Changes since lecture

Changed `list.count()` to `list.length()`
The slides were based on an older version of my list code

We decided on a new midterm date: Monday October 19
Room TBD, but hopefully somewhere bigger than BRK 101 so we have space
Midterm 1 schedule?

Option 1: Wednesday, October 14 (right after break, no review?)
Option 2: Friday, October 16 (Grace Murray Hopper Conference?)
Option 3: Monday, October 19 (swap lectures around?)
Functional programming?

FP is a methodology that feels like basic arithmetic

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

(Example: If you write $x = f(x)$, that’s a statement of truth, not an assignment to a variable.)

FP is easier to think through before you start writing code

FP is easier to debug

FP is easier to parallelize
Simple example: Lists

```java
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
```

```java
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

```java
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
```

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

You can’t easily hang on to “older” versions without making copies

Things get ugly when:
- programming in parallel (Comp322)
- getting extra fancy with the type system (Comp311/411/511)
- “proving” things about your program
Functional APIs

API calls don’t change the “value” of the object

When there’s a new “value”, you get a new object back

This lets you chain / pipeline together different methods

```java
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!
Q: “What is a list?”

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

The “functional rules”:
(a) Only assign to members in the constructor
(The “final” keyword enforces this.)
(b) Subclasses for the different cases
(Override methods as necessary.)
(c) Recursion follows the data definition
(Smaller problem, every step.)
(d) Write tests that express what you expect
(Again, follow the data definition.)

```java
public class FList {
    final Object value;
    final FList tailList;

    FList(Object value, FList tailList) {
        this.value = value;
        this.tailList = tailList;
    }

    public FList add(Object value) {
        return new FList(value, this);
    }
}

public class FListEmpty extends FList {
    ...
}
```
public class FList {
    final Object value;
    final FList tailList;

    ... 

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

public class FListEmpty extends FList {
    ...

    public int length() {
        return 0;
    }
}
public class FList {
    final Object value;
    final FList tailList;

    ...

    public boolean contains(Object o) {
        if (value.equals(o)) return true;
        return tailList.contains(o);
    }
}

public class FListEmpty extends FList {
    ...

    public boolean contains(Object o) {
        return false;
    }
}
public class FList {
    final Object value;
    final FList tailList;

    public String toString() {
        return value.toString() + " " +
            tailList.toString();
    }
}

public class FListEmpty extends FList {

    public String toString() {
        return "";
    }
}
public class FList {
    final Object value;
    final FList tailList;

    public String toString() {
        if (tailList.empty())
            return value.toString();
        else
            return value.toString() + " " + tailList.toString();
    }
}

public class FListEmpty extends FList {
    public String toString() {
        return "";
    }
}
null is special. It’s the default value of every object pointer.

null sucks. You can’t call any methods on it without a crash.

```
if (this.value.equals(o)) return true;
```

fails if `value` is `null`

null makes all your code more complicated, checking that it’s not there.

There are insanely sophisticated tools to catch these mistakes.

The latest: Facebook Infer (http://fbinfer.com/)

Singleton “empty list” types (FEmptyList, etc.) let us avoid needing null.
public class FList {
    final Object value;
    final FList tailList;

    // we don't want others using this
    protected FList(Object value, FList tailList) {
        this.value = value;
        this.tailList = tailList;
    }

    // returns a new list with value in the front
    public FList add(Object value) {
        // works even if 'this' is FEmptyList
        return new FList(value, this);
    }
}

public class FEmptyList extends FList {
    private final static FEmptyList singleton =
        new FEmptyList();

    // we don't want others using this
    protected FEmptyList() {
        super(null, null);
    }

    // call this to “create” an empty list, even
    // though it just returns one we already have
    public static FList create() {
        return singleton;
    }
}
Singletons and null avoidance

Don’t expose constructors directly.
Use static helper functions instead (e.g., to “create” an empty list).

Don’t return null if you can help it.
Coming in week 4, a functional way to deal with returning “something or nothing”.
Bonus Java feature: inner classes

No need for a separate .java file
Keep the two cases (empty-list, non-empty-list) nearby

“public static class”
Equivalent to putting it in a separate .java file.

This one is named “FList.Empty”

```java
public class FList {
  final Object value;
  final FList tailList;

  public static class Empty extends FList {
    private final static Empty singleton = new Empty();

    private Empty() {
      super(null, null);
    }

    // call this to “create” an empty list, even
    // though it just returns one we already have
    public static FList create() {
      return singleton;
    }
  }
}
```
But what if I really want mutation-style?

You can mix and match different style. Java doesn’t force functional behavior.

```java
public class MList {
    private FList list;
    public MList() { list = FList.Empty.create(); }
    public void add(Object o) { list = list.add(o); }
    public boolean contains(Object o) { return list.contains(o); }

    public Object getHead() {
        if(list.empty()) return null; // bad!
        Object result = list.head();
        list = list.tail();
        return result;
    }
}
```

The MList client has no idea there’s a FList lurking under the hood.
You should always write test cases. We’ll be using “JUnit4”. How it works:

(a) If you just created FList.java, create a new FListTest.java file
(b) Write simple tests that exercise your code

```java
import org.junit.Test;
import static org.junit.Assert.*;

public class FListTest {
    @Test
    public void testLength() throws Exception {
        FList emptyList = FList.Empty.create();
        FList list = emptyList.add("Alice").add("Bob").add("Charlie");

        assertEquals(emptyList.length(), 0);
        assertEquals(list.length(), 3);
    }
    
...
IntelliJ can help you get started

Produces boilerplate JUnit code. You still have to write the tests.
Unit tests are magic

They’re really part of the definition
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!)

Unit tests are an industry-standard practice
It’s cheaper to find bugs before shipping a product.
Tests help you find bugs.
Shipping buggy products sucks.
Ergo...
Live coding demo

Writing JUnit tests for FList