Functional Programming in Java

How functional techniques improve your Java programs

Pierre-Yves Saumont

MANNING

SAMPLE CHAPTER

Functional Programming in Java

by Pierre-Yves Saumont

Chapter 4

brief contents

4 Recursion, corecursion, and memoization

This chapter covers

- Understanding recursion and corecursion
- Working with recursive functions
- Composing a huge number of functions
- Speeding up functions with memoization

The previous chapter introduced powerful methods and functions, but some shouldn't be used in production because they can overflow the stack and crash the application (or at least the thread in which they're called). These "dangerous" methods and functions are mainly explicitly recursive, but not always. You've seen that composing functions can also overflow the stack, and this can occur even with nonrecursive functions, although this isn't common.

In this chapter, you'll learn how to turn stack-based functions into heap-based functions. This is necessary because the stack is a limited memory area. For recursive functions to be safe, you have to implement them in such a way that they use the heap (the main memory area) instead of the limited stack space. To understand the problem completely, you must first understand the difference between recursion and corecursion.

4.1 Understanding corecursion and recursion

Corecursion is composing computing steps by using the output of one step as the input of the next one, starting with the first step. *Recursion* is the same operation, but starting with the last step. In recursion, you have to delay evaluation until you encounter a base condition (corresponding to the first step of corecursion).

Let's say you have only two instructions in your programming language: incrementation (adding 1 to a value) and decrementation (subtracting 1 from a value). As an example, you'll implement addition by composing these instructions.

4.1.1 Exploring corecursive and recursive addition examples

To add two numbers, x and y, you can do the following:

- If y = 0, return x.
- Otherwise, increment x, decrement y, and start again.

This can be written in Java as follows:

```
static int add(int x, int y) {
  while(y > 0) {
    x = ++x;
    y = --y;
  }
  return x;
}
```

Here's a simpler approach:

```
static int add(int x, int y) {
  while(y-- > 0) {
    x = ++x;
  }
  return x;
}
```

There's no problem with using the parameters `x` and `y` directly, because in Java, all parameters are passed by value. Also note that you use post-decrementation to simplify coding. You could have used pre-decrementation by slightly changing the condition, thus switching from iterating from `y` to `1`, to iterating from `y - 1` to `0`:

```
static int add(int x, int y) {
  while(--y >= 0) {
    x = ++x;
  }
  return x;
}
```

The recursive version is trickier, but still simple:

```
static int addRec(int x, int y) {
  return y == 0
      ? x
      : addRec(++x, --y);
}
```

Both approaches seem to work, but if you try the recursive version with big numbers, you may have a surprise. Although this version,

```
addRec(10000, 3);
```

produces the expected result of 10,003, switching the parameters, like this,

```
addRec(3, 10000);
```

produces a `StackOverflowException`.

4.1.2 Implementing recursion in Java

To understand what's happening, you must look at how Java handles method calls. When a method is called, Java suspends what it's currently doing and pushes the environment on the stack to make a place for executing the called method. When this method returns, Java pops the stack to restore the environment and resume program execution. If you call one method after another, the stack always holds at most one of these method call environments.

But methods aren't composed only by calling them one after the other. Methods call methods. If `method1` calls `method2` as part of its implementation, Java again suspends the `method1` execution, pushes the current environment on the stack, and starts executing `method2`. When `method2` returns, Java pops the last pushed environment from the stack and resumes execution (of `method1` in this case). When `method1` completes, Java again pops the last environment from the stack and resumes what it was doing before calling this method.

Method calls may be deeply nested, and this nesting depth does have a limit, which is the size of the stack. In current situations, the limit is somewhere around a few thousand levels, and it's possible to increase this limit by configuring the stack size. But because the same stack size is used for all threads, increasing the stack size generally wastes space. The default stack size varies from 320 KB to 1024 KB, depending on the version of Java and the system used. For a 64-bit Java 8 program with minimal stack usage, the maximum number of nested method calls is about 7,000. Generally, you won't need more, except in specific cases. One such case is recursive method calls.

4.1.3 Using tail call elimination

Pushing the environment on the stack is typically necessary in order to resume computation after the called method returns, but not always. When the call to a method is the last thing the calling method does, there's nothing to resume when the method returns, so it should be OK to resume directly with the caller of the current method instead of the current method itself. A method call occurring in the last position, meaning it's the last thing to do before returning, is called a *tail call*. Avoiding pushing the environment to the stack to resume method processing after a tail call is an optimization technique known as *tail call elimination* (TCE). Unfortunately, Java doesn't use TCE.

Tail call elimination is sometimes called *tail call optimization* (TCO). TCE is generally an optimization, and you can live without it. But when it comes to recursive function calls, TCE is no longer an optimization. It's a necessary feature. That's why TCE is a better term than TCO when it comes to handling recursion.

4.1.4 Using tail recursive methods and functions

Most functional languages have TCE. But TCE isn't enough to make every recursive call possible. To be a candidate for TCE, the recursive call must be the last thing the method has to do.

Consider the following method, which is computing the sum of the elements of a list:

```
static Integer sum(List<Integer> list) {
    return list.isEmpty()
        ? 0
        : head(list) + sum(tail(list));
  }
```

This method uses the `head` and `tail` methods from chapter 3. The recursive call to the `sum` method isn't the last thing the method has to do. The four last things the method does are as follows:

- Calls the `head` method
- Calls the `tail` method
- Calls the `sum` method
- Adds the result of `head` and the result of `sum`

Even if you had TCE, you wouldn't be able to use this method with lists of 10,000 elements. But you can rewrite this method in order to put the call to `sum` in the tail position:

```
static Integer sum(List<Integer> list) {
  return sumTail(list, 0);
}

static Integer sumTail(List<Integer> list, int acc) {
  return list.isEmpty()
      ? acc
      : sumTail(tail(list), acc + head(list));
}
```

Here, the `sumTail` method is tail recursive and can be optimized through TCE.

4.1.5 Abstracting recursion

So far, so good, but why bother with all this if Java doesn't have TCE? Well, Java doesn't have it, but you can do without it. All you need to do is the following:

- Represent unevaluated method calls
- Store them in a stack-like structure until you encounter a terminal condition
- Evaluate the calls in "last in, first out" (LIFO) order

Most examples of recursive methods use the factorial function. Other examples use the Fibonacci series. The factorial method presents no particular interest beside being recursive. The Fibonacci series is more interesting, and we'll come back to it later. To start with, you'll use the much simpler recursive addition method shown at the beginning of this chapter.

Recursive and corecursive functions are both functions where `f(n)` is a composition of `f(n - 1)`, `f(n - 2)`, `f(n - 3)`, and so on, until a terminal condition is encountered (generally `f(0)` or `f(1)`). Remember that in traditional programming, composing generally means composing the results of an evaluation. This means that composing function `f(a)` and `g(a)` consists of evaluating `g(a)` and then using the result as input to `f`. But it doesn't have to be done that way. In chapter 2, you developed a `compose` method to compose functions, and a `higherCompose` function to do the same thing. Neither evaluated the composed functions. They only produced another function that could be applied later.

Recursion and corecursion are similar, but there's a difference. You create a list of function calls instead of a list of functions. With corecursion, each step is terminal, so it may be evaluated in order to get the result and use it as input for the next step. With recursion, you start from the other end, so you have to put non-evaluated calls in the list until you find a terminal condition, from which you can process the list in reverse order. You stack the steps until the last one is found, and then you process the stack in reverse order (last in, first out), again evaluating each step and using the result as the input for the next (in fact, the previous) one.

The problem is that Java uses the thread stack for both recursion and corecursion, and its capacity is limited. Typically, the stack overflows after 6,000 to 7,000 steps. What you have to do is create a function or a method returning a non-evaluated step. To represent a step in the calculation, you'll use an abstract class called `TailCall` (because you want to represent a call to a method that appears in the tail position).

This `TailCall` abstract class has two subclasses. One represents an intermediate call, when the processing of one step is suspended to call the method again for evaluating the next step. This is represented by a subclass named `Suspend`. It's instantiated with `Supplier<TailCall>>`, which represents the next recursive call. This way, instead of putting all `TailCalls` in a list, you'll construct a linked list by linking each tail call to the next. The benefit of this approach is that such a linked list is a stack, offering constant time insertion as well as constant time access to the last inserted element, which is optimal for a LIFO structure.

The second subclass represents the last call, which is supposed to return the result, so you'll call it `Return`. It won't hold a link to the next `TailCall`, because there's nothing next, but it'll hold the result. Here's what you get:

```
public abstract class TailCall<T> {
  public static class Return<T> extends TailCall<T> {
    private final T t;
    public Return(T t) {
      this.t = t;
```

```
    }
  }

  public static class Suspend<T> extends TailCall<T> {
    private final Supplier<TailCall<T>> resume;
    private Suspend(Supplier<TailCall<T>> resume) {
      this.resume = resume;
    }
  }
}
```

To handle these classes, you'll need some methods: one to return the result, one to return the next call, and one helper method to determine whether a `TailCall` is a `Suspend` or a `Return`. You could avoid this last method, but you'd have to use `instanceof` to do the job, which is ugly. The three methods are as follows:

```
public abstract TailCall<T> resume();
public abstract T eval();
public abstract boolean isSuspend();
```

The `resume` method has no implementation in `Return` and will throw a runtime exception. The user of your API shouldn't be in a situation to call this method, so if it's eventually called, it'll be a bug and you'll stop the application. In the `Suspend` class, this method will return the next `TailCall`.

The `eval` method returns the result stored in the `Return` class. In the first version, it'll throw a runtime exception if called on the `Suspend` class.

The `isSuspend` method returns `true` in `Suspend`, and `false` in `Return`. The following listing shows this first version.

Listing 4.1 The `TailCall` interface and its two implementations

```
public abstract class TailCall<T> {

  public abstract TailCall<T> resume();
  public abstract T eval();
  public abstract boolean isSuspend();

  public static class Return<T> extends TailCall<T> {

    private final T t;

    public Return(T t) {
      this.t = t;
    }

    @Override
    public T eval() {
      return t;
    }

    @Override
    public boolean isSuspend() {
      return false;
    }
```

```
    @Override
    public TailCall<T> resume() {
      throw new IllegalStateException("Return has no resume");
    }
  }

  public static class Suspend<T> extends TailCall<T> {

    private final Supplier<TailCall<T>> resume;

    public Suspend(Supplier<TailCall<T>> resume) {
      this.resume = resume;
    }

    @Override
    public T eval() {
      throw new IllegalStateException("Suspend has no value");
    }

    @Override
    public boolean isSuspend() {
      return true;
    }

    @Override
    public TailCall<T> resume() {
      return resume.get();
    }
  }
}
```

To make the recursive method `add` work with any number of steps (within the limits of available memory!), you have a few changes to make. Starting with your original method,

```
static int add(int x, int y) {
  return y == 0
      ? x
      : add(++x, --y) ;
}
```

you need to make the modifications shown in the following listing.

Listing 4.2 The modified recursive method

```
static TailCall<Integer> add(int x, int y) {
  return y == 0
      ? new TailCall.Return<>(x)
      : new TailCall.Suspend<>(() -> add(x + 1, y - 1));
}
```

B Method returns a TailCall

C In terminal condition, a Return is returned

D In nonterminal condition, a Suspend is returned

This method returns a `TailCall<Integer>` instead of an `int` **B**. This return value may be a `Return<Integer>` if you've reached a terminal condition **C**, or a `Suspend`

<Integer> if you haven't D. The Return is instantiated with the result of the computation (which is x, because y is 0), and the Suspend is instantiated with a Supplier <TailCall<Integer>>, which is the next step of the computation in terms of execution sequence, or the previous in terms of calling sequence. It's important to understand that Return corresponds to the last step in terms of the method call, but to the first step in terms of evaluation. Also note that we've slightly changed the evaluation, replacing ++x and --y with x + 1 and y – 1. This is necessary because we're using a closure, which works only if closed-over variables are effectively final. This is cheating, but not too much. We could have created and called two methods, dec and inc, using the original operators.

This method returns a chain of TailCall instances, all being Suspend instances except the last one, which is a Return.

So far, so good, but this method isn't a drop-in replacement for the original one. Not a big deal! The original method was used as follows:

```
System.out.println(add(x, y))
```

You can use the new method like this:

```
TailCall<Integer> tailCall = add(3, 100000000);
while(tailCall.isSuspend()) {
  tailCall = tailCall.resume();
}
System.out.println(tailCall.eval());
```

Doesn't it look nice? If you feel frustrated, I understand. You thought you would just use a new method in place of the old one in a transparent manner. You seem to be far from this. But you can make things much better with a little effort.

4.1.6 Using a drop-in replacement for stack-based recursive methods

In the beginning of the previous section, I said that the user of your recursive API would have no opportunity to mess with the TailCall instances by calling resume on a Return or eval on a Suspend. This is easy to achieve by putting the evaluation code in the eval method of the Suspend class:

```
public static class Suspend<T> extends TailCall<T> {

  ...

  @Override
  public T eval() {
    TailCall<T> tailRec = this;
    while(tailRec.isSuspend()) {
      tailRec = tailRec.resume();
    }
    return tailRec.eval();
  }
```

Now you can get the result of the recursive call in a much simpler and safer way:

```
add(3, 100000000).eval()
```

But this isn't what you want. You want to get rid of this call to the `eval` method. This can be done with a helper method:

```
public static int add(int x, int y) {
  return addRec(x, y).eval();
}

private static TailCall<Integer> addRec(int x, int y) {
  return y == 0
      ? ret(x)
      : sus(() -> addRec(x + 1, y - 1));
}
```

Now you can call the `add` method exactly as the original one. You can make your recursive API easier to use by providing static factory methods to instantiate `Return` and `Suspend`, which also allows you to make the `Return` and `Suspend` internal subclasses private:

```
public static <T> Return<T> ret(T t) {
  return new Return<>(t);
}

public static <T> Suspend<T> sus(Supplier<TailCall<T>> s) {
  return new Suspend<>(s);
}
```

The following listing shows the complete `TailCall` class. It adds a private no-args constructor to prevent extension by other classes.

Listing 4.3 The complete `TailCall` class

```
public abstract class TailCall<T> {

  public abstract TailCall<T> resume();
  public abstract T eval();
  public abstract boolean isSuspend();

  private TailCall() {}

  private static class Return<T> extends TailCall<T> {

    private final T t;

    private Return(T t) {
      this.t = t;
    }

    @Override
    public T eval() {
      return t;
    }

    @Override
    public boolean isSuspend() {
      return false;
    }
```

```
    @Override
    public TailCall<T> resume() {
      throw new IllegalStateException("Return has no resume");
    }
  }

  private static class Suspend<T> extends TailCall<T> {

    private final Supplier<TailCall<T>> resume;

    private Suspend(Supplier<TailCall<T>> resume) {
      this.resume = resume;
    }

    @Override
    public T eval() {
      TailCall<T> tailRec = this;
      while(tailRec.isSuspend()) {
        tailRec = tailRec.resume();
      }
      return tailRec.eval();
    }

    @Override
    public boolean isSuspend() {
      return true;
    }

    @Override
    public TailCall<T> resume() {
      return resume.get();
    }
  }

  public static <T> Return<T> ret(T t) {
    return new Return<>(t);
  }

  public static <T> Suspend<T> sus(Supplier<TailCall<T>> s) {
    return new Suspend<>(s);
  }
}
```

Now that you have a stack-safe tail recursive method, can you do the same thing with a function?

4.2 Working with recursive functions

In theory, recursive functions shouldn't be more difficult to create than methods, if functions are implemented as methods in an anonymous class. But lambdas aren't implemented as methods in anonymous classes.

The first problem is that, in theory, lambdas can't be recursive. But this is theory. In fact, you learned a trick to work around this problem in chapter 2. A statically defined recursive add function looks like this:

```
static Function<Integer, Function<Integer, TailCall<Integer>>> add =
    a -> b -> b == 0
```

```
        ? ret(a)
        : sus(() -> ContainingClass.add.apply(a + 1).apply(b - 1));
```

Here, `ContainingClass` stands for the name of the class in which the function is defined. Or you may prefer an instance function instead of a static one:

```
Function<Integer, Function<Integer, TailCall<Integer>>> add =
    a -> b -> b == 0
        ? ret(a)
        : sus(() -> this.add.apply(a + 1).apply(b - 1));
```

But here, you have the same problem you had with the `add` method. You must call `eval` on the result. You could use the same trick, with a helper method alongside the recursive implementation. But you should make the whole thing self-contained. In other languages, such as Scala, you can define helper functions locally, inside the main function. Can you do the same in Java?

4.2.1 Using locally defined functions

Defining a function inside a function isn't directly possible in Java. But a function written as a lambda is a class. Can you define a local function in that class? In fact, you can't. You can't use a static function, because a local class can't have static members, and anyway, they have no name. Can you use an instance function? No, because you need a reference to `this`. And one of the differences between lambdas and anonymous classes is the `this` reference. Instead of referring to the anonymous class instance, the `this` reference used in a lambda refers to the enclosing instance.

The solution is to declare a local class containing an instance function, as shown in the following listing.

Listing 4.4 A standalone tail recursive function

```
static Function<Integer, Function<Integer, Integer>> add = x -> y -> {
  class AddHelper {
    Function<Integer, Function<Integer, TailCall<Integer>>> addHelper =
        a -> b -> b == 0
            ? ret(a)
            : sus(() -> this.addHelper.apply(a + 1).apply(b – 1));
  }
  return new AddHelper().addHelper.apply(x).apply(y).eval();
};
```

The this reference refers to the AddHelper class.

This function may be used as a normal function:

```
add.apply(3).apply(100000000)
```

4.2.2 Making functions tail recursive

Previously, I said that a simple recursive functional method computing the sum of elements in a list couldn't be handled safely because it isn't tail recursive:

```
static Integer sum(List<Integer> list) {
  return list.isEmpty()
      ? 0
      : head(list) + sum(tail(list));
}
```

You saw that you had to transform the method as follows:

```
static Integer sum(List<Integer> list) {
    return sumTail(list, 0);
}

static Integer sumTail(List<Integer> list, int acc) {
  return list.isEmpty()
      ? acc
      : sumTail(tail(list), acc + head(list));
}
```

The principle is quite simple, although it's sometimes tricky to apply. It consists of using an accumulator holding the result of the computation. This accumulator is added to the parameters of the method. Then the function is transformed into a helper method called by the original one with the initial value of the accumulator. It's important to make this process nearly instinctive, because you'll have to use it each time you want to write a recursive method or function.

It may be OK to change a method into two methods. After all, methods don't travel, so you only have to make the main method public and the helper method (the one doing the job) private. The same is true for functions, because the call to the helper function by the main function is a closure. The main reason to prefer a locally defined helper function over a private helper method is to avoid name clashes.

A current practice in languages that allow locally defined functions is to call all helper functions with a single name, such as `go` or `process`. This can't be done with nonlocal functions (unless you have only one function in each class). In the previous example, the helper function for `sum` was called `sumTail`. Another current practice is to call the helper function with the same name as the main function with an appended underscore, such as `sum_`. Whatever system you choose, it's useful to be consistent. In the rest of this book, I'll use the underscore to denote tail recursive helper functions.

4.2.3 Doubly recursive functions: the Fibonacci example

No book about recursive functions can avoid the Fibonacci series function. Although it's totally useless to most of us, it's ubiquitous and fun. Let's start with the requirements, in case you've never met this function.

The Fibonacci series is a suite of numbers, and each number is the sum of the two previous ones. This is a recursive definition. You need a terminal condition, so the full requirements are as follows:

- $f(0) = 0$
- $f(1) = 1$
- $f(n) = f(n-1) + f(n-2)$

This isn't the original Fibonacci series, in which the first two numbers are equal to 1. Each number is supposed to be a function of its position in the series, and that position starts at 1. In computing, you generally prefer to start at 0. Anyway, this doesn't change the problem.

Why is this function so interesting? Instead of answering this question right now, let's try a naive implementation:

```
public static int fibonacci(int number) {
  if (number == 0 || number == 1) {
    return number;
  }
  return fibonacci(number - 1) + fibonacci(number - 2);
}
```

Now let's write a simple program to test this method:

```
public static void main(String args[]) {
  int n = 10;
  for(int i = 0; i <= n; i++){
    System.out.print(fibonacci(i) +" ");
  }
}
```

If you run this test program, you'll get the first 10 (or 9, according to the original definition) Fibonacci numbers:

```
0 1 1 2 3 5 8 13 21 34 55
```

Based on what you know about naive recursion in Java, you may think that this method will succeed in calculating `f(n)` for `n`, up to 6,000 to 7,000 before overflowing the stack. Well, let's check it. Replace `int n = 10` with `int n = 6000` and see what happens. Launch the program and take a coffee break. When you return, you'll realize that the program is still running. It will have reached somewhere around 1,836,311,903 (your mileage may vary—you could get a negative number!), but it'll never finish. No stack overflow, no exception—just hanging in the wild. What's happening?

The problem is that each call to the function creates two recursive calls. So to calculate `f(n)`, you need $2n$ recursive calls. Let's say your method needs 10 nanoseconds to execute. (Just guessing, but you'll see soon that it doesn't change anything.) Calculating `f(5000)` will take $2^{5000} \times 10$ nanoseconds. Do you have any idea how long this is? This program will never terminate because it would need longer than the expected duration of the solar system (if not the universe!).

To make a usable Fibonacci function, you have to change it to use a single tail recursive call. There's also another problem: the results are so big that you'll soon get an arithmetic overflow, resulting in negative numbers.

EXERCISE 4.1

Create a tail recursive version of the Fibonacci functional method.

HINT

The accumulator solution is the way to go. But there are two recursive calls, so you'll need two accumulators.

SOLUTION 4.1

Let's first write the signature of the helper method. It'll take two `BigInteger` instances as accumulators, and one for the original argument, and it'll return a `BigInteger`:

```
private static BigInteger fib_(BigInteger acc1, BigInteger acc2,
                                                BigInteger x) {
```

You must deal with the terminal conditions. If the argument is `0`, you return `0`:

```
private static BigInteger fib_(BigInteger acc1, BigInteger acc2,
                                                BigInteger x) {
  if (x.equals(BigInteger.ZERO)) {
    return BigInteger.ZERO;
```

If the argument is `1`, you return the sum of the two accumulators:

```
private static BigInteger fib_(BigInteger acc1, BigInteger acc2,
                                                BigInteger x) {
  if (x.equals(BigInteger.ZERO)) {
    return BigInteger.ZERO;
  } else if (x.equals(BigInteger.ONE)) {
    return acc1.add(acc2);
```

Eventually, you have to deal with recursion. You must do the following:

- Take accumulator 2 and make it accumulator 1.
- Create a new accumulator 2 by adding the two previous accumulators.
- Subtract 1 from the argument.
- Recursively call the function with the three computed values as its arguments.

Here's the transcription in code:

```
private static BigInteger fib_(BigInteger acc1, BigInteger acc2,
                                                BigInteger x) {
  if (x.equals(BigInteger.ZERO)) {
    return BigInteger.ZERO;
  } else if (x.equals(BigInteger.ONE)) {
    return acc1.add(acc2);
  } else {
    return fib_(acc2, acc1.add(acc2), x.subtract(BigInteger.ONE));
  }
}
```

The last thing to do is to create the main method that calls this helper method with the initial values of the accumulators:

```
public static BigInteger fib(int x) {
  return fib_(BigInteger.ONE, BigInteger.ZERO, BigInteger.valueOf(x));
}
```

This is only one possible implementation. You may organize accumulators, initial values, and conditions in a slightly different manner, as long as it works. Now you can call `fib(5000)`, and it'll give you the result in a couple of nanoseconds. Well, it'll take a few dozen milliseconds, but only because printing to the console is a slow operation. We'll come back to this shortly.

The result is impressive, whether it's the result of the computation (1,045 digits!) or the increase in speed due to the transformation of a dual recursive call into a single one. But you still can't use the method with values higher than 7,500.

EXERCISE 4.2

Turn this method into a stack-safe recursive one.

SOLUTION 4.2

This should be easy. The following code shows the needed changes:

```
BigInteger fib(int x) {
  return fib_(BigInteger.ONE, BigInteger.ZERO,
                              BigInteger.valueOf(x)).eval();
}

TailCall<BigInteger> fib_(BigInteger acc1, BigInteger acc2, BigInteger x) {
  if (x.equals(BigInteger.ZERO)) {
    return ret(BigInteger.ZERO);
  } else if (x.equals(BigInteger.ONE)) {
    return ret(acc1.add(acc2));
  } else {
    return sus(() -> fib_(acc2, acc1.add(acc2), x.subtract(BigInteger.ONE)));
  }
}
```

You may now compute `fib(10000)` and count the digits in the result!

4.2.4 Making the list methods stack-safe and recursive

In the previous chapter, you developed functional methods to work on lists. Some of these methods were naively recursive, so they couldn't be used in production. It's time to fix this.

EXERCISE 4.3

Create a stack-safe recursive version of the `foldLeft` method.

SOLUTION 4.3

The naively recursive version of the `foldLeft` method was tail recursive:

```
public static <T, U> U foldLeft(List<T> ts, U identity,
                                Function<U, Function<T, U>> f) {
  return ts.isEmpty()
      ? identity
      : foldLeft(tail(ts), f.apply(identity).apply(head(ts)), f);
}
```

Turning it into a fully recursive method is easy:

```
public static <T, U> U foldLeft(List<T> ts, U identity,
                                Function<U, Function<T, U>> f) {
  return foldLeft_(ts, identity, f).eval();
}

private static <T, U> TailCall<U> foldLeft_(List<T> ts, U identity,
                                  Function<U, Function<T, U>> f) {
  return ts.isEmpty()
      ? ret(identity)
      : sus(() -> foldLeft_(tail(ts),
                        f.apply(identity).apply(head(ts)), f));
}
```

EXERCISE 4.4

Create a fully recursive version of the recursive `range` method.

HINT

Beware of the direction of list construction (`append` or `prepend`).

SOLUTION 4.4

The `range` method isn't tail recursive:

```
public static List<Integer> range(Integer start, Integer end) {
  return end <= start
      ? list()
      : prepend(start, range(start + 1, end));
}
```

You have to first create a tail recursive version, using an accumulator. Here, you need to return a list, so the accumulator will be a list, and you'll start with an empty list. But you must build the list in reverse order:

```
public static List<Integer> range(List<Integer> acc,
                                  Integer start, Integer end) {
  return end <= start
      ? acc
      : range(append(acc, start), start + 1, end);
}
```

Then you must turn this method into a main method and a helper method by using true recursion:

```
public static List<Integer> range(Integer start, Integer end) {
  return range_(list(), start, end).eval();
}

private static TailCall<List<Integer>> range_(List<Integer> acc,
                                              Integer start, Integer end) {
  return end <= start
      ? ret(acc)
      : sus(() -> range_(append(acc, start), start + 1, end));
}
```

The fact that you had to reverse the operation is important. Can you see why? If not, try the next exercise.

EXERCISE 4.5 (HARD)

Create a stack-safe recursive version of the `foldRight` method.

SOLUTION 4.5

The stack-based recursive version of the `foldRight` method is as follows:

```
public static <T, U> U foldRight(List<T> ts, U identity,
                                 Function<T, Function<U, U>> f) {
  return ts.isEmpty()
      ? identity
      : f.apply(head(ts)).apply(foldRight(tail(ts), identity, f));
}
```

This method isn't tail recursive, so let's first create a tail recursive version. You might end up with this:

```
public static <T, U> U foldRight(U acc, List<T> ts, U identity,
                                 Function<T, Function<U, U>> f) {
  return ts.isEmpty()
      ? acc
      : foldRight(f.apply(head(ts)).apply(acc), tail(ts), identity, f);
}
```

Unfortunately, this doesn't work! Can you see why? If not, test this version and compare the result with the standard version. You can compare the two versions by using the test designed in the previous chapter:

```
public static String addIS(Integer i, String s) {
  return "(" + i + " + " + s + ")";
}

List<Integer> list = list(1, 2, 3, 4, 5);
System.out.println(foldRight(list, "0", x -> y -> addIS(x, y)));
System.out.println(foldRightTail("0", list, "0", x -> y -> addIS(x, y)));
```

You'll get the following result:

```
(1 + (2 + (3 + (4 + (5 + 0)))))
(5 + (4 + (3 + (2 + (1 + 0)))))
```

This shows that the list is processed in reverse order. One easy solution is to reverse the list in the main method before calling the helper method. If you apply this trick while making the method stack-safe and recursive, you'll get this:

```
public static <T, U> U foldRight(List<T> ts, U identity,
                                 Function<T, Function<U, U>> f) {
  return foldRight_(identity, reverse(ts), f).eval();
}

private static <T, U> TailCall<U> foldRight_(U acc, List<T> ts,
                                             Function<T, Function<U, U>> f) {
```

```
  return ts.isEmpty()
      ? ret(acc)
      : sus(() -> foldRight_(f.apply(head(ts)).apply(acc), tail(ts), f));
}
```

In chapter 5, you'll develop the process of reversing the list by implementing `foldLeft` in terms of `foldRight`, and `foldRight` in terms of `foldLeft`. But this shows that the recursive implementation of `foldRight` won't be optimal because `reverse` is an O(n) operation: the time needed to execute it is proportional to the number of elements in the list, because you must traverse the list. By using reverse, you double this time by traversing the list twice. The conclusion is that when considering using `foldRight`, you should do one of the following:

- Not care about performance
- Change the function (if possible) and use `foldLeft`
- Use `foldRight` only with small lists
- Use an imperative implementation

4.3 Composing a huge number of functions

In chapter 2, you saw that you'll overflow the stack if you try to compose a huge number of functions. The reason is the same as for recursion: because composing functions results in methods calling methods.

Having to compose more than 7,000 functions may be something you don't expect to do soon. On the other hand, there's no reason not to make it possible. If it's possible, someone will eventually find something useful to do with it. And if it's not useful, someone will certainly find something fun to do with it.

EXERCISE 4.6

Write a function, `composeAll`, taking as its argument a list of functions from `T` to `T` and returning the result of composing all the functions in the list.

SOLUTION 4.6

To get the result you want, you can use a right fold, taking as its arguments the list of functions, the `identity` function (obtained by a call to the statically imported `Function.identity()` method), and the `compose` method written in chapter 2:

```
static <T> Function<T, T> composeAll(List<Function<T, T>> list) {
  return foldRight(list, identity(), x -> y -> x.compose(y));
}
```

To test this method, you can statically import all the methods from your `CollectionUtilities` class (developed in chapter 3) and write the following:

```
Function<Integer, Integer> add = y -> y + 1;
System.out.println(composeAll(map(range(0, 500), x -> add)).apply(0));
```

If you don't feel comfortable with this kind of code, it's equivalent to, but much more readable than, this:

```
List<Function<Integer, Integer>> list = new ArrayList<>();
for (int i = 0; i < 500; i++) {
  list.add(x -> x + 1);
}

int result = composeAll(list).apply(0);
System.out.println(result);
```

Running this code displays 500, as it's the result of composing 500 functions incrementing their argument by 1. What happens if you replace 500 with 10,000? You'll get a `StackOverflowException`. The reason should be obvious.

By the way, on the machine I used for this test, the program breaks for a list of 2,856 functions.

EXERCISE 4.7

Fix this problem so you can compose an (almost) unlimited number of functions.

SOLUTION 4.7

The solution to this problem is simple. Instead of composing the functions by nesting them, you have to compose their results, always staying at the higher level. This means that between each call to a function, you'll return to the original caller. If this isn't clear, imagine the imperative way to do this:

```
T y = identity;

for (Function<T, T> f : list) {
  y = f.apply(y);
}
```

Here, `identity` means the identity element of the given function. This isn't composing functions, but composing function applications. At the end of the loop, you'll get a `T` and not a `Function<T, T>`. But this is easy to fix. You create a function from `T` to `T`, which has the following implementation:

```
static <T> Function<T, T> composeAll(List<Function<T, T>> list) {
  return x -> {
    T y = x;
    for (Function<T, T> f : list) {
      y = f.apply(y);
    }
    return y;
  };
}
```

A copy of x is made; you can't modify x because it must be effectively final.

You can't use `x` directly, because it would create a closure, so it should be effectively final. That's why you make a copy of it. This code works fine, except for two things.

The first is that it doesn't look functional. This can be fixed easily by using a fold. It can be either a left fold or a right fold:

```
<T> Function<T, T> composeAllViaFoldLeft(List<Function<T, T>> list) {
  return x -> foldLeft(list, x, a -> b -> b.apply(a));
}

<T> Function<T, T> composeAllViaFoldRight(List<Function<T, T>> list) {
  return x -> foldRight(list, x, a -> a::apply);
}
```

You're using a method reference for the `composeAllViaFoldRight` implementation. This is equivalent to the following:

```
<T> Function<T, T> composeAllViaFoldRight(List<Function<T, T>> list) {
  return x -> FoldRight.foldRight(list, x, a -> b -> a.apply(b));
}
```

If you have trouble understanding how it works, think about the analogy with `sum`. When you defined `sum`, the list was a list of integers. The initial value (`x` here) was `0`; `a` and `b` were the two parameters to add; and the addition was defined as `a + b`. Here, the list is a list of functions; the initial value is the identity function; `a` and `b` are functions; and the implementation is defined as `b.apply(a)` or `a.apply(b)`. In the `foldLeft` version, `b` is the function coming from the list, and `a` is the current result. In the `foldRight` version, `a` is the function coming from the list, and `b` is the current result.

To see this in action, refer to the unit tests in the code available from the book's site (https://github.com/fpinjava/fpinjava).

EXERCISE 4.8

The code has two problems, and you fixed only one. Can you see another problem and fix it?

HINT

The second problem isn't visible in the result because the functions you're composing are specific. They are, in fact, a single function from integer to integer. The order in which they're composed is irrelevant. Try to use the `composeAll` method with the following function list:

```
Function<String, String> f1 = x -> "(a" + x + ")";
Function<String, String> f2 = x -> "{b" + x + "}";
Function<String, String> f3 = x -> "[c" + x + "]";
System.out.println(composeAllViaFoldLeft(list(f1, f2, f3)).apply("x"));
System.out.println(composeAllViaFoldRight(list(f1, f2, f3)).apply("x"));
```

SOLUTION 4.8

We've implemented `andThenAll` rather than `composeAll`! To get the correct result, you first have to reverse the list:

```
<T> Function<T, T> composeAllViaFoldLeft(List<Function<T, T>> list) {
  return x -> foldLeft(reverse(list), x, a -> b -> b.apply(a));
}
```

```
<T> Function<T, T> composeAllViaFoldRight(List<Function<T, T>> list) {
  return x -> foldRight(list, x, a -> a::apply);
}

<T> Function<T, T> andThenAllViaFoldLeft(List<Function<T, T>> list) {
  return x -> foldLeft(list, x, a -> b -> b.apply(a));
}

<T> Function<T, T> andThenAllViaFoldRight(List<Function<T, T>> list) {
  return x -> foldRight(reverse(list), x, a -> a::apply);
}
```

4.4 Using memoization

In section 4.2.3, you implemented a function to display a series of Fibonacci numbers. One problem with this implementation of the Fibonacci series is that you want to print the string representing the series up to `f(n)`, which means you have to compute `f(1)`, `f(2)`, and so on, until `f(n)`. But to compute `f(n)`, you have to recursively compute the function for all preceding values. Eventually, to create the series up to `n`, you'll have computed `f(1)` `n` times, `f(2)` `n – 1` times, and so on. The total number of computations will then be the sum of the integers 1 to *n*. Can you do better? Could you possibly keep the computed values in memory so you don't have to compute them again if they're needed several times?

4.4.1 Memoization in imperative programming

In imperative programming, you wouldn't even have this problem, because the obvious way to proceed would be as follows:

```
public static void main(String args[]) {
  System.out.println(fibo(10));
}

public static String fibo(int limit) {
  switch(limit) {
    case 0:
      return "0";
    case 1:
      return "0, 1";
    case 2:
      return "0, 1, 1";
    default:
      BigInteger fibo1 = BigInteger.ONE;
      BigInteger fibo2 = BigInteger.ONE;
      BigInteger fibonacci;
      StringBuilder builder = new StringBuilder("0, 1, 1");
      for (int i = 3; i <= limit; i++) {
        fibonacci = fibo1.add(fibo2);
        builder.append(", ").append(fibonacci);
        fibo1 = fibo2;
        fibo2 = fibonacci;
      }
      return builder.toString();
  }
}
```

Stores f(n – 1) for the next pass

Stores f(n) for the next pass

Although this program concentrates most of the problems that FP is supposed to avoid or to solve, it works and is much more efficient than your functional version. The reason is memoization.

Memoization is a technique that keeps in memory the result of a computation so it can be returned immediately if you have to redo the same computation in the future. Applied to functions, memoization makes the functions memorize the results of previous calls, so they can return the results much faster if they're called again with the same arguments.

This might seem incompatible with functional principles, because a memoized function maintains a state. But it isn't, because the result of the function is the same when it's called with the same argument. (You could even argue that it's more the same, because it isn't computed again!) The side effect of storing the results must not be visible from outside the function.

In imperative programming, this might not even be noticed. Maintaining state is the universal way of computing results, so memoization isn't even noticed.

4.4.2 Memoization in recursive functions

Recursive functions often use memoization implicitly. In your example of the recursive Fibonacci function, you wanted to return the series, so you calculated each number in the series, leading to unnecessary recalculations. A simple solution is to rewrite the function in order to directly return the string representing the series.

EXERCISE 4.9

Write a stack-safe tail recursive function taking an integer `n` as its argument and returning a string representing the values of the Fibonacci numbers from `0` to `n`, separated by a comma and a space.

HINT

One solution is to use `StringBuilder` as the accumulator. `StringBuilder` isn't a functional structure because it's mutable, but this mutation won't be visible from the outside. Another solution is to return a list of numbers and then transform it into a `String`. This solution is easier, because you can abstract the problem of the separators by first returning a list and then writing a function to turn the list into a comma-separated string.

SOLUTION 4.9

The following listing shows the solution using `List` as the accumulator.

Listing 4.5 Recursive Fibonacci with implicit memoization

Calls the fibo_ helper method to get the list of Fibonacci numbers

```
public static String fibo(int number) {
  List<BigInteger> list = fibo_(list(BigInteger.ZERO),
       BigInteger.ONE, BigInteger.ZERO, BigInteger.valueOf(number)).eval();
  return makeString(list, ", ");
}
```

```
private static <T> TailCall<List<BigInteger>> fibo_(List<BigInteger> acc,
                       BigInteger acc1, BigInteger acc2, BigInteger x) {
  return x.equals(BigInteger.ZERO)
      ? ret(acc)
      : x.equals(BigInteger.ONE)
          ? ret(append(acc, acc1.add(acc2)))
          : sus(() -> fibo_(append(acc, acc1.add(acc2)),
                      acc2, acc1.add(acc2), x.subtract(BigInteger.ONE)));
}

public static <T> String makeString(List<T> list, String separator) {
  return list.isEmpty()
      ? ""
      : tail(list).isEmpty()
          ? head(list).toString()
          : head(list) + foldLeft(tail(list), "",
                                          x -> y -> x + separator + y);
}
```

Formats the list into a comma-separated string through a call to the makeString method

RECURSION OR CORECURSION?

This example demonstrates the use of implicit memoization. Don't conclude that this is the best way to solve the problem. Many problems are much easier to solve when twisted. So let's twist this one.

Instead of a suite of numbers, you could see the Fibonacci series as a suite of pairs (tuples). Instead of trying to generate this,

```
0, 1, 1, 2, 3, 5, 8, 13, 21, ...
```

you could try to produce this:

```
(0, 1), (1, 1), (1, 2), (2, 3), (3, 5), (5, 8), (8, 13), (13, 21), ...
```

In this series, each tuple can be constructed from the previous one. The second element of tuple `n` becomes the first element of tuple `n + 1`. The second element of tuple `n + 1` is equal to the sum of the two elements of tuple `n`. In Java, you can write a function for this:

```
x -> new Tuple<>(x._2, x._1.add(x._2));
```

You can now replace the recursive method with a corecursive one:

```
public static String fiboCorecursive(int number) {
  Tuple<BigInteger, BigInteger> seed =
                          new Tuple<>(BigInteger.ZERO, BigInteger.ONE);
  Function<Tuple<BigInteger, BigInteger>,Tuple<BigInteger, BigInteger>> f =
                                    x -> new Tuple<>(x._2, x._1.add(x._2));
  List<BigInteger> list = map(List.iterate(seed, f, number + 1), x -> x._1);
  return makeString(list, ", ");
}
```

The `iterate` method takes a seed, a function, and a number *n*, and creates a list of length *n* by applying the function to each element to compute the next one. Here's its signature:

```
public static <B> List<B> iterate(B seed, Function<B, B> f, int n)
```

This method is available in the `fpinjava-common` module.

4.4.3 Automatic memoization

Memoization isn't mainly used for recursive functions. It can be used to speed up any function. Think about how you perform multiplication. If you need to multiply 234 by 686, you'll probably need a pen and some paper, or a calculator. But if you're asked to multiply 9 by 7, you can answer immediately, without doing any computation. This is because you use a memoized multiplication. A memoized function works the same way, although it needs to make the computation only once to retain the result.

Imagine you have a functional method `doubleValue` that multiplies its argument by 2:

```
Integer doubleValue(Integer x) {
  return x * 2;
}
```

You could memoize this method by storing the result into a map:

```
Map<Integer, Integer> cache = new ConcurrentHashMap<>();
Integer doubleValue(Integer x) {
  if (cache.containsKey(x)) {
    return cache.get(x);
  } else {
    Integer result = x * 2;
    cache.put(x, result) ;
    return result;
  }
}
```

Map is used to store the results

Looks in the map to see if the result has already been computed

If found, returns the result

If not found, computes the result

Puts the result in the map

Returns the result

In Java 8, this can be made much shorter:

```
Map<Integer, Integer> cache = new ConcurrentHashMap<>();

Integer doubleValue(Integer x) {
  return cache.computeIfAbsent(x, y -> y * 2);
}
```

If you prefer using functions (which is likely, given the subject of this book), you can apply the same principle:

```
Function<Integer, Integer> doubleValue =
                      x -> cache.computeIfAbsent(x, y -> y * 2);
```

But two problems arise:

- You have to repeat this modification for all functions you want to memoize.
- The map you use is exposed to the outside.

The second problem is easy to address. You can put the method or the function in a separate class, including the map, with private access. Here's an example in the case of a method:

```
public class Doubler {

  private static Map<Integer, Integer> cache = new ConcurrentHashMap<>();

  public static Integer doubleValue(Integer x) {
    return cache.computeIfAbsent(x, y -> y * 2);
  }
}
```

You can then instantiate that class and use it each time you want to compute a value:

```
Integer y = Doubler.doubleValue(x);
```

With this solution, the map is no longer accessible from the outside. You can't do the same for functions, because functions can't have static members. One possibility is to pass the map to the function as an additional argument. This can be done through a closure:

```
class Doubler {
  private static Map<Integer, Integer> cache = new ConcurrentHashMap<>();

  public static Function<Integer, Integer> doubleValue =
                                x -> cache.computeIfAbsent(x, y -> y * 2);
}
```

You can use this function as follows:

```
Integer y = Doubler.doubleValue.apply(x);
```

This gives no advantage compared to the method solution. But you can also use this function in more idiomatic examples, such as this:

```
map(range(1, 10), Doubler.doubleValue);
```

This is equivalent to using the method version with the following syntax:

```
map(range(1, 10), Doubler::doubleValue);
```

THE REQUIREMENTS

What you need is a way to do the following:

```
Function<Integer, Integer> f = x -> x * 2;
Function<Integer, Integer> g = Memoizer.memoize(f);
```

Then you can use the memoized function as a drop-in replacement for the original one. All values returned by function `g` will be calculated through the original function `f` the first time, and returned from the cache for all subsequent accesses. By contrast, if you create a third function,

```
Function<Integer, Integer> f = x -> x * 2;
Function<Integer, Integer> g = Memoizer.memoize(f);
Function<Integer, Integer> h = Memoizer.memoize(f);
```

the values cached by `g` won't be returned by `h`; `g` and `h` will use separate caches.

IMPLEMENTATION

The `Memoizer` class is simple and is shown in the following listing.

Listing 4.6 The `Memoizer` class

```
public class Memoizer<T, U> {

  private final Map<T, U> cache = new ConcurrentHashMap<>();

  private Memoizer() {}

  public static <T, U> Function<T, U> memoize(Function<T, U> function) {
    return new Memoizer<T, U>().doMemoize(function);
  }

  private Function<T, U> doMemoize(Function<T, U> function) {
    return input -> cache.computeIfAbsent(input, function::apply);
  }
}
```

The memoized method returns a memoized version of its function argument.

The doMemoize method handles the computation, calling the original function if necessary.

The following listing shows how this class can be used. The program simulates a long computation to show the result of memoizing the function.

Listing 4.7 Demonstrating the memoizer

```
private static Integer longCalculation(Integer x) {
    try {
      Thread.sleep(1_000);
    } catch (InterruptedException ignored) {}
    return x * 2;
  }

  private static Function<Integer, Integer> f =
                                  MemoizerDemo::longCalculation;

  private static Function<Integer, Integer> g = Memoizer.memoize(f);

  public static void automaticMemoizationExample() {
    long startTime = System.currentTimeMillis();
    Integer result1 = g.apply(1);
    long time1 = System.currentTimeMillis() - startTime;
    startTime = System.currentTimeMillis();
```

Simulates a long computation

The function to memoize

The memoized function

```
    Integer result2 = g.apply(1);
    long time2 = System.currentTimeMillis() - startTime;
    System.out.println(result1);
    System.out.println(result2);
    System.out.println(time1);
    System.out.println(time2);
  }
```

Running the `automaticMemoizationExample` method on my computer produces the following result:

```
2
2
1000
0
```

Note that the exact result will depend on the speed of your computer.

You can now make memoized functions out of ordinary ones by calling a single method, but to use this technique in production, you'd have to handle potential memory problems. This code is acceptable if the number of possible inputs is low, so you can keep all results in memory without causing memory overflow. Otherwise, you can use soft references or weak references to store memoized values.

MEMOIZATION OF "MULTIARGUMENT" FUNCTIONS

As I said before, there's no such thing in this world as a function with several arguments. Functions are applications of one set (the source set) to another set (the target set). They can't have several arguments. Functions that appear to have several arguments are one of these:

- Functions of tuples
- Functions returning functions returning functions ... returning a result

In either case, you're concerned only with functions of one argument, so you can easily use your `Memoizer` class.

Using functions of tuples is probably the simplest choice. You could use the `Tuple` class written in previous chapters, but to store tuples in maps, you'd have to implement `equals` and `hashcode`. Besides this, you'd have to define tuples for two elements (pairs), tuples for three elements, and so on. Who knows where to stop?

The second option is much easier. You have to use the curried version of the functions, as you did in previous chapters. Memoizing curried functions is easy, although you can't use the same simple form as previously. You have to memoize each function:

```
Function<Integer, Function<Integer, Integer>> mhc =
                                  Memoizer.memoize(x ->
                                          Memoizer.memoize(y -> x + y));
```

You can use the same technique to memoize a function of three arguments:

```
Function<Integer, Function<Integer, Function<Integer, Integer>>> f3 =
                                                  x -> y -> z -> x + y - z;
```

```
Function<Integer, Function<Integer, Function<Integer, Integer>>> f3m =
                  Memoizer.memoize(x ->
                          Memoizer.memoize(y ->
                                  Memoizer.memoize(z -> x + y - z));
```

The following listing shows an example of using this memoized function of three arguments.

Listing 4.8 Testing a memoized function of three arguments for performance

```
Function<Integer, Function<Integer, Function<Integer, Integer>>> f3m =
      Memoizer.memoize(x ->
              Memoizer.memoize(y ->
                      Memoizer.memoize(z ->
        longCalculation(x) + longCalculation(y) - longCalculation(z))));

  public void automaticMemoizationExample2() {
    long startTime = System.currentTimeMillis();
    Integer result1 = f3m.apply(2).apply(3).apply(4);
    long time1 = System.currentTimeMillis() - startTime;
    startTime = System.currentTimeMillis();
    Integer result2 = f3m.apply(2).apply(3).apply(4);
    long time2 = System.currentTimeMillis() - startTime;
    System.out.println(result1);
    System.out.println(result2);
    System.out.println(time1);
    System.out.println(time2);
  }
```

This program produces the following output:

```
2
2
3002
0
```

This shows that the first access to the `longCalculation` method has taken 3,000 milliseconds, and the second has returned immediately.

On the other hand, using a function of a tuple may seem easier after you have the `Tuple` class defined. The following listing shows an example of `Tuple3`.

Listing 4.9 An implementation of `Tuple3`

```
public class Tuple3<T, U, V> {

  public final T _1;
  public final U _2;
  public final V _3;

  public Tuple3(T t, U u, V v) {
    _1 = Objects.requireNonNull(t);
    _2 = Objects.requireNonNull(u);
    _3 = Objects.requireNonNull(v);
  }
```

```
  @Override
  public boolean equals(Object o) {
    if (!(o instanceof Tuple3)) return false;
    else {
      Tuple3 that = (Tuple3) o;
      return _1.equals(that._1) && _2.equals(that._2)
                                        && _3.equals(that._3);
    }
  }

  @Override
  public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + _1.hashCode();
    result = prime * result + _2.hashCode();
    result = prime * result + _3.hashCode();
    return result;
  }
}
```

The following listing shows an example of testing a memoized function taking `Tuple3` as its argument.

Listing 4.10 A memoized function of `Tuple3`

```
Function<Tuple3<Integer, Integer, Integer>, Integer> ft =
                    x -> longCalculation(x._1)
                                  + longCalculation(x._2)
                                            - longCalculation(x._3);
Function<Tuple3<Integer, Integer, Integer>, Integer> ftm =
                                              Memoizer.memoize(ft);

public void automaticMemoizationExample3() {
  long startTime = System.currentTimeMillis();
  Integer result1 = ftm.apply(new Tuple3<>(2, 3, 4));
  long time1 = System.currentTimeMillis() - startTime;
  startTime = System.currentTimeMillis();
  Integer result2 = ftm.apply(new Tuple3<>(2, 3, 4));
  long time2 = System.currentTimeMillis() - startTime;
  System.out.println(result1);
  System.out.println(result2);
  System.out.println(time1);
  System.out.println(time2);
}
```

ARE MEMOIZED FUNCTIONS PURE?

Memoizing is about maintaining state between function calls. A memoized function is a function whose behavior is dependent on the current state. But it'll always return the same value for the same argument. Only the time needed to return the value will be different. So the memoized function is still a pure function if the original function is pure.

A variation in time may be a problem. A function like the original Fibonacci function needing many years to complete may be called *nonterminating*, so an increase in time may create a problem. On the other hand, making a function faster shouldn't be a problem. If it is, there's a much bigger problem somewhere else!

4.5 Summary

- A recursive function is a function that's defined by referencing itself.
- In Java, recursive methods push the current computation state onto the stack before recursively calling themselves.
- The Java default stack size is limited. It can be configured to a larger size, but this generally wastes space because all threads use the same stack size.
- Tail recursive functions are functions in which the recursive call is in the last (tail) position.
- In some languages, recursive functions are optimized using tail call elimination (TCE).
- Java doesn't implement TCE, but it's possible to emulate it.
- Lambdas may be made recursive.
- Memoization allows functions to remember their computed values in order to speed up later accesses.
- Memoization can be made automatic.