Breaking things down

- In functional (and procedural) programming, break programs down into functions that perform some operation
- In object-oriented programming, break programs down into classes that give behavior to some kind of data

This lecture:
- These two forms of decomposition are so exactly opposite that they are two ways of looking at the same “matrix”
- Which form is “better” is somewhat personal taste, but also depends on how you expect to change/extend software
- For some operations over two (multiple) arguments, functions and pattern-matching are straightforward, but with OOP we can do it with double dispatch (multiple dispatch)

The expression example

Well-known and compelling example of a common pattern:
- Expressions for a small language
- Different variants of expressions: ints, additions, negations, …
- Different operations to perform: eval, toString, hasZero, …

Leads to a matrix (2D-grid) of variants and operations
- Implementation will involve deciding what “should happen” for each entry in the grid regardless of the PL

Standard approach in ML

- Define a datatype, with one constructor for each variant
  - (No need to indicate datatypes if dynamically typed)
- Define a function for each operation
- So “fill out the grid” via one function per column with one case-expression branch for each grid position
  - Can use a wildcard pattern if there is a default for multiple entries in a column

See lec23_stage1.sml

Standard approach in OOP

- Define a class, with one abstract method for each operation
  - (No need to indicate abstract methods if dynamically typed)
- Define a subclass for each variant
- So “fill out the grid” via one class per row with one method implementation for each grid position
  - Can use a method in the superclass if there is a default for multiple entries in a column

See lec23_stage1.rb and lec23_stage1.java

A big CSE341 punchline

- FP and OOP often doing the same thing in exact opposite way
  - Organize the program “by rows” or “by columns”
- Which is “most natural” may depend on what you are doing (e.g., an interpreter vs. a GUI) or personal taste
- Code layout is important, but there’s no perfect way since software has many dimensions of structure
  - Tools, IDEs can help with multiple “views” (e.g., rows / columns)
Now for stage 2: FP

<table>
<thead>
<tr>
<th>eval</th>
<th>toString</th>
<th>hasZero</th>
<th>noNegConstants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Negate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mult</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- For implementing our grid so far, SML / Racket style usually by column and Ruby / Java style usually by row
- But beyond just style, this decision affects what (unexpected?) software extensions are easy and/or do not change old code
- Functions:
  - Easy to add a new operation, e.g., noNegConstants
  - Adding a new variant, e.g., Mult requires modifying old functions, but ML type-checker gives a to-do list if we avoided wildcard patterns in Stage 1

Fall 2011  CSE341: Programming Languages

Now for stage 2: OOP

<table>
<thead>
<tr>
<th>eval</th>
<th>toString</th>
<th>hasZero</th>
<th>noNegConstants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Negate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mult</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- For implementing our grid so far, SML / Racket style usually by column and Ruby / Java style usually by row
- But beyond just style, this decision affects what (unexpected?) software extensions are easy and/or do not change old code
- Objects:
  - Easy to add a new variant, e.g., Mult
  - Adding a new operation, e.g., noNegConstants requires modifying old classes, but Java type-checker gives a to-do list if we avoided default methods in Stage 1

Thoughts on Extensibility

- Making software extensible is valuable and hard
  - If you know you want new operations, use FP
  - If you know you want new variants, use OOP
  - If both? Languages like Scala try; it’s a hard problem
  - Reality: The future is often hard to predict!

- Extensibility is a double-edged sword
  - Code more reusable without being changed later
  - But makes original code more difficult to reason about locally or change later (could break extensions)
  - Often language mechanisms to make code less extensible (ML modules hide datatypes; Java’s final prevents subclassing/overriding)

Stage 3: Binary operations

- Situation is more complicated if an operation is defined over multiple arguments that can have different variants
  - Can arise in original program or after an extension

- Our example:
  - Include variants String and Rational
  - (Re)define Add to work on any pair of Int, String, Rational in either order
    - String-concatenation if >= 1 arg is a String, else math
    - (Just to keep example smaller, Negate and Mult still work only on Int, with a run-time error for a String or Rational)

Binary operation in SML

Add works differently for most combinations of Int, String, Rational
- Run-time error for any other kinds of expression
Natural approach: pattern-match on the pair of values
- For commutative possibilities, can re-call with (v2,v1)

```sml
fun add_values (v1,v2) = 
  case (v1,v2) of
  | (Int i, Int j) => Int (i+j)
  | (Int i, String s) => String (Int.toString i ^ s)
  | (Int i, Rational(j,k)) => Rational (i*k+j,k)
  | (Rational _, Int _) => add_values (v2,v1)
  | ... (* 5 more cases (3^2 total): see lec23.sml *)

fun eval e =
  case e of
  | Add(v1,v2) => add_values (eval v1, eval v2)```

Fall 2011  CSE341: Programming Languages
**Binary operation in OOP: first try**

- Normal dynamic dispatch gives us separate methods for the variant of the first argument (the receiver)
  - We could then abandon OOP style and use Racket-style type tests for branching on the 2nd argument's variant
  - 9 cases total: 3 in Int's `add_values`, 3 in String's `add_values`, 3 in Rational's `add_values`

```
class Int
  def add_values other
    if other.is_a? Int
      ...
    elsif other.is_a? Rational
      ...
    else ...
  end
end
```

**A more OO style**

- The FP approach had 3*3 case-expression branches
- Our half-OOP approach had 3 methods with 3 branches
- A full-OOP would have 9 methods, with dynamic dispatch picking the right one
  - There are languages that have such multimethods, i.e., method calls that use dynamic dispatch on > 1 argument
  - Ruby & Java (& C++ & C# & …) have no such feature
  - But we can code it up ourselves in an OOP way using the `double-dispatch idiom` (next slide)
    - (If we had three arguments, could use triple dispatch, etc., but double-dispatch is already fairly unwieldy)

**The double-dispatch “trick”**

- If Int, String, and Rational all define all of `addInt`, `addString`, and `addRational`, that's 9 cases
  - For example, String’s `addInt` is for additions of the form "i + s" where i is an int and s is a string (i.e., self is "on the right")

- Add's `eval` method calls `el1.eval.add_values el2.eval`, which dispatches to `add_values` in Int, String, or Rational
  - Int's `add_values`: other.addInt self
  - String's `add_values`: other.addString self
  - Rational `add_values`: other.addRational self
  - So `add_values` performs "the 2nd dispatch" to the correct case!

See lec23.rb

**Works in Java too**

- In a statically typed language, double-dispatch works fine
  - Just need all the dispatch methods in the type

```
abstract class Value extends Exp {
  abstract Value add_values(Value other);
  abstract Value addInt(Int other);
  abstract Value addString(String other);
  abstract Value addRational(Rational other);
}
class Int extends Value { ... }
class String extends Value { ... }
class Rational extends Value { ... }
```

See lec23.java

**Summary**

- “The 2-D grid” is a fundamental truth about software, essential to understanding how OOP and procedural decomposition relate
- Software extensibility is easy in some ways and hard in others
  - Which ways are which depend on how code is structured
- Double-dispatch is how you “stay OOP” in a language without multimethods for operations that take multiple arguments of different variants
  - Is “staying OOP” here worth it?