Thinking Functionally

Dan S. Wallach and Mack Joyner, Rice University
Reminder: Fill out our web form!

Fill this out ASAP if you haven’t already.
http://goo.gl/forms/ArYKWBc0zY

We need this to connect you to Subversion and Piazza.
You need that for project 1. Due Sunday night!

You must be registered for Comp215 by Monday morning
Otherwise, you’ll be removed from Subversion and Piazza.
**FP is a methodology that feels like basic arithmetic**

In no math class have you ever *mutated* a variable

Example: if you wrote \( x = f(x) \) in a math class...

- You’d be saying “\( x \) is a fixed point of \( f \)”
- Not “overwrite \( x \) with \( f(x) \)”
- If you really needed to, you’d invent new variables, e.g.:
  - \( x_{n+1} = f(x_n) \)

**FP: Define things once, use them many times**
Why Functional Programming?

FP is easier to think about before you start writing your code.

FP is easier to test and debug.
Same inputs yield same outputs every time.

FP is easier to parallelize.
You’re describing what you want, not how you want to get it.
FP abstractions are much easier to run concurrently.
Giant “cloud” cluster computations use FP abstractions.
Simple example: Lists

// // Mutating lists //
public class MList {
    public void add(Object o) { ... }
    public boolean contains(Object o) { ... }
    public Object getHead() { ... }
    public boolean empty() { ... }
}

MList ml = new MList();
ml.add("Hello");
ml.add("Rice");
ml.add("Owls");

System.out.println(ml.getHead()); // Owls
System.out.println(ml.getHead()); // Rice
System.out.println(ml.getHead()); // Hello

// // Functional lists //
public class FList {
    public FList add(Object o) { ... }
    public boolean contains(Object o) { ... }
    public Object head() { ... }
    public FList tail() { ... }
    public boolean empty() { ... }
}

FList fl = new FList()
    .add("Hello")
    .add("Rice")
    .add("Owls");

System.out.println(fl.head()); // Owls
System.out.println(fl.tail().head()); // Rice
System.out.println(fl.tail().tail().head()); // Hello
Mutating APIs

Standard practice before Java 8:
java.util “Collections” APIs like this.

API calls change the “value” of the object every time.

You can’t easily hang onto “old copies.”
$O(n)$ copy cost.

Things get ugly:

In parallel (Comp322)
Proving correctness (Comp411 / 511)

```java
// Mutating lists
public class MList {
    public void add(Object o) { ... }
    public boolean contains(Object o) { ... }
    public Object getHead() { ... }
    public boolean empty() { ... }
}

MList ml = new MList();
ml.add("Hello");
ml.add("Rice");
ml.add("Owls");

System.out.println(ml.getHead()); // Owls
System.out.println(ml.getHead()); // Rice
System.out.println(ml.getHead()); // Hello
```
Functional APIs

API calls don’t change the “value” of the object every time.
Every function returns a new object, leaves the old alone.
Making “copies” is O(1).
You can “chain” calls together.

```java
// Functional lists
public class FList {
    public FList add(Object o) { ... }
    public boolean contains(Object o) { ... }
    public Object head() { ... }
    public FList tail() { ... }
    public boolean empty() { ... }
}

FList fl = new FList()
    .add("Hello")
    .add("Rice")
    .add("Owls");

System.out.println(fl.head()); // Owls
System.out.println(fl.tail().head()); // Rice
System.out.println(fl.tail().tail().head()); // Hello
```
No mutation!
API design challenge: how do we represent an empty list?
Sir Tony Hoare on *null* references:

I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare
What’s wrong with **null**?

Java (and many other languages) allows you to pass a **null** anywhere you would pass a reference to an object.

Super convenient when you want to represent a pointer to nothing

- *Error conditions*: how should I return “nothing”?
- *Uninitiated fields/members*: how should I represent “unitialized”?

You can’t actually call a method on a **null** reference

`NullPointerException` at runtime, forces **null** checks everywhere

In Comp215, you will never return **null**

You’ll notice `@NotNull` annotations throughout the code we give you

- You don’t have to use these in your own code
- IntelliJ will yell at you if you try to pass a **null** to one of these functions
Q: “What is a list?”

Data definition: A list is either:
   1) a value and another list
   2) an empty-list

Functional programming “rules”:
Only assign to members in the constructor.
(the final keyword enforces this)
Different classes for each case.
(shared interface for both)
Recursion follows the data definition.
(smaller problem, each step)
Write tests that express what you expect.
(Again, following the data definition)

/**
 * Interface for functional lists of objects.
 */
public interface ObjectList {
    boolean empty();
    Object head();
    ObjectList tail();
    ObjectList add(Object o);
    int length();
    ObjectList concat(ObjectList other);
    boolean contains(Object o);
}
Functional List Implementation

class Empty implements ObjectList {
  private static final ObjectList SINGLETON = new Empty();
  private Empty() { }

  public boolean empty() {
    return true;
  }

  public Object head() {
    throw new ListException("can't take head of an empty list");
  }

  public ObjectList tail() {
    return this;
  }

  public ObjectList add(Object o) {
    return new ObjectList(o, this);
  }

  public int length() {
    return 0;
  }
}

class Cons implements ObjectList {
  private final Object headVal;
  private final ObjectList tailVal;

  private Cons(Object headVal, ObjectList tailVal) {
    this.headVal = headVal;
    this.tailVal = tailVal;
  }

  public boolean empty() {
    return false;
  }

  public Object head() {
    return headVal;
  }

  public ObjectList tail() {
    return tailVal;
  }

  public ObjectList add(Object o) {
    return new ObjectList(o, this);
  }

  public int length() {
    return 1 + tailVal.length();
  }
}
class Empty implements ObjectList {
    private static final ObjectList SINGLETON = new Empty();

    private Empty() { }

    public boolean empty() {
        return true;
    }

    public Object head() {
        throw new ListException("can't take head of an empty list");
    }

    public ObjectList tail() {
        return this;
    }

    public ObjectList add(Object o) {
        return new ObjectList(o, this);
    }

    public int length() {
        return 0;
    }
}

Private constructors: we force the programmer to start with an empty-list, then build it with add(). No null!
Functional List Implementation

```
class Cons implements ObjectList {
    private final Object headVal;
    private final ObjectList tailVal;
    
    private Cons(Object headVal, ObjectList tailVal) {
        this.headVal = headVal;
        this.tailVal = tailVal;
    }
    
    public boolean empty() {return false;}
    
    public Object head() {return headVal;}
    
    public ObjectList tail() {return tailVal;}
    
    public ObjectList add(Object o) {
        return new ObjectList(o, this);
    }
    
    public int length() {
        return 1 + tailVal.length();
    }
}

class Empty implements ObjectList {
    private static final ObjectList SINGLETON = new Empty();
    private Empty() {
    }
    
    public boolean empty() {
        return true;
    }
    
    public Object head() {
        throw new ListException("can't take head of an empty list");
    }
    
    public ObjectList tail() {
        return this;
    }
    
    public ObjectList add(Object o) {
        return new ObjectList(o, this);
    }
    
    public int length() {
        return 0;
    }
}

Singleton: only ever one instance of the empty-list.
```
Null-avoidance strategy: singletons

Two different classes: `ObjectList.Cons` and `ObjectList.Empty`
Follows our data definition!
Yes, you can define a class inside an interface!

Only one instance of `ObjectList.Empty`
Hidden (`private`) constructor
Static method on `ObjectList` to “make” an empty-list → same one every time

External users only see `ObjectList`, not `.Cons` or `.Empty`
Instead, you start with an `ObjectList.makeEmpty()` and then `add()` things
Interface vs. implementation: we’re hiding the implementation classes (abstraction!)
Remember: these are “functional”, so old lists never change
  • You get a “new” list every time
Two different classes: `ObjectList.Cons` and `ObjectList.Empty`
Follows our data definition!
Yes, you can define a class inside an interface!

Only one instance of `ObjectList.Empty`
Hidden (private) constructor
Static method on `ObjectList` to “make” an empty-list
→ same one every time

External users only see `ObjectList`
Instead, you start with `ObjectList.makeEmpty()` and then `add()` things

Interface vs. implementation: we’re hiding the implementation classes (abstraction!)
Remember: these are “functional”, so old lists never change

Vocabulary alert: When hiding constructors and exposing public static methods instead, we call them factory methods. This style is sometimes called a factory pattern.
class Cons implements ObjectList {
    private final Object headVal;
    private final ObjectList tailVal;

    public int length() {
        return 1 + tailVal.length();
    }
}

class Empty implements ObjectList {
    public int length() {
        return 0;
    }
}
class Cons implements ObjectList {
    private final Object headVal;
    private final ObjectList tailVal;

    public boolean contains(Object o) {
        if (headVal.equals(o)) {
            return true;
        } else {
            return tailVal.contains(o);
        }
    }
}

class Empty implements ObjectList {
    public boolean contains(Object o) {
        return false;
    }
}
Functional List: list to string

class Cons implements ObjectList {

    public String toString() {
        if (tailVal.empty()) {
            return headVal.toString();
        } else {
            return headVal.toString() + " " + tailVal.toString();
        }
    }
}

class Empty implements ObjectList {

    public String toString() {
        return "";
    }
}
class Cons implements ObjectList {
    public ObjectList add(Object o) {
        return new Cons(o, this);
    }
}

class Empty implements ObjectList {
    public ObjectList add(Object o) {
        return new Cons(o, this);
    }
}
Functional List: add an element to a list

class Cons implements ObjectList {
    public ObjectList add(Object o) {
        return new Cons(o, this);
    }
}

class Empty implements ObjectList {
    public ObjectList add(Object o) {
        return new Cons(o, this);
    }
}

Same exact code for both cases!
“Promoting” code to be a default method

```java
interface ObjectList {
    default ObjectList add(Object o) {
        return new Cons(o, this);
    }
}

class Cons implements ObjectList {
    public ObjectList add(Object o) {
        return new Cons(o, this);
    }
}

class Empty implements ObjectList {
    public ObjectList add(Object o) {
        return new Cons(o, this);
    }
}
```

Default add( ) method is now used by both Cons and Empty

We can delete the add( ) method bodies from Cons and Empty. *Don’t repeat yourself!*
The “standard” Comp215 representation

When you’ve got a data structure with multiple cases:
Lists: empty-list vs. cons
Trees: leaf vs. interior

It’s handy to have everything in the same file
Java lets you define inner classes inside interfaces
  • And vice versa!

```java
interface ObjectList {
  class Cons implements ObjectList {
    . . .
  }
}

class Empty implements ObjectList {
  . . .
}
```
Features of the “standard” representation

Public static “factory” methods of the (outer) interface
Private constructors of the (inner) classes
  • Force external users to call the factory methods instead
  • Hiding the constructors allows for reuse (e.g., singletons)

Default methods of the (outer) interface
Methods are “copied” into each (inner) class
Default methods can only call other interface methods
  • Makes it easier to implement the classes
But what if you really want mutation?

Mutation is a useful way to maintain state
Example: you’ve got a game, and the state of the world changes.

Strategy: use a functional list on the inside and delegate to it

```java
public class MList {
    private ObjectList list = ObjectList.makeEmpty();

    public void add(Object o) {
        list = list.add(o);
    }

    public Object getHead() {
        Object headVal = list.head();
        list = list.tail();
        return headVal;
    }
}
```

```java
MList ml = new MList();
assertTrue(ml.empty());
ml.add("Hello");
ml.add("Rice");
ml.add("Owls");
assertFalse(ml.empty());
assertTrue(ml.contains("Rice"));
assertFalse(ml.contains("Harvard"));
assertEquals("Owls", ml.getHead());
assertEquals("Rice", ml.getHead());
assertEquals("Hello", ml.getHead());
```
Testing!
Unit testing: `assertTrue`, `assertEquals`, etc.

We’re using JUnit4, natively supported by IntelliJ
You write assertions, JUnit verifies them.

We’ll provide unit tests for your assignments.
Your grade comes (mostly) from passing the tests.

```java
package edu.rice.week1lists;
import org.junit.Test;
import static org.junit.Assert.*;
public class MListTest {

    @Test
    public void testBasics() throws Exception {
        MList ml = new MList();
        assertTrue(ml.empty());
        ml.add("Hello");
        ml.add("Rice");
        ml.add("Owls");
        assertFalse(ml.empty());
        assertTrue(ml.contains("Rice"));
        assertFalse(ml.contains("Harvard"));
        assertEquals("Owls", ml.getHead());
        assertEquals("Rice", ml.getHead());
        assertEquals("Hello", ml.getHead());
    }
}
```
IntelliJ can help you with the boilerplate
Unit testing is an industry-standard practice

They’re really part of the definition of your code
Unit tests lay out how you expect things to work. “Extreme” programmers write their tests before writing their code.

They’re a “bacon-saving device”
Whenever you make a change, re-run your tests. Whenever you fix a bug, create a test to verify it. (Never again!)

Why does industry love unit tests?
It’s cheaper to find bugs before shipping a product. Tests help you find bugs. Shipping buggy products sucks. Ergo...
Unit testing + functional programming

Functional code is *much* easier to test than mutating code
Same input $\rightarrow$ same output, every time

Modern engineering practice:
Functional “core”
Mutating “wrapper”

*Lots more about this as the semester goes on.*
Live coding: let’s write some unit tests

Guided tour of IntelliJ features
Code directory structure (src/main/java vs. src/test/java)
Creating / running unit tests
Gradle & CheckStyle
Version control: Committing your code

ObjectList: let’s write tests for toString(), etc.