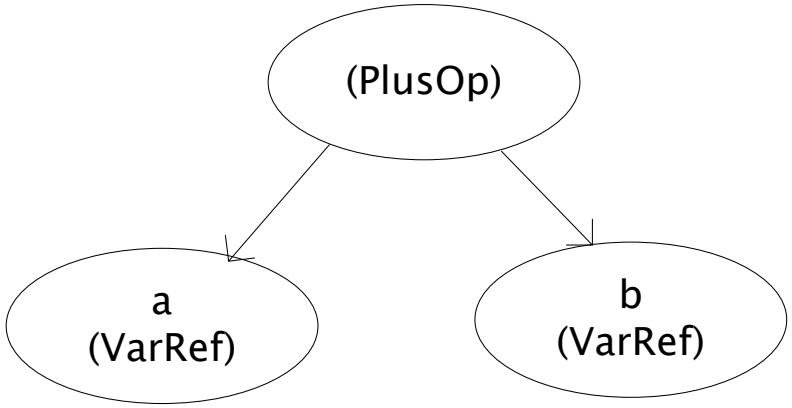


Abstract syntax tree (AST) for Java code

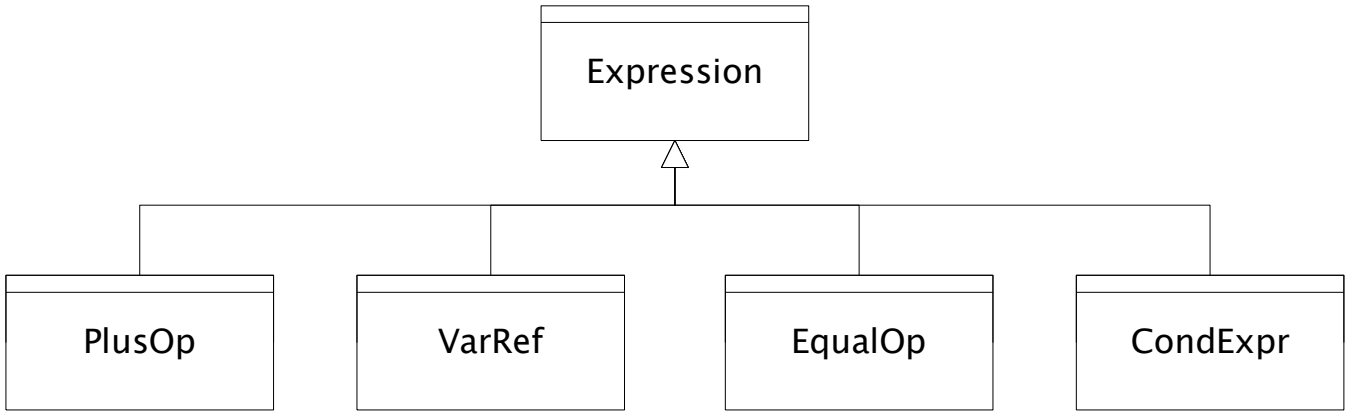
```
class PlusOp extends Expression { // + operation
  Expression leftExp;
  Expression rightExp;
}
class VarRef extends Expression { // variable use
  String varname;
}
class EqualOp extends Expression {  // test a==b;
  Expression leftExp; // left-hand side: a in a==b
  Expression rightExp; // right-hand side: b in a==b
}
class CondExpr extends Expression {  // a?b:c
  Expression testExp;
  Expression thenExp;
  Expression elseExp;
}
```

Object model vs. type hierarchy

- AST for **`a + b`**:



- Class hierarchy for **`Expression`**:



Operations on abstract syntax trees

Need to write code for each entry in this table

Operations \ Types of Objects	CondExpr	EqualOp
typecheck		
print		

- Question: Should we group together the code for a particular operation or the code for a particular expression?
  - That is, do we group the code into rows or columns?
- Given an operation and an expression, how do we "find" the proper piece of code?

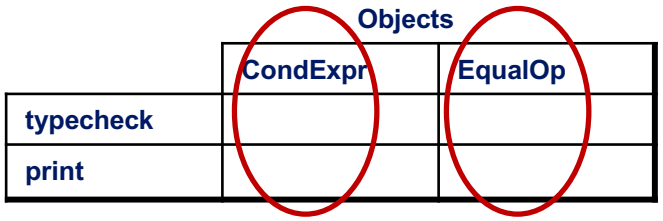
Interpreter and procedural patterns

Interpreter: collects code for similar objects, spreads apart code for similar operations
- easy to add new types
- hard to add operations
- Composite pattern

Procedural: collects code for similar operations, spreads apart code for similar objects
- easy to add operations
- hard to add new types
- Visitor pattern

(See CSE341 for an extended take on this question:
- statically typed functional languages help with procedural whereas statically typed OO languages help with interpreter)

Interpreter pattern



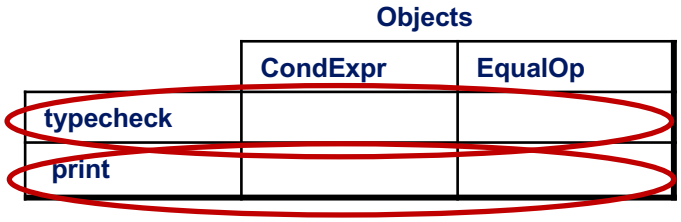
Add a method to each class for each supported operation

```
abstract class Expression {
  ...
  Type typecheck();
  String print();
}
class EqualOp extends Expression {
  ...
  Type typecheck() { ... }
  String print() { ... }
}
class CondExpr extends Expression {
  ...
  Type typecheck() { ... }
  String print() { ... }
}
```

Dynamic dispatch chooses the right implementation, for a call like `e.typeCheck()`

Overall type-checker spread across classes

Procedural pattern



Create a class per operation, with a method per operand type

```
class Typecheck {
  Type typeCheckCondExpr(CondExpr e) {
    Type condType = typeCheckExpr(e.condition);
    Type thenType = typeCheckExpr(e.thenExpr);
    Type elseType = typeCheckExpr(e.elseExpr);
    if (condType.equals(BoolType) &&
            thenType.equals(elseType)))
      return thenType;
    else
      return ErrorType;
  }
  Type typeCheckEqualOp(EqualOp e) {
    ...
  }
}
```

How to invoke the right method for an expression `e`?

Definition of `typeCheckExpr` (using procedural pattern)

```
class Typecheck {
  ...
  Type typeCheckExpr(Expression e) {
    if (e instanceof PlusOp) {
      return typeCheckPlusOp((PlusOp)e);
    } else if (e instanceof VarRef) {
      return typeCheckVarRef((VarRef)e);
    } else if (e instanceof EqualOp) {
      return typeCheckEqualOp((EqualOp)e);
    } els
      ret
    } els
    ...
  }
}
```

Maintaining this code is tedious and error-prone

- No help from type-checker to get all the cases (unlike in functional languages)

Cascaded if tests are likely to run slowly (in Java)

Need similar code for each operation

Visitor pattern: A variant of the procedural pattern

- Nodes (objects in the hierarchy) accept visitors for traversal
- Visitors visit nodes (objects)

```
class SomeExpression extends Expression {
  void accept(Visitor v) {
    for each child of this node {
      child.accept(v);
    }
    v.visit(this);
  }
}
class SomeVisitor extends Visitor {
  void visit(SomeExpression n) {
    perform work on n
  }
}
```

`n.accept(v)` traverses the structure rooted at `n`, performing `v`'s operation on each element of the structure

Example: accepting visitors

```
class VarOp extends Expression {
  …
  void accept(Visitor v) {
    v.visit(this);
  }
class EqualsOp extends Expression {
  …
  void accept(Visitor v) {
    leftExp.accept(v);
    rightExp.accept(v);
    v.visit(this);
  }
}
class CondOp extends Expression {
  …
  void accept(Visitor v) {
    testExp.accept(v);
    thenExp.accept(v);
    elseExp.accept(v);
    v.visit(this);
  }
}
```

First visit all children

Then pass "self" back to visitor

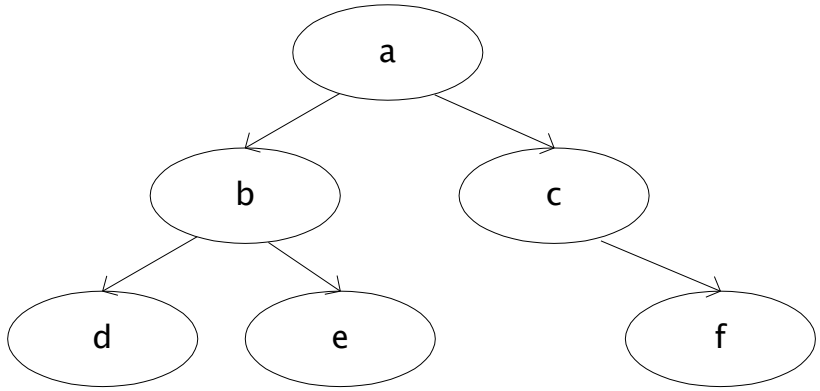
The visitor has a **`visit`** method for each kind of expression, thus picking the right code for this kind of expression

- Overloading makes this look more magical than it is…

Lets clients provide unexpected visitors

Sequence of calls to accept and visit

```
a.accept(v)
    b.accept(v)
        d.accept(v)
            v.visit(d)
        e.accept(v)
            v.visit(e)
        v.visit(b)
    c.accept(v)
        f.accept(v)
            v.visit(f)
        v.visit(c)
    v.visit(a)
```



Sequence of calls to visit: d, e, b, f, c, a

Example: Implementing visitors

```
class TypeCheckVisitor
   implements Visitor {
  void visit(VarOp e) { … }
  void visit(EqualsOp e) { …
  void visit(CondOp e) { … }
}

class PrintVisitor implement
   Visitor {
  void visit(VarOp e) { … }
  void visit(EqualsOp e) { …
  void visit(CondOp e) { … }
}
```

Now each operation has its cases back together

And type-checker should tell us if we fail to implement an abstract method in Visitor

Again: overloading just a nicety

Again: An OOP workaround for procedural pattern

- Because language/type-checker is not instance-of-test friendly