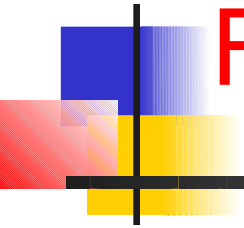
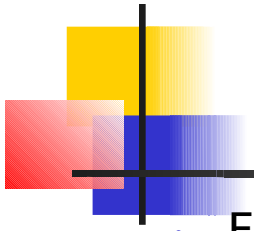


First-class Functions and Patterns

Corky Cartwright
Department of Computer Science
Rice University

Encoding First-class Functions in Java

- Java methods are *not* data values; *they cannot be used as values*.
- But java classes include methods so we can implicitly pass methods (functions) by passing class instances containing the desired method code.
- Moreover, Java includes a mechanism that closes over the free variables in the method definition!
- Hence, first-class functions (closures) are available implicitly, but the syntax is wordy.
- Example: Scheme **`map`**

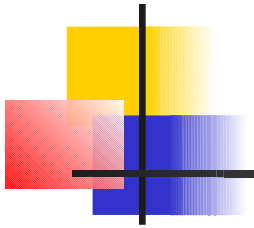


Interfaces for Representing Functions

- For accurate typing, we need different interfaces for different arities. With generics, we can define parameterized interfaces for each arity. For now, we will have to define separate interfaces for each desired typing.
- `map` example:

```
interface Lambda {
  Object apply(Object arg);  // Object -> Object
}

abstract class ObjectList {
  ObjectList cons(Object n) {
    return new ConsObjectList(n, this);
  }
  abstract ObjectList map(Lambda f);
}
   ...
```



Representing Specific Functions

- For each function that we want to use a value, we must define a class, preferably a singleton. Since the class has no fields, all instances are effectively identical.
- Defining a class seems unduly *heavyweight*, but it works in principle.
- In OO parlance, an instance of such a class is called a strategy.
- Java provides a lightweight notation for singleton classes called anonymous classes. Moreover these classes can refer to fields and `final` method variables that are in scope. In DrJava language levels, all variables are `final`. `final` fields and variables cannot be rebound to new values after they are initially defined (immutable). `final` methods cannot be overridden.
- Anonymous class notation:

```
new <type>() {
  <member1>
  ...
  <membern>
}
```

Anonymous Class Example

```
new Lambda() {
  Object apply(Object arg) {
    return EmptyObjectList.ONLY.cons(arg);
  }
}
```

There are pending proposals to provide better notation for lambda abstractions. For now, you must pay attention to the interface signature defined in the library/program you are using.

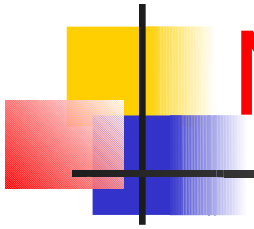
This interface (**`Lambda`**) together with its implentation classes is called the strategy pattern.

Another Example: Building Sort Objects

- Recall the **`IntList`** class from Lecture 20. Assume that we want to create sort functions that sort **`IntList`**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`**.
- Note: we introduced an interface specific to this problem (instead of using **`Lambda`** to suppot a more precise typing. (With generics (classes and methods parameerized by type) there is no advantage to creating a special interface.)

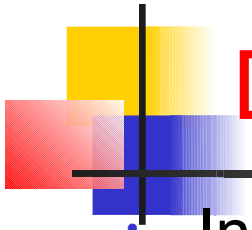


Naive Coding of `UpSorter`

```
class UpSorter implements Sorter {
  private UpSorter() { }
  IntList sort(IntList host) {
    if (host.equals(EmptyIntList.ONLY)) return host;
    ConsIntList cHost = (ConsIntList) host;
    return insert(sort(cHost.rest()), cHost.first());
  }
  IntList insert(IntList host, int elt) {
    if (host.equals(EmptyIntList.ONLY))
      return EmptyIntList.ONLY.cons(elt);
    ConsIntList cHost = (ConsIntList) host;
    if (elt <= cHost.first()) return cHost.cons(elt);
    return insert(cHost.rest(), elt).cons(cHost.first());
  }
}
```

What Is Ugly About Class `UpSorter`?

- 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 pattern so it has extra structure, namely a clause (method) for each variant (concrete class) in the composite. Think of it as a closure (data object representing a function) with explicit interpreter pattern structure.



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 to write each concrete method definition:

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

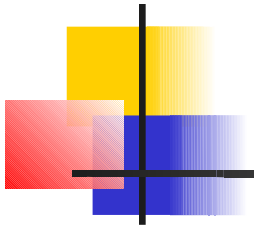
Invoking Visitors

- 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); }
}
```



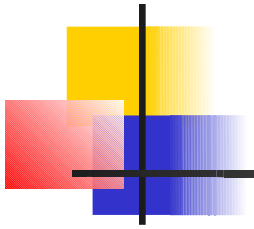
Defining Visitors

- Easy case: method of no arguments.
- Example: upsort (we drop the "up" qualifier since ascending order is assumed)

```
class UpSortVisitor implements IntListVisitor {
  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! We embed the argument in the visitor object (which is exactly what happens when we apply a Scheme closure):

```
class InsertVisitor implements IntListVisitor {
  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);
  }
}
```



Defining Visitors cont.

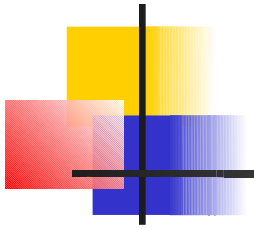
Last line of `UpSortVisitor` becomes

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

Yielding:

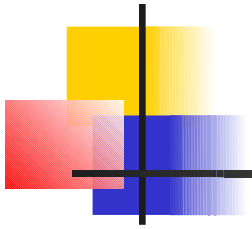
```
class UpSortVisitor {
  IntList forEmptyIntList(EmptyIntList host) { return host; }
  IntList forConsIntList(ConsIntList host) {
    /* visitor equivalent of rest().sort().insert(first) */
    return host.rest().accept(this).accept(new InsertVisitor(host.first()));
  }
}
```

The preceding is standard OO visitor code. You need to learn to understand it as a transliteration of cleaner interpreter code which is a transliteration of still cleaner functional code.



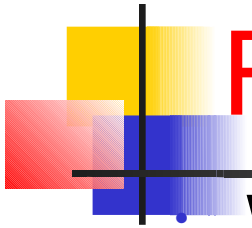
Defining Visitors cont.

- 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**.
- But `SortVisitor` is a closure class akin to `UpSorter`. So why bother with the `Sorter` interface at all? We can use `IntListVisitor` instead.
- What is the primary software engineering disadvantage of visitors (and OO in general)? Dynamic dispatch makes tracing code during debugging difficult.



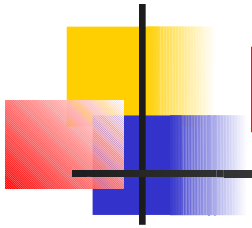
`UpSort` Example

- Go to DrJava
- See code files saved with this lecture in the course wiki.



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
  - `final` 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 any problems with DrJava Language Levels. I have not had any problems in testing/revising class solutions.