



# Loose Ends and First-class Functions

---

Corky Cartwright

Department of Computer Science

Rice University




# Loose Ends: DrJava LL Bug

---

- In some cases, it does not erase old `.java` and `.class` files when compiling. Solution: delete these files using (in Linux or Mac OS X):

```
/bin/rm *.java *.class
```

in the directory where your files are stored.



# 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>(<arg1>, ..., <argn>)`.
- The Java libraries include on the order of 100 different exception classes signifying different forms of error. They 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.
- 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 converts 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 **merge** method on **ComparableList** for today's homework (HW7) written using the conventional Scheme solution.



# merge Example

---

```
abstract class ComparableList {
    ComparableList cons(Comparable n) { return new ConsComparableList(n, this); }
    abstract ComparableList merge(ComparableList other);
}

class EmptyComparableList extends ComparableList {
    static EmptyComparableList ONLY = new EmptyComparableList();
    private EmptyComparableList() { }
    ComparableList merge(ComparableList other) { return other;}
}

class ConsComparableList extends ComparableList {
    Comparable first;
    ComparableList rest;
    ComparableList insert(ComparableList other) {
        if (other == EmptyComparableList.ONLY) return this;
        ConsComparableList o = ((ConsComparableList) other;
        if (first < o.first()) return rest.merge(o).cons(first);
        return merge(o.rest()).cons(o.first());
    }
}
```



# 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`).



# 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 pass methods (functions) by passing an appropriate class implementing an interface type that is designed exclusively to represent Java functions.
- 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 UnaryFun {
    Object apply(Object arg); // Object -> Object
}

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



# Representing Specific Funcions

---

- 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 DrJava language levels, all variables are **final**. **final** fields and variables cannot be rebound to a new value after they are initially defined (immutable). **final** methods cannot be overridden.
- Anonymous class notation:

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



# Anonymous Class Example

---

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

There are pending proposals to provide better notation for lambda abstractions.



## For Next Class

---

- Spring Break next week.
- New Homework due Friday, March 13. Note that the homework incidentally treats `char` as a primitive type. `char` constants are enclosed in single quote marks (the single quote is on the same key as the usual double quotation mark, between the colon/semicolon key and `return`). Some examples are: `'a'` `'b'` `'c'` `'1'`
- Reading: OO Design Notes, Ch 1.9.