

# *IMMUTABILITY, FUNCTIONAL PROGRAMMING*

**Prof. Chris Jermaine**  
**cmj4@cs.rice.edu**

# A Few Words On References

- Repeatedly made the case: references are difficult to reason about
- Why? It's the same old story...
  - Method method1 creates an object called obj1
  - Puts a reference to obj1 into container object obj2 (aliasing!)
  - method1 calls method method2, passes (as a param) a reference to obj2
  - method2 gets the reference to obj1 via the obj2 parameter; updates obj1
  - method2 completes execution
  - Back in method1, obj1 has changed, though method1 never sent it as a param!
  - When you debug, it looks like magic...

# How To Deal With This?

- Classic solution:
  - Make all of your objects “immutable”
  - That is, unchangeable after initialization
  - So (in Java) all member variables are “final”
- Many basic Java types are immutable
  - Strings, integers, doubles
- Why does this help?
  - Aliasing can't be a problem if you can't update object state, right?
  - No one can ever change a value out from under you!

# But How Do You Write Programs?

- OK, having the little build-in types be immutable is fine
- But can you make more interesting types immutable?
  - Ex: how to insert into a container
  - Does that not change the state of a container?
- It's actually easy
  - Especially if you are not too concerned with performance
  - Just make every method a *function* (in the purest mathematical sense)
  - A function is equivalent to a map
  - Takes an input tuple (set of params)
  - Maps it to an output object
  - No alteration of input in a function... it's just a map!

# “Function Heads” Take This to the Extreme

- They argue no assignment after initialization
  - Ever!
  - Will come back to this shortly

# Example “Functional” Linked List

- Remember this?

```
interface ListWRemove <T extends Comparable <T>> {  
    // insert an item into the list  
    public void insert (T insertMe);  
  
    // remove a specific item  
    public T remove (T removeMe);  
  
    // print the list so the first item inserted is first  
    public void print ();  
}
```

# Example “Functional” Linked List

- Here’s the functional version

```
interface ListWRemove <T extends Comparable <T>> {  
    // insert an item into the list  
    public ListWRemove <T> insert (T insertMe);  
  
    // remove a specific item  
    public Tuple <T, ListWRemove <T>> remove (T removeMe);  
  
    // print the list so the first item inserted is first  
    public void print ();  
}
```

- Few notes

- Every method returns result, **does not change input params**
- We assume we have a “Tuple” generic that allows two things to be returned
- Note the “void” type on print... true function-heads **hate** I/O!

Term used lovingly!



## To Imp This, Need a Node

```
abstract class GenericNode <T extends Comparable <T>> {  
  
    // insert an item into the list, returns new list  
    public GenericNode <T> insert (T insertMe);  
  
    // remove a specific item, return the resulting list  
    public Tuple <T, GenericNode <T>> remove (T removeMe);  
  
    // print the list so the first item inserted is first  
    public void print ();  
}
```



# Concrete For End-Of-List Is Easy

```
class EndNode <.> extends GenericNode <.> {  
  
    public GenericNode <T> insert (T insertMe) {  
        return new NodeWithChild <T> (insertMe, this);  
    }  
  
    // remove a specific item, return the resulting list  
    public Tuple <T, GenericNode <T>> remove (T removeMe) {  
        return new Tuple <T, GenericNode <T>> (null, this);  
    }  
  
    // print the list so the first item inserted is first  
    public void print () {}  
}
```

# Node With a Child Is Not Too Bad

```
class NodeWithChild <.> extends GenericNode <.> {  
  
    private final GenericNode <T> child;  
    private final T myGuy;  
  
    public GenericNode insert (T insertMe) {  
        return new NodeWithChild <T> (insertMe, this);  
    }  
  
    public Tuple <T, GenericNode <T>> remove (T removeMe) {  
        if (removeMe.compareTo (myGuy) == 0) {  
            return new Tuple <T, GenericNode <T>> (myGuy, child);  
        } else {  
            Tuple <T, GenericNode <T>> res = child.remove (removeMe);  
            return new Tuple <.> (res.getFirst (),  
                new NodeWithChild <T> (myGuy, res.getSecond (())));  
        }  
    }  
  
    public NodeWithChild (T data, GenericNode <T> nextOne) {  
        myGuy = data;  
        child = nextOne;  
    }  
}
```

# What About Actual List?

```
class ChrisList <.> implements ListWRemove <.> {  
  
    private final GenericNode <T> listHead;  
  
    public ListWRemove <T> insert (T insertMe) {  
        return new ChrisList <T> (listHead.insert ());  
    }  
  
    public Tuple <T, ListWRemove> remove (T removeMe) {  
        Tuple <T, GenericNode <T>> res = listHead.remove ();  
        return new Tuple <T, ChrisList <T>> (res.getFirst (),  
            new ChrisList <T> (res.getSecond()));  
    }  
  
    private public ChrisList (GenericNode <T> useThisHead) {  
        listHead = useThisHead;  
    }  
  
    public ChrisList () {  
        listHead = new EndNode <T> ();  
    }  
}
```

# What Is Different From Before?

- At top level
  - All ops over nodes return the head of a new list
  - “ChrisList” always constructs a new list with this new head
- Removing an item
  - We don’t just cut out the item
  - Because just cutting it out would require changing the reference at the cut
  - So we effectively cut it out and then build a copy of the list before the cut

# Using This Immutable List Type

- Let's insert a bunch of numbers into it:

```
ChrisList <Integer> foo0 = new ChrisList <Integer> ();  
ChrisList <Integer> foo1 = foo0.Insert (1);  
ChrisList <Integer> foo2 = foo1.Insert (2);  
ChrisList <Integer> foo3 = foo2.Insert (3);
```

- Can now look at each of the 4 lists...
  - The “i”th list will contain the numbers from 1 through i
  - Insertion did not change any of the lists

# Can Take This Idea Even Further

- Say I want to insert 20 numbers into a ChrisList

```
ChrisList <Integer> foo = new ChrisList <Integer> ();  
for (int i = 0; i < 20; i++) {  
    foo = foo.insert (i + 1);  
}
```

- A true function head won't like this... why?

# Can Take This Idea Even Further

- You are assigning to “foo” and to “i” after initialization
- They’d argue your code should have looked like:

```
class RecursionShell {  
  
    public ChrisList <Integer> loadUp (int i) {  
        if (i == 0) {  
            return new ChrisList <Integer> ();  
        } else {  
            return loadUp (i - 1).insert (i);  
        }  
    }  
}  
...  
RecursionShell temp = new RecursionShell ();  
ChrisList <Integer> foo = new temp.loadUp (20);
```

## Another Example: “Functional” BST

- Here was the interface from lect 16 (with a new remove method)

```
interface ListWFastFind <T extends Comparable <T>> {  
  
    // insert an item into the list  
    public void insert (T insertMe);  
  
    // find a specific item  
    public boolean isThere (T findMe);  
  
    // remove a specific item  
    public T remove (T removeMe);  
}
```



# Another Example: “Functional” BST

- And here is the functional version

```
interface ListWFastFind <T extends Comparable <T>> {  
  
    // insert an item into the list  
    public ListWFastFind <T> insert (T insertMe);  
  
    // find a specific item  
    public boolean isThere (T findMe);  
  
    // remove a specific item  
    public Tuple <T, ListWFastFind <T>> remove (T removeMe);  
}
```

# Here Is The Node Code

```
protected abstract class BSTNode <.> {  
    public abstract boolean isThere (T findMe);  
    public abstract BSTNode <T> insert (T insertMe);  
}  
  
protected class EmptyNode <.> extends BSTNode <.> {  
  
    public boolean isThere (T findMe) {  
        return false;  
    }  
  
    public BSTNode <T> insert (T insertMe) {  
        return new EmptyNode <T> (insertMe);  
    }  
  
    public Tuple <T, BSTNode <T>> remove (T removeMe) {  
        return new Tuple <T, BSTNode <T>> (null, this);  
    }  
}
```

- Note: all of this should be parameterized with <T> if it is **outside of** ChrisBST
- But you can leave off the <T> if it is **inside of** ChrisBST

# Here Is The Node Code

```
protected class InternalNode <.> extends BSTNode <.> {
    ... // declare private member variables here

    public InternalNode (BSTNode <T> leftTreeIn, BSTNode <T> rightTreeIn, T dataIn) {...}

    public boolean isThere (T findMe) {
        if (myData.compareTo (findMe) == 0) {
            return true;
        } else if (myData.compareTo (findMe) > 0) {
            return leftSubtree.isThere (findMe);
        } else {
            return rightSubtree.isThere (findMe);
        }
    }

    public BSTNode <T> insert (T insertMe) {
        if (myData.compareTo (insertMe) >= 0) {
            return new InternalNode <T> (leftSubtree.insert (insertMe), rightSubtree, myData);
        } else {
            return new InternalNode <T> (leftSubtree, rightSubtree.insert (insertMe), myData);
        }
    }
}
```

# Here Is The Node Code

```
protected class InternalNode <.> extends BSTNode <.> {
    ... // declare private member variables here

    public Tuple <T, BSTNode <T>> remove (T removeMe) {
        if (myData.compareTo (removeMe) == 0) {
            Tuple <T, BSTNode <T>> temp = rightSubtree.remove (rightSubtree.findSmallest ());
            if (temp.getFirst () == null) {
                return new Tuple <T, BSTNode <T>> (myData, leftSubtree);
            } else {
                return new Tuple <T, BSTNode <T>> (myData,
                    new InternalNode <T> (leftSubtree, temp.getSecond (), temp.getFirst ()));
            }
        } else if (myData.compareTo (findMe) > 0) {
            Tuple <T, BSTNode <T>> temp = rightSubtree.remove (findMe);
            return new Tuple <T, BSTNode <T>> (temp.getFirst (),
                new InternalNode <T> (temp.getSecond (), rightSubtree, myData));
        } else {
            Tuple <T, BSTNode <T>> temp = leftSubtree.remove (findMe);
            return new Tuple <T, BSTNode <T>> (temp.getFirst (),
                new InternalNode <T> (leftSubtree, temp.getSecond (), myData));
        }
    }
}
```

— Will assume you can write “findSmallest”

# Putting It Together

```
public class ChrisBST <.> implements ... {  
  
    private final BSTNode <T> root;  
  
    private ChrisBST (BSTNode <T> useMe) {  
        root = useMe;  
    }  
  
    public ChrisBST () {  
        root = new EmptyNode <T> ();  
    }  
  
    public boolean isThere (T findMe) {  
        return root.isThere (findMe);  
    }  
  
    public ListWFastFind <T> insert (T insertMe) {  
        return new ListWFastFind <T> (root.insert (insertMe));  
    }  
  
    public Tuple <T, ListWFastFind <T>> remove (T removeMe) {  
        Tuple <T, BSTNode <T>> temp = root.remove (removeMe);  
        return new Tuple <T, ListWFastFind <T>> (temp.getFirst (),  
            new ListWFastFind <T> (temp.getSecond ()));  
    }  
}
```

# Some Final Topics Related to FP

- Deep copies and the “clone” method
- Lambdas
- Final thoughts re. Java and suitability for FP

# Deep Copies and Cloning

- We've seen that it is possible to write purely “functional” code
  - Even some non-trivial containers
  - But it required re-designing algorithms and re-writing a lot of code
- Say you want to employ some of these ideas in your programs
  - Even if you don't want to go all the way and be a “function head”
- Does this mean you have to re-write the standard library?

# Deep Copies and Cloning

- Is there an easier way?
- Might copying substitute?
  - In theory, sure. Say you want to call a method that modifies an object
  - But you want to be functional
  - The easiest way is to make a copy and **then** modify the copy
  - Might be inefficient, but you don't have to write new code
  - And you know you won't have bugs due to aliasing
  - Efficiency is often over-rated
  - And this is idiot-proof, right?



# Wrong!

- Beware... in the general case, there is no easy way to copy in Java
- So if you use copying as a path to FP, be aware...
  - You are going to need to write your own copy code
- You might reply: “Hey, doesn’t Object have a clone method?”
  - Yes it does
  - The convention is that “clone ()” first calls “super.clone ()”
  - Then it clones its internal structure
  - But it does *not* clone any objects it has a reference to
  - Why?!? I have no idea
  - So it can be dangerous to use

# So Sometimes It Does What You Want

```
public void testX() {  
  
    TreeMap <Integer, Integer> foo = new TreeMap <Integer, Integer> ();  
    for (int i = 0; i < 10; i++) {  
        foo.put (i, i);  
    }  
  
    TreeMap <Integer, Integer> bar = (TreeMap <Integer, Integer>) foo.clone ();  
    for (int i = 10; i < 20; i++) {  
        foo.put (i, i);  
    }  
  
    System.out.println (foo);  
    System.out.println (bar);  
}
```

— As you would expect, this will output

```
{0=0, 1=1, 2=2, 3=3, 4=4, 5=5, 6=6, 7=7, 8=8, 9=9, 10=10, 11=11, 12=12, 13=13, 14=14,  
15=15, 16=16, 17=17, 18=18, 19=19}  
{0=0, 1=1, 2=2, 3=3, 4=4, 5=5, 6=6, 7=7, 8=8, 9=9}
```

# But Often It Does Not

```
public void testY() {  
  
    ArrayList <ArrayList <Integer>> foo = new ArrayList <ArrayList <Integer>> ();  
    for (int i = 0; i < 10; i++) {  
        ArrayList <Integer> temp = new ArrayList <Integer> ();  
        for (int j = 0; j < 2; j++) {  
            temp.add (i);  
        }  
        foo.add (temp);  
    }  
  
    ArrayList <ArrayList <Integer>> bar = (ArrayList <ArrayList <Integer>>) foo.clone ();  
    for (int i = 0; i < 10; i++) {  
        foo.get (i).add (12);  
    }  
  
    System.out.println (foo);  
    System.out.println (bar);  
}
```

— This will output

```
[[0, 0, 12], [1, 1, 12], [2, 2, 12], [3, 3, 12], [4, 4, 12], [5, 5, 12], [6, 6, 12], [7, 7,  
12], [8, 8, 12], [9, 9, 12]]  
[[0, 0, 12], [1, 1, 12], [2, 2, 12], [3, 3, 12], [4, 4, 12], [5, 5, 12], [6, 6, 12], [7, 7,  
12], [8, 8, 12], [9, 9, 12]]
```

# Moral Of The Story

- Use “clone” with care!

# Lambdas

- One key idea from FP that is now almost universal...
  - ...except for Java...
  - ...is the idea of a “Lambda”
  - Have been around since the days of Lisp
  - Appears to be coming with Java 8 (in 18 to 24 months)
- We'll briefly cover it now
  - That way, when you run into it, you'll know what it is
  - Seems like no modern intro programming class should skip lambdas
  - Even if the class is in Java!

# What Is a Lambda?

- Think of it as a “function variable”
  - Whose “value” is set at run time, as the lambda is created
  - Lambdas are slightly different in every PL
  - Here’s an example, in a made-up mix of Java and C#

# What Is a Lambda?

```
delegate void TellMeIfIHaveEnough (int amtToCheck); // a lambda type (using C# syntax)

class LifeSavings {

    private final int moneyIHave;
    public LifeSavings (int moneyIn) {moneyIHave = moneyIn;}

    public TellMeIfIHaveEnough TellMeTheTruth () {

        TellMeIfIHaveEnough returnVal (int amtToCheck) => {
            if (amtToCheck > moneyIHave)
                System.out.println ("You don't have enough money");
            else
                System.out.println ("You have got enough money"); }

        return returnVal;
    }

    public TellMeIfIHaveEnough LieToMe (int moneyToPretendIHave) {
        LifeSavings temp = new LifeSavings (moneyToPretendIHave);
        return temp.TellMeTheTruth ();
    }
}

...
LifeSavings mySavings = new LifeSavings (10000);
TellMeIfIHaveEnough myFunc = mySavings.TellMeTheTruth ();
myFunc (1000000); // will say we don't have enough
myFunc = mySavings.LieToMe (2000000);
myFunc (1000000); // will say that we have enough
```

# Java and FP

- So, why has functional programming been relegated to some dead space towards the end of class?
  - Not because functional programming is useless
  - Or because it is unloved
  - Or because these ideas are unimportant
  - See the current buzz around Scala
  - Or look at the widespread use of “lambdas”



# Java and FP

- So, why has functional programming been relegated to some dead space towards the end of class?
  - Not because functional programming is useless
  - Or because it is unloved
  - Or because these ideas are unimportant
  - See the current buzz around Scala
  - Or look at the widespread use of “lambdas”
- It’s just that Java is an exceedingly un-functional language
  - In fact, Java does not even have functions!
  - That’s why we have the silly “RecursionShell” class
  - The “Java style” does not encourage immutability
  - No lambdas in Java. Wikipedia lists 39 widely-used languages; only 3 don’t have any recognizable form of lambdas (C, Java, Pascal)

# Java and FP

- So my final message is:
  - FP is a great paradigm to be aware of
  - You should always have the functional ideal in mind whenever you write code
  - Use it when appropriate
- But the language you use *does* have an effect on how you code
  - IMHO is not natural to write truly functional code in Java
  - IMHO, if you want to practice “pure” FP, choose another language (Haskell?)
  - Some others might disagree!

# Questions?