How to Hack Java Like a Functional Programmer

Nels E. Beckman

March 22, 2009

Chapter 1

Introduction

The first question that may come to mind when reading the title of the paper is, "why would I want to use Java to do something a fucntional language can already do better?" Well, that's probably not a bad question. In fact, these days picking the language that makes you the most productive is becoming easier and easier. Projects like Scala, and F# and SML.NET allow developpers to write code in strongly-typed, functional languages while still taking advantage of large existing code libraries and optimizing compilers, all thanks to the magic of byte-code.

But the reality is, sometimes you just have to use a particular language. Maybe it's because you are working in a team of programmers who don't know your language of choice. Maybe you're working on an existing program written in another language. The reasons very, but the end result is the same. In this article, I take it as a given that you're working in the Java programming language because it's the language I know best, but a lot of these techniques could be used in other languages as well.

So the next question you might ask is, "why would I want to hack like a functional programmer at all?" In other words, what are the benefits? Well, there are numerous nice things about functional languages that can help make us more productive as programmers. For one thing, functional programming languages tend to have really nice tools for manipulating collections, like lists and sets. Functions like `fold` and `map` are excellent general-purpose tools for modifying collections because they are both powerful and succinct. Another great thing about functional languages is their use of immutable data, which helps eliminate a lot of "gotchas" that seem to happen when using languages like C and Java. I don't know about other programmers, but when I use imperative languages I tend to spend a lot of time in the debugger, asking questions like, "why did this variable change?" and"what code is modifying this variable behind my back?" Because functional lanugages are structured around, well, functions, the variables that affect the current state of the program are usually on the stack, and the dataflow of the program is easy to follow and fix. There are plenty of other great reasons to program functionally.

Finally, who am I targeting with this article? Probably the most honest answer would be, myself three years ago. At that point in my life, I had just learned SML after previously programming exclusively in OO languages. I was amazed by their expressiveness. It also seemed like the code I wrote was frequently correct the first time I ran it, and/or was easy to debug. At the same time, all of the real research projects that I spent my time coding were in Java and C++. In those projects, I found that even though I wasn't using functional languages, my code could benefit from many of their ideas. I picked up most of the tricks and principles described in this article from friends, colleagues and from source code that I had to read myself. A better answer to the question is probably, anyone with an appreciation for the power of functional languages who regularly uses Java or other Object-Oriented programming languages.

So, let's get started!

Chapter 2

General Principles

In the first chapter, we discussed a couple of reasons why you might want to write "functional" Java code. Now we'll talk about *how* to do just that (and we'll talk *more* about why). In the next chapter, we will apply some of the general techniques we learned in this chapter to the task of writing more convenient collections and collection operations.

2.1 Be Immutable

The most important rule about writing functional code is to at all costs avoid writing code that modifies memory in place. Generally, objects in your program should be immutable and methods should really be functions over the parameters they are passed and the receiver object.

This guideline immediately causes four questions to come to mind, which I will answer in turn;

1. Why is immutability a good thing?
2. How do I make my objects immutable?
3. How do I accomplish the tasks I used to accomplish through effect-ful operators now that I have removed them?
4. Aren't there times when using effects really is a better way to go?

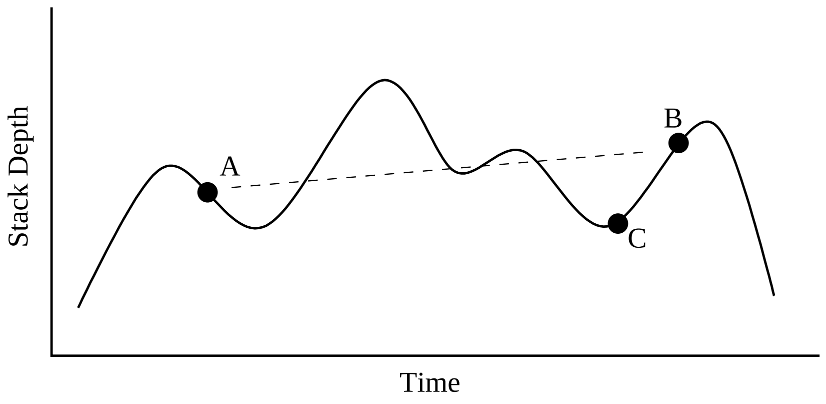
Why are immutable objects good? One of the things programmers first note when using functional languages is the unusual behavior of variables. Variables aren't really variable. Variables are "bound" to one particular value, and while I can create a new binding of the same variable, the original one still exists.

```
let str = "Hi" in
let f = (fn _ => str) in
let str = "Bye" in
...
```

In the above example, just because we have changed the value bound to the `str` variable does not mean that we have changed what earlier code sees that refers to the previous `str` variable. In other words, calling `f()` yields "Hi" rather than "Bye."

This behavior, while awkward at first, has great benefits when it comes to our ability to understand the code that we or others have written. Functional code is generally easier to understand because, as a result of the general immutability of memory, the flow of data through your program becomes much more explicit. If there is a problem with the data that appears in one part of a program, it is generally quite easy to work backwards to the location in your code that generated the bad data.

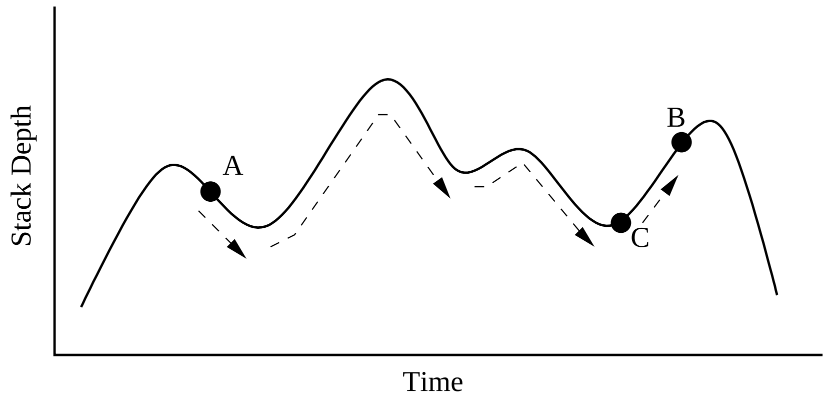
I like to illustrate this principle with two figures. In both figures, we imagine the depth of a program's stack graphed as a function over time (where higher up means more calls deep into the stack). The first figure graphs code that is using a mutable object `o`.



Let's suppose we are currently using a debugger because we have noticed a problem with our program's behavior, and we have the debugger stopped at a breakpoint at point `B`, which is in some function in our program. At this point, we notice that `o`'s field `f` has a value that it shouldn't have. How did it get this way? This object defines imperative methods, or methods that directly update the field `f`, and `f`'s value was changed to the wrong value at point `A` on the graph above.

Unfortunately, finding point `A` so that we can fix that code is quite difficult! In fact, many of us have had this experience, and it usually results in our shouting out, "how did the field get this value?" With a debugger it is very easy to examine all of the stack frames above `B` up to `C` to see which values were passed in to the current frame and how they were generated. But the current value of `f` was not generated somewhere above us in the stack. Rather it was the result of a modifying method directly changing the heap in a way that we could later observe. I have illustrated this "back channel" with a dotted line in the above figure.

When we use immutable objects, the situation is more like the second illustration.



When using immutable objects we are forced to return new objects whenever a method creates some kind of result. Therefore data in functional programs flows along the stack, as illustrated by the dotted arrows. In this case, if we observe an incorrect value, we can use our debugger to step up the stack one frame at a time. The erroneous value of `f` will undoubtably have come from an earlier method call, but finding which method call will be easy, as we can trace the origin of that object back to some method's *returned value*.

There's one more great reason to use immutable objects! It makes writing concurrent software much easier. This is true because many of the most insideous difficulties associated with concurrent programming are due to the trouble we must go through to protect state that multiple threads will read and at least one will modify. If no thread can ever modify an object *because it cannot be modified*, then access to that object never needs to be protected, for example, with `synchronized` blocks.

How do I make my objects immutable? It takes two steps to define immutable class `c`:

1. Mark all of `c`'s fields as **final**.
2. Make sure that any object referred to by a field of `c` is iteself immutable.

The first step is easy. Marking a field of a class as **final** assures that the field's value can only be set once, at construction time. After that attempting to assign to the field will result in a compile-time error. This will automatically prevent you from accidentally implementing mutating methods, like a setter, later on.

Coincidentally, **final** is a great modifier, and one that all Java programmers should understand well, as it holds different meanings in different situations.

(As testament to the power of this keyword, the book "Hardcore Java" [5] has dedicated and entire chapter to its various uses and benefits.) For our purposes, it is enough to know that Java will now attempt to optimize your code, armed with the knowledge that **final** fields will never change.

The seconds step is a little harder, and can be accomplished in a number of ways, but by requiring transitively reachable objects to also be immutable, we can ensure that an object will not be affected by the side-effects of other objects. To do this, we can do a couple of things. First off, certain built-in classes like `Integer`, `String` and `Float` are already immutable. There are also some methods provided by the Java standard library which allow us to make otherwise mutable objects into immutable ones. Next, if you are following my advice, most or at least many of the classes you define should be immutable as well. Finally, as a last resort, our class `c` can carefully control access to an object, and promise itself not to modify it.

Let's see each of these ideas in an example. We'll use the cannonical functional programming example, a simple expression language. The `IntValue` class, shown in Figure 2.1, is immutable because its only field points to an immutable class from the standard library, `Integer`. `Add`, from Figure 2.1, is immutable because its only fields are of type `Expr`, an interface that we have ourselves have created and that we know to be implemented only be immutable classes. Finally `SuperAdd`, which returns the sum of several expressions, takes two steps to achieve immutability; since the `List` interface is not generally implemented with immutable objects, we first take a defensive copy of the list passed into the constructor. This ensures that if the original list is later modified by another method, it won't affect the value of our new object. Then, to prevent our accidental modification, which could happen over time as the `SuperAdd` class evolves, we wrap our new list by calling `unmodifiableList`, which will throw an exception whenever a modifying method such as `add` is called.

```
class IntValue implements Expr {
  private final Integer value;

  public IntValue(int value) {
    this.value = Integer.valueOf(value);
  }

  public int val() { returns value.intValue(); }
}
```

Figure 2.1: The IntValue class is immutable because its only field, which is itself **final**, points to an immutable object defined by the Java standard library.

How do I accomplish the tasks I used to accomplish through effect-ful operators now that I have removed them? Okay, so we've removed muta-

```
class Add implements Expr {
  private final Expr e1;
  private final Expr e2;

  public Add(Expr e1, Expr e2) {
    this.e1 = e1;
    this.e2 = e2;
  }

  public int val() { return e1.val() + e2.val(); }
}
```

Figure 2.2: The Add class is immutable because its only fields point to objects of type `Expr`, an interface that we have defined ourselves and know all subtypes to be immutable.

ble methods from our class, but in order for an application to actual accomplish anything, it's going to have to transform data in some way. That's mostly what those old setter methods and other modifying methods were doing. How do we perform that functionality without modifying methods? To do this, we'll take a page straight out of the functional programming playbook. We'll take all our methods that used to modify the receiver object directly and have them return a new object. This object will be identical to the old one in every way, with the exception of the field or fields that we would have directly modified with our old method.

Let's look at an example inspired by some great code that I saw in the Polyglot Language Framework [7]. Polyglot contains a full Java typechecker, and to this end defines a number of classes that are used to define Java abstract syntax trees (ASTs). The following example defines the class that will be used to represent Java's conditional expression. (The conditional expression, whose syntax is $\mathtt{e}_b$? $\mathtt{e}_1$: $\mathtt{e}_2$, is the expression form of an "if" statement, and is covered in greater detail in Section 2.3.2.)

```
class ConditionalExpr implements Expr {
  private final Expr guard;
  private final Expr trueBranch;
  private final Expr falseBranch;

  private final Type type;

  ...
}
```

As you might anticipate, `ConditionalExpr` hold three expressions: one for the boolean expression that is first evaluated, one to be evaluated in the event

```
class SuperAdd implements Expr {
  private final List<Expr> es;

  public SuperAdd(List<Expr> es) {
    es = new ArrayList<Expr>(es);
    this.es = Collections.unmodifiableList(es);
  }

  public int val() {
    int result = 0;
    for( Expr e : es )
      result += e.val();

    return result;
  }
}
```

Figure 2.3: The SuperAdd class is immutable because it creates a defensive copy of the list that is passed into its constructor. Moreover, it uses the `unmodifiableList` utility method so if we accidentally try to modify it, an exception will be thrown.

that the guard was true, and another to be executed in the even the guard was false. It also holds a type. When the type-checker runs over a program's AST, it stores the type for each expression in the `type` field. This is great, except for one problem; type-checking occurs many stages after the initial AST is constructed, but all of the classes that make up Polyglot's AST are immutable. How will we assign a value to the `type` field when we compute it?

The answer, naturally, is that we won't assign any value to the field, but instead we'll construct an entirely new conditional expression object whose only difference is the value of the `type` field. Here's how we do it. Our class will define a getter method `type` but it will also define a "setter" method `type` which given a `Type` returns a new `ConditionalExpression`.

```
class ConditionalExpr implements Expr {
  ...

  private final Type type;

  public Type type() { return type; }

  public ConditionalExpr type(Type type) {
    return new
      ConditionalExpr(this.guard, this.trueBranch,
                      this.falseBranch, type);
  }

  ...
}
```

While this may seem a little strange, returning a new object rather than modifying the original object enormously simplifies the reasoning about your program. The reason is that any given object may refer to other objects through its fields, and certain invariants of that object may depend on the state of the objects to which its fields refer. Any time we modify an object an object directly, we risk invalidating invarints of other objects that refer to the mutated object, unless we are very careful. By creating a new object, we do not have to worry about objects whose invariants depended on the old one!

Note that this has performance implications, *both good and bad* that we will discuss in Section 2.1.1.

Aren't there times when using effects really is a better way to go? Of course functional code does not completely elliminate the idea of memory effects, for the simple reason that they are useful, and in many cases are necessary for getting certain jobs done. A computer is inherantly effect-ful. When we show something on a computer screen, for example, we are affecting the real world in a tangible way. And all computers work by effectively updating values held at certain locations inside of memory. It's just that these effects are hard for our puny brains to think about, so better to do without them as much as our existing programming abstractions allow.

That being said, when the problem you are solving demands the mutation of memory, you should not be afraid to do so. Many effecient algorithms from your favorite algorithms textbook [3] make use of mutable storage. As alluded to above, if you are dealing with the outside world in any way, either by interacting with a user, a file, or another machine, you will almost certainly have to write effect-ful code, and that code may very well mutate memory in place. Ideally, you should try to limit the effects of an operation to one part of a program, and not let the effect-ful nature of some code "leak" out of its interface. For example, many effecient aglorithms use mutable storage internally in such as way that is not visible to their clients. Of course this sort of isolation may not be possible.

If anything, the culture of functional programming should teach us that many more things may be accomplished without mutating memory than we previously thought, and there are a large number of elegant and effecient functional algorithms [6] that may be appropriate for any given programming challenge.

2.1.1 The Performance Costs of Immutability

Now we turn our attention to questions of performance. Once we start programming with large numbers of immutable objects, are we not setting ourselves up for decreased performance? It certainly would seem that way, given our suggestion of creating "setter" methods which allocate new objects and copy the contents of old ones. Will this style increase our rates of allocation and in turn lead to decreased performance?

I think that to some extent, the answer is "yes." Greater rates of allocation can lead to decreased performance. But in my experience this seems to rarely matter. Of course your millage may vary. But immutability can also open up new avenues for performance gain, primarily through the addition of multithreading and object sharing.

Regarding the first point, there is plenty of folk wisdom describing when to focus on performance improvements. Usually programmers agree that performance improvements can wait until later. I tend to agree, mainly because the majority of code that I write is not really performance intensive in any sense of the word. I would suggest that this is more than likely true for the code that you write as well, and I suspect that a large percentage of the code that is written every year has very low performance needs. The great thing about immutability is that it is an "optimization" of quality attributes that are almost always important; readability, understand-ability and ease of maintenance.

But of course performance cannot always be added after the fact. Somtimes early architectural decisions will fundamentally limit the performance of a particular application, and in these cases I can only offer some simple advice; learn to recognize potential performance pitfalls, at early stages. If this advice is weak, it is only a reflection of my own experience.

Yet I also believe that in many cases immutability can offer performance improvements. The first way is by allowing for more object sharing. When a program, or even a single class, has immutability as an invariant, we can start sharing objects without concern for program correctness. What is object sharing? It is when we take two separate objects that are both equal (in the `Object.equals` sense of the word) and replace them with one object. The `Boolean` class is a simple example of this concept. If we want to create an object that wraps the primitive value `true`, we can do so in two ways using the `Boolean` class, with a constructor or with the `valueOf` method:

```
Boolean b_1 = new Boolean(true);
Boolean b_2 = Boolean.valueOf(true);
```

The difference? The former approach must by definition allocate a new object, while the latter almost certainly will not. The reason? `Boolean` is an immutable class, and so every instance of the type that wraps the value `true` are equally true. Therefore, the `valueOf` method will return one of two existing instances every time it is called, preventing unncessary allocation. This technique is known as the Flyweight pattern [4] in OO circles, and is built into many of Java's library types. (For example, all of the primitive wrapper types as well as Strings, through a mechanism known as "string interning.") Used aggressively, this technique can greatly reduce the rate of allocation on certain object types.

The other reason that immutability can lead to increased performance is because it can allow us to more easily parallelize our code. This is particularly important as multi-core machines become a main-stay; we may all be writing a lot of multi-threaded code in the near future.

Object immutability makes multi-threaded programming easier for the simple reason that it allows us to side-step the things that traditionally make multi-threaded programming hard! When writing multi-threaded code, we typically worry about locks for providing mutual exclusion, and monitors or semaphores for inter-thread communication. But both of this tools are centered around mutability. When programmers use locks, it is to enable the modification of a memory location that multiple threads can concurrently access. If memory reachable from multiple threads is not to be modified, as is the case with immutable objects, then there are no actions which must be protected with mutual exclusion. In practice, it is much easier to write concurrent programs, or to transform serial programs into concurrent ones, when the majority of the operations in that program are performed on mutable data.

2.2 Interfaces to the Max

In this section we discuss interfaces. Interfaces are important, and as I will try to explain, probably should be your go-to means of defining a type when you are programming in Java.

2.2.1 When in Doubt: Interfaces and Delegation

For those writing Java who still have trouble designing their programs, I can offer two pieces of advice that will make you code dramatically easier to understand:

1. Use Java interfaces rather than classes for typing annotations that form part of a logical interface.
2. Accomplish code reuse through *delegation* rather than *inheritance*.

Let's go through these two points in a little bit more detail. The first point is a tad bit confusing because of it uses interface in two different ways, but the basic idea is simple: Use Java interfaces in method parameters. In other words, favor the former specification over the latter:

```
Iterable<String> intersection(Collection<String> c_1,
    Collection<String> c_2);

ArrayList<String> intersection(ArrayList<String> al_1,
    ArrayList<String> al_2);
```

In this case, most programmers would recognize the former as preferable due to their familiarity with the Java collections library. What I want to stress though is that this sort of reasoning should be use throughout your programs, even with the types that you define yourself.

Interfaces are flexible. They define the bare minimum necessary to have a type, and they force you as a programmer to think about exactly which services a given object intends to provide to clients. By keeping interfaces small and logically cohesive, we help promote reuse. Any given class will typically provide a number of services, many of which are orthogonal. For example, the ability to compare to other objects, the ability to posess an address in some coordinate space, or the ability to be sent over a network. By recognizing these orthogonal properties up front, we can define interfaces that capture those behaviors, and later define methods that will act on *all* object, even if unrelated, that have that particular behavior.

Interfaces also allow us to reimplement a functionaliy in a completely different way at a later point in time. If we must call a method that requires a class, rather than an interface, and we want to perform the services provided by that class in some completely different manner, our only choice is to extend the class, an inelegant solution which requires us to inherit code that we want nothing to do with.

Small, cohesive interfaces also allow us the posibilty to easily create annonymous inner classes, one-off classes that perform a particular task given a set of variables in scope. Anonymous classes are the first-class functions of the Java world, but really rely on small, well-defined interfaces, so that your code can remain readable.

While others have made these same points more elegantly that I have [2, item XXX], an interface-first style seems particularly well suited once we start viewing Java's types as ML data-types (Section 2.2.3).

The next point is probably a bit more controversial. I claim that you should use delegation whenever possible rather than inheritance. But this point is related to the first. Let me first show what I mean and then explain why I think it is a good idea.

Let's supposed you wanted to wanted to create a program with a resusable algorithm framework, which allows later classes to implement the details of each step of the algorithm. Common OO wisdom tells us that we shoud use the Template Pattern [**?** , chapter XXX], as in the example lifted from Wikipedia [**?**], shown in Figure 2.4.

The idea is, later on we can implement many different games, all of which will share the same basic code for game execution. For example, we could implement Monopoly, as shown in Figure 2.5.

2.2.2 Dynamic Dispatch as Pattern Matching

2.2.3 Datatypes

2.3 Expressions, Not Statements

2.3.1 The Poor-Man's 'Let' Binding

```
let x = e_1 : C_1 in e_2 : C_2
```

```
new Object() {
  C_1 eval(C_2 x) {
    return e_2;
  }
}.eval(e_1)
```

2.3.2 The Conditional Expression

2.3.3 Recursive Methods

2.4 Classes as Modules

Here's a opportunity for Kevin's "functor" trick.

```
public static <T> List<T>
    createList(Class<? extends List> cl, T... items) {
  List<T> result = cl.newInstance();

  for(T t : items) {
    result.add(t);
  }
  return result;
}
```

2.5 Design Patterns?

The "FUNCTIONAL" design patterns are Builder, Adapter and its cousin Decorator, Flyweight.

2.6 Good Things in Static Land

2.6.1 Static Methods, Not Fields!

2.6.2 Java Has Type Inference! Well, sort of...

```
/**
 * An abstract class that is common to several games in
 * which players play against the others, but only one is
 * playing at a given time.
 */

abstract class Game {

  protected int playersCount;

  abstract void initializeGame();

  abstract void makePlay(int player);

  abstract boolean endOfGame();

  abstract void printWinner();

  /* A template method : */
  final void playOneGame(int playersCount) {
      this.playersCount = playersCount;
      initializeGame();
      int j = 0;
      while (!endOfGame()) {
          makePlay(j);
          j = (j + 1) % playersCount;
      }
      printWinner();
  }
}
```

Figure 2.4: An abstract game with a template method. Concrete games must implement each step of the game-playing process.

```
class Monopoly extends Game {

  /* Implementation of necessary concrete methods */

  void initializeGame() {
      // ...
  }

  void makePlay(int player) {
      // ...
  }

  boolean endOfGame() {
      // ...
  }

  void printWinner() {
      // ...
  }

  /* Specific declarations for the Monopoly game. */

  // ...

}
```

Figure 2.5: A Monopoly game, which implements the steps of the `Game` template it extends.

Chapter 3

Case Study: Collections

One of the first things I noticed about SML was how easily it allowed programmers to created and modify collections and other data structures. Java has a large standard library full of well-written code, but sometimes its collections just aren't really all that convenient to use. In this section I will discuss a few tricks that will help improve your use of collections.

3.0.3 Tuples are Your Friend

Tuples! Once you use them, you'll wonder how you ever got by without them. The ability to, say, return multiple arguments from a function without defining a new data type is incredibly convenient. So convenient, in fact, that you wonder why other languages don't have them. Unfortunately, it's true; Java has no tuples, but that's no reason why we can't create our own.[1] Sure, this idea is pretty simple, but one I've found remarkably useful.

Consider the code shown in Figure 3.1. This code is pretty simple, but shows you a basic immutable pair. This code uses Java's Generics to define a pair of two objects `fst` and `snd`. The methods for returning these elements are similarly named. (Many of the examples I show in this article use functional naming conventions, rather than OO ones.)

The most interesting thing to note about this example is the static factory method, which we will see in several other examples as well. The `createPair` method can be used, like the constructor itself, to create a new Pair object from its two constituents. However, thanks to a requirement in the Java specification, the `createPair` method can be called without specifying the type arguments. The same cannot be said for the the constructor. In fact, liberal use of static factory methods can greatly decrease the number of required typing annotions. In this example, creating a pair goes from:

```
Pair<Integer,Boolean> p =
```

[1] Thanks to Kevin Bierhoff for showing me this idea.

```java
  final class Pair<T,S> {
    private final T fst;
    private final S snd;

    public Pair(T fst, S snd) {
      this.fst = fst;
      this.snd = snd;
    }

    public static <T,S>
    Pair<T,S> createPair(T fst, S snd) {
      return new Pair<T,S>(fst, snd);
    }

    public T fst() { return fst; }

    public S snd() { return snd; }
  }
```

Figure 3.1: A simple pair class in Java. Notice the static factory method, so that clients do not have to specify type arguments to the constructor.

```java
  new Pair<Integer,Boolean>(1,true);
```

to

```java
Pair<Integer,Boolean> p = createPair(1,true);
```

And may also make it easier to modify the object later on, if object invariants change.

There's no reason why you can't create larger tuples as well, but you'll probably need to define a new class for each one. I've used triples myself, but no larger. Unfortunately, thanks to Java's verbose typing syntax for Generics, the typing annotations get pretty unwieldy quickly. In a language like SML, this problem doesn't exist because of type inference. Nonetheless, I still find pairs to be convenient is many situations.

3.0.4 Intializing Collections

Java has very few collections that are actually part of the syntax itself. Okay, really there's just one: arrays. And while arrays are occasionally useful, it's nice to have other collections supported natively by the language itself. In functional languages, lists are ubiquitous, but some languages even have buit-in syntax for records and maps. Here are a couple of ways we can initialize collections as part of larger expressions.

Using Varargs Java has var-args, and its high time we use them! One thing that is extremely convenient in languages like SML is the existence of special list syntax. At any point in code, I can directly create a list initialized to values of my choosing. For instance,

```
  let l = [1;2;3;4;5] in ...
```

allows me to create the list consisting of the first five natural numbers in order. Using methods from the java.util.Collections class, we can create empty lists, and singleton lists, two degenerate cases, corresponding to `[]` and `[1]` in ML, but lists of size greater than one must be created using the `add` method. Unfortunately, the signature of this method is,

```
  boolean add(E o);
```

making it impossible to chain together several calls to `add` in a row.

Fortunately, we could use Java's var-args functionality to achieve something close to the level of succint-ness provided by SML. In Figure 3.2, you can see the definition of the `createList` method. This trick comes to me via the AtomJava source code [1], a good source for functional Java. The `createList` method takes a variable number of arguments of type `T`, which are put into the array `items`. This array is then used to construct a new `ArrayList`, which is returned.

```
public static <T> List<T> createList(T... items) {
  List<T> result = new ArrayList<T>(items.length);
  for(T t : items) {
    result.add(t);
  }
  return result;
}
```

Figure 3.2: Var-args as a "poor man's" list literal. The definition of the createList method static factory method is shown here..

We would use this list as follows:

```
  List<Integer> l = createList(1,2,3,4,5);
```

Again, we because we've created a static factory method, it is not (usually) necessary to pass explicit typing arguments. It might be a good idea to return an immutable view of this list using the java.util.Collections.immutableList() method. We cover immutability (and why this might be a good idea) in Section 2.1.

Initialize Collections in Intializers This trick (and I purposely use the word "trick") should be shown with some caveats. It may may your code less readable, and it is not in general safe to use with any class. In fact, when my colleague Neel first showed me this trick,[2] it took my research group several minutes to figure out why it even worked. If that's not a reason to take something with a grain of salt, I'm not sure what is.

Nonetheless, occasionally you would like to create and initialize collections all in one expression, and those collections (maps being the best example) are not amenable to the var-args trick. (HERE'S WHERE I LOOK UP IN THE SPEC WHEN THE INIT BLOCK IS EXECUTED). This trick takes advantage of our old friend, the annonymous inner class:

```
foo(new HashMap<Integer,Boolean>(){{put(4,true);
                                    put(5,false);}});
```

What's going on here? Well, all in one expression, we are creating a new, anonymous sub-class of the `HashMap` class, and in the constructor of this new class, we call the put method, adding two items to the map. This works because the implicit call to the super-constructor creates and initializes the map before our new constructor code is called. Finally, in the super-expression, we pass an instance of this new class to the `foo` method. We couldn't do this merely by stringing together calls to the `put` method on a regular map because that methods return type is a boolean.

3.0.5 Roll Your Own Collection Operators

Now that we've got freshly initialized collections, let's develop some of our own FP-like collection utilities. Almost all functional languages take advantage of higher-order functions by providing some neat collection utilities in their libraries. Functions like `map` and `fold` are perfect examples of this. They allow programmers to define functions that will then be executed on each element of a collection. Java is missing first-class functions, but thanks to anonymous inner classes and Java generics, we can achieve something remarkably similay, albeit with somewhat uglier syntax. (Notice a reoccuring theme here?)

Let's start with our own version of `map`, seen in Figure 3.3. The job of `map` is to take a list and a function and apply that function to each element of the list, putting the result of the function call into a new list that will be returned. Since I can't write a method that takes a map and a function, as functions aren't first-class, I instead define a new interface, `Mapping`. `Mapping` simply exists so that clients who want to use the `map` method have a way of defining the function that will be called on each element of the list. Looking at the implementation of the `map` method, we see that we simply create a new list, call `eval` on each element of the loop and put the result into a newly created list. Finally, notice our use of the wildcard (?) which allows the mapping interface to operate on lists that contain sub-types of the argument it expects.

Here's how we might use our new method:

```
  Util.map(l, new Mapping<Integer,Integer>(){
```

[2]This trick comes to me by way of Neelakantan Krishnashwami.

```
  interface Mapping<I,O> {
    public O eval(I elem);
  }

  static <I,O> List<O> map(List<? extends I> list,
                           Mapping<I,O> fun) {
    List<O> result = new ArrayList<O>(list.size());
    for( I elem : list ) {
      result.add(fun.eval(elem));
    }
    return result;
  }
```

Figure 3.3: A user-defined map function and the Mapping interface..

```
    public Integer eval(Integer elem) {
      return 2 * elem; }
  });
```

The iter method defined in Figure 3.0.5 is extremely similar in its conception, but rather than creating a new list that is the result of applying the method to each element, the purpose is to apply a method to each element purely for its side-effect.

```
interface Action<I> {
  public void eval(I elem);
}

static <I> void iter(List<? extends I> list,
                     Action<I> fun) {
  for(I elem : list) {
    fun.eval(elem);
  }
}
```

Figure 3.4: A user-defined iter function and the Action interface.

One might note that `iter` is conceptually quite similar to Java's enhanced for loop (in fact, it *contains* an enhanced for loop). Is this method redundant? I would argue that `iter` can help avoid code duplication in situations where the same opertation must be applied based on the contents of multiple collections. Of course, this probably only matters if the task you intend to perform is non-trivial.

How about other ways to inspect and create news lists? I'm glad you asked!

In Figure 3.0.5 I've recreated the classic list operations, head (hd), tail (tl) and cons. These methods allows us to manipulate lists in an immutable fashion which can be quite useful. There's not too much to these methods except pointing out that they can in fact be created.

```java
static <T> T hd(List<? extends T> list) {
  return list.get(0);
}

static <T> List<T> tl(List<? extends T> list) {
  if( list.size() <= 1 )
    return Collections.emptyList();
  else
    return
    new ArrayList<T>(list.subList(1, list.size()-1));
}

static <T> List<T> cons(T head, List<? extends T> tail) {
  final List<T> result = new ArrayList<T>(tail.size()+1);
  result.add(head);
  result.addAll(tail);
  return result;
}
```

Figure 3.5: A collection of non-mutating list methods.

3.0.6 Impedence Matching with Simple Maps

Chapter 4

Conclusion

4.1 Further Reading

Bibliography

[1] AtomJava. Atomjava. `http://wasp.cs.washington.edu/wasp_atomjava.html`.

[2] Joshua Bloch. *Effective Java (2nd Edition)*. Prentice Hall, 2008.

[3] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*. McGraw-Hill Science/Engineering/Math, 2003.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[5] Robert Simmons Jr. *Hardcore Java*. O'Reilly Media, 2004.

[6] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[7] Polyglot. Polyglot. `http://www.cs.cornell.edu/projects/polyglot/`.