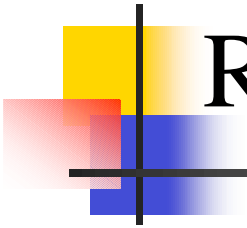
The Strategy and Visitor Patterns

Corky Cartwright
Department of Computer Science
Rice University



Review: First-class Functions in Java

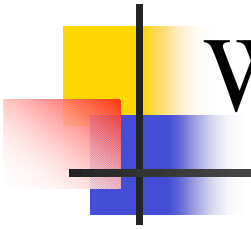
- Methods are not data values in Java, but objects containing methods are data values. Hence, we can pass functions using degenerate objects with a single method as the primary member.
- When we pass such a value to an object and perhaps store that object in a field of the receiving object (as in passing an argument to a constructor), we are using what OO designer call the *strategy* pattern. The passed object is intuitively viewed as a strategy (behavior) to be used by the receiving object. The type of the method in the "function object" indicates what form of strategy is being passed.
- Example: passing a layout manager to a GUI component.

Another Example: Building Sort Objects

- Recall the `IntList` class from Lecture ??. Assume that we want to create sort functions that sort . s in different ways (using different algorithms and orderings). How can create such objects and how can we apply them to `IntList`s? By:
- Defining a `Sorter` interface

```
interface Sorter {
  IntList sort(IntList host);
}
```

- Defining a hook in `IntList` for applying `Sorter` objects to `this`.
- Defining strategy objects that implement `Sorter`.



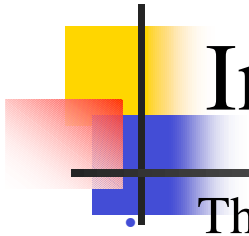
What Is Ugly About Class `UpSort`?

- It defines sorting code statically using a decision tree of predicates, just like a functional program. Ugh …
- How can we do better? Need hooks in our `IntList` class that let us write essentially the interpreter pattern code for a method on `IntList` as a first-class data object. This is really a first-class function written according to the interpreter patter so it has extra structure, namely a clause (method) for each different kind of concrete class in the composite.

Deconstructing the Interpreter Pattern

- In the interpreter pattern, the method is declared as abstract in the root class/interface and defined concretely in each concrete variant (subclass). To package the code for a method defined by the interpreter pattern in a separate object (called a *visitor*) we need:

```
class ... {
  Object forEmptyIntList(EmptyIntList host) {
    ... <method code using host instead of this> ...
  }
  Object forConsIntList(ConsIntList host) {
    ... <method code using host instead of this> ...
  }
}
```



Invoking Visitors

- The corresponding composite class must include hooks to invoke visitors for the class. To specify the signature of these hooks, we need to introduce an interface for all visitors that return `IntList`

```
interface IntListVisitor {
  IntList forEmptyIntList(EmptyIntList host)
  IntList forConsIntList(ConsIntList host)
}
```

- The hook methods have trivial definitions:

```
abstract class IntList {
  ...
  abstract Object accept(IntListVisitor v);
}
class EmptyIntList extends IntList {
  ...
  IntList accept(IntListVisitor v) { return forEmptyIntList(this); }
}
class ConsIntList extends IntList {
  ...
  IntList accept(IntListVisitor v) { return forConsIntList(this); }
}
```

Defining Visitors

- Easy case: method of no arguments.
- Example: upsort

```
class UpSortVisitor {
  IntList forEmptyIntList(EmptyIntList host) { return host; }
  IntList forConsIntList(ConsIntList host) {
    return host.rest().accept(this).insert(host.first());
  }
}
```

- Oops! We have to write `insert` as a visitor. But it has an argument!

```
class InsertVisitor {
  int elt;
  IntList forEmptyIntList(EmptyIntList host) { return host.cons(elt); }
  IntList forConsIntList(ConsIntList host) {
    int first = host.first();
    IntList rest = host.rest();
    if (elt <= first) return host.cons(elt);
    return rest.accept(this).cons(first);
  }
}
```

- Last line of `UpSortVisitor` becomes

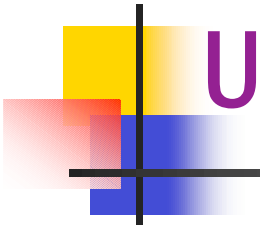
```
return host.rest().accept(this).accept(new InsertVistor(host.first()));
```

Defining Visitors cont.

- Revised upsort

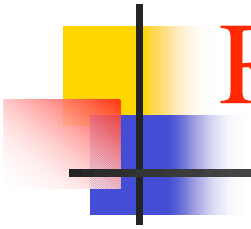
```
class UpSortVisitor {
  IntList forEmptyIntList(EmptyIntList host) { return host; }
  IntList forConsIntList(ConsIntList host) {
    host.rest().accept(this).
      accept(new InsertVisitor(host.first());
  }
}
```

- Remember: the problem decomposition is not affected by using visitor pattern; only the syntax!
- Why use visitors? There is a compelling reason in addition to "elegance". It the same reason why the interpreter pattern is far superior to static method definitions. **Inheritance**.



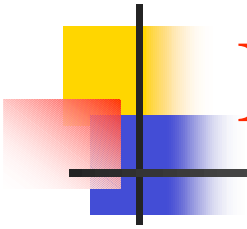
`UpSort` Example

Go to DrJava



Reprise: Anonymous Classes

- What do free variables mean inside anonymous classes. What do they mean in λ-expressions?
- In Java, the free variables can be either:
  - fields, or
  - local (method) variables.
- Use them in doing the `filter` problem in HW8.



For Next Class

- Labs today and tomorrow. Covering first-class functions and visitors.
- Get comfortable with visitors; you will use them extensively in the next assignment.
- Please report problems with DrJava Language Levels.