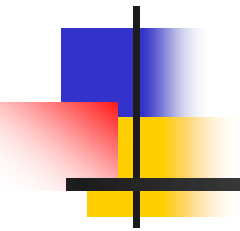


Strategy Pattern: Encoding Functions As Data Values

Corky Cartwright
Department of Computer Science
Rice University

Supporting First-class Functions in Java

- Java methods are *not* data values; *they cannot be used as values.* (Java 8 introduces method references which are abbreviations for Java lambda-abstractions, also introduced in Java 8. These extensions of Java are syntactic sugar for the inner class encodings we are going to discuss; they may be useful. I don't have enough experience with the notation to have an definitive opinion.)
- Since java classes include methods, we can effectively pass methods (functions) by passing an appropriate class singleton implementing an interface type that is designed exclusively to represent Java functions.
- We need to design an interface corresponding to a function of type

  $A_1\ A_2\ \ldots.\ A_n \rightarrow R$
- Stay tuned …

Interfaces for Function Types

Given the function type $A_1\ A_2\ \ldots\ A_n \rightarrow R$,

what is a corresponding interface the Java encoding?

```
interface ILambdaN<A1,A2,...,An, R> {
  R apply(A1 x1, A2 x2, ..., An xn);
}
```

Note: if you do not need multiple instantiations of **ILambdaN** in your program, you can drop the generic type parameters as appropriate.

Let's add the method ***map*** (for unary functions) to out generic functional List class using the interpreter pattern. (Why not visitor pattern?)

- In `List<E>`

```
abstract <R> List<R> map(ILambda<E,R> f);
```

- In `Empty<E>`

```
<R> List<R> map(ILambda<E,R> f) { return new EmptyList<E>(); }
```

- In `Cons<E>`

```
<R> List<R> map(ILambda<E,R> f) {
   return rest.map(f).cons(f.apply(first));
}
```

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.
- 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 the DrJava Function Language, all variables are **`final`**. **`final`** fields and method variables cannot be rebound to a new value after they are initially defined, *i.e.*, they are immutable.
- Anonymous class notation:

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

Anonymous Class Example

The following anonymous class defines the common “drop” operation in the context of a composite class hierarchy for the generic root class `List<T>`

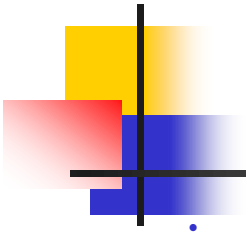
```
new ILambda1<T, List<T>>() {
  apply(T arg) {
    return new EmptyList<T>().cons(arg);
  }
}
```

Beware that `List<T>` refers to our generic functional list class, not `java.util.List<T>`.

Java 8 introduced more concise notation for lambda abstractions:

```
T arg -> new EmptyList<T>().cons(arg);
```

which I almost never use because I internalized the explicit representation using anonymous classes decades ago. Moreover, the newer notation has a serious flaw; the keyword `this` means the enclosing class instance rather than the anonymous class, preventing simple recursion. The notation for anonymous classes gets it right!



Example

- An anonymous class denoting the factorial function:

```
new ILambda1<Integer, Integer>() {
  apply(Integer n) {
    if (n <= 1) return 1;
    return n * apply(n - 1);
  }
}
```

- Note that the call `apply(n - 1)` implicitly uses `this` which refers to the anonymous class instance NOT the enclosing instance. Unfortunately, there is no equivalent expression using the new "lambda" notation. Regrettably, the Oracle committee controlling the evolution of Java believes

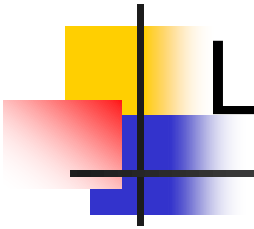
  Consistency is the hobgoblin of little minds ...

- Erasure-based generics are so brittle that I often write code that is correct for first-class generics (where type parameters have the same status as conventional types) and subsequently address any problems (violations of erasure-based restrictions) reported by the compiler.

Loose Ends: Exceptions

- In Java, error values are called exceptions. Exceptions are conventional objects and hence are created by expressions of the form **`new <exception-class>(<arg`$_1$`>, ..., <arg`$_n$`>)`**.
- The Java libraries include on the order of 100 different exception classes signifying different forms of error. The all inherit from the class **`Exception`**. Moreover, the most useful and convenient form of exception is a subclass of **`Exception`** called **`RuntimeException`**. All of the exceptions that we will use will belong to type (subclasses of) **`RuntimeException`** except some choices already dictated in the libraries. The only defensible use of "checked" exceptions is for explicit exception-based control (which I dislike because it smells bad in OO designs).
- Some of the important exception classes are:

  `NullPointerException`
  `ClassCastException`
  `IllegalArgumentException`
  `java.util.NoSuchArgumentException`



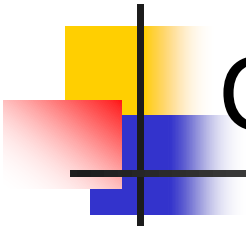
Loose Ends: Exceptions cont.

- To explicitly raise an exception in Java code, you simply **throw** it using the syntax
  throw <except-expr>
  where **<except-expr>** is a an expression (typically a **new** expression) denoting an exception.
- Examples:

```
throw new IllegalArgumentException("max applied to an empty list")
throw new java.util.NoSuchElementException("max applied to an empty list")
```

Loose Ends: Casts

- The Java static type system uses simple rules to infer types for Java expressions.
- The inferred type for an expression is conservative; it is guaranteed to be correct, but it may be weaker than what is required for a particular computation. As a result, Java supports type coercions called casts of the form
  `(<type>) <expr>`
  that simply convert the type of `<expr>` to `<type>` for type-checking purposes. If the value of `<expr>` does not have type `<type>`, the computation throws a `ClassCastException`. The type information from a cast is purely local, it does not affect the inferred type of subsequent occurrences of `<expr>`. As a result, Java code must repeatedly cast such expressions to narrower type *or* introduce a new variable of the narrower type bound to the value of `<expr>`.
- **Example**: recall the insertion sort example we studied in Racket. We can easily do the same in Java.



Casting: A Final Comment

- The Java compiler disallows casts

  (<type>) <expr>

  where **<type>** is an object (*reference*) type and the static type of **<expr>** and **<type>** do not overlap (ignoring **null**).

Example **InsertSort** (part 1)

```
abstract class List<T extends Comparable<T>> {
  EmptyList<T> empty() { return new EmptyList<T>(); }
  ConsList<T> cons(T n) { return new ConsList<T>(n, this); }
  abstract List<T> insert(T t);
  abstract List<T> insertSort();
  /* [inherited] public boolean equals(Object other) must be overridden */
  /** A help function to support overriding toString() to produce a Lisp-like
    representation for List<T>  */
  abstract String toStringHelp();
}
final class EmptyList<T extends Comparable<T>> extends List<T> {
  EmptyList() { }
  public boolean equals(Object other) { return other instanceof EmptyList; }
  List<T> insert(T t) { return this.cons(t); }
  List<T> insertSort() { return this; }
  public String toString() { return "()"; }  // must be public to match
    declaration in Object
  String toStringHelp() { return ")"; }
}
```

Example **InsertSort** (part 2)

```
final class ConsList<T extends Comparable<T>> extends List<T> {
  T first;
  List<T> rest;
  T first() { return first; }
  List<T> rest() { return rest; }
  ConsList(T f, List<T> r) { first = f; rest = r; }
  public boolean equals(Object other) { // must be public to match declaration
    in Object
    if (! (other instanceof ConsList)) return false;
    ConsList o = (ConsList) other;
    return first.equals(o.first()) && rest.equals(o.rest());
  }
  List<T> insert(T t) {
    if (t.compareTo(first) <= 0) return this.cons(t);
    return rest.insert(t).cons(first);
  }
  List<T> insertSort() { return rest.insertSort().insert(first); }
  public String toString() { return "(" + first + rest.toStringHelp(); }
  String toStringHelp() { return " " + first + rest.toStringHelp(); }
}
```