Functional Style

1. Imperative vs. Functional Style

Imperative Style

In an imperative style, you express your code in an imperative nature:

- You tell it what to do
- Step by Step
- **Mutability** - create variables and continuously modify them as you progress through your computations. In the imperative approach, code is written with specific steps needed to accomplish the goal of the program. It is *algorithmic*. What matters are:
  - The objects/variables that you create.
  - The order of execution
  - Flow controlled through loops, conditionals and method calls.
  - Class structure and instances of classes.
    Source

An example of an imperative style:
Java

```java
import java.util.ArrayList;
import java.util.List;

public class Sample {
        public static void main(String[] args) {
                List<Integer> values = new ArrayList<Integer>();
                values.add(1);
                values.add(2);
                values.add(3);
                System.out.println(totalValues(values));
        }
        private static int totalValues(List<Integer> values){
                int total = 0;
                for(int value : values) {
                        total += value;
                }
                return total;
```

```
        }
}
```

Notice that we create a variable named `total` . We then continuously modifying its value over and over.

Also notice the **external iterator**. You are iterating one element at a time, continuously going to the next variable.

What is Functional Style?

1. Functions are first class citizens.
2. Functions have a higher order than variables.
3. Functions don't have any side effect.

You are able to decompose your application using functions not just objects.

> **NOTE:** In an imperative style, your application is broken down into objects and classes. In an OOP language, we often focus on creating objects, encapsulating data and behavior.

****Higher Order of Functions**
In a functional approach, functions are valuable form of abstraction, you can also create functions from functions, and you can pass functions to functions.

****Immutability**
Functions promote and maintain immutability.

- You don't modify anything, you change them.

> **Example of a functional approach that's supported by immutability: Bank Balance**
> Say you access your bank to check your balance at a given time. A value can be returned from a function that retrieves your account balance at that time. That particular balance at that particular instance is a value that can't be changed again.

You aren't modifying existing variables, you will be **recreating** a different value.

No side effects
The function is not affected by anything outside, and not affecting anything inside. If you are passing the exact same input into the function, you will get exactly the same output from this function.

- As long as the input is the same, the output will be exactly the same.
  Referential transparency - take and reorder sub-computations.
  for example: addition is commutative, `a + b + c + d` could be reordered as `t1 = a + b`, `t2 = c + d` which equals `t1 + t2`, `t3 = c + d`, `t4 = a + b`, `t3 + t4` will be the same.
  Not only can you use **referential transparency** to reorder sub-computations but you can also use multi-threading and concurrency with this characteristic of functional styling.

2. Total Using Inject

Let's look back at the imperative code we wrote above. Now let's try to rewrite this code but this time we total without modifying any value.

We want to total without using mutability in Java.

```
import java.util.ArrayList;
import java.util.List;

public class Sample {
        public static void main(String[] args) {
                List<Integer> values = new ArrayList<Integer>();
                values.add(1);
                values.add(2);
                values.add(3);
                System.out.println(totalValues(values, values.size(), 0));
        }
        private static int totalValuesImmutable(List<Integer> values, int size, int
currentTotal){
                if(size == 0)
                        return currentTotal
                //get the last element of the list and total it
                return totalValuesImmutable(values, size-1, currentTotal +
values.get(size-1)
        }
}
```

We can take advantage of a stack to create a new value and push it to the top of a stack.

- For this we can either use a stack data structure or use a stack from a function call (recursion).
  We continuously create a new size value on each function call and recursively return the values as you receive them.

Not a single variable is modified in `totalValuesImmutable`. We do decrement size but we haven't set it to any variable, we just passed it along as a variable on the stack. Then we call

the function.

- This function has no side effects.
- It doesn't modify any data given to it.

You don't need a functional language to write functional styled code. It's more of a paradigm.

Our example in Ruby

Not Ruby is not a functional language but Ruby does have a functional style of programming in it.

```
values = [1, 2, 3, 4, 5, 6]

puts values.count

values.each { |e| puts e }
```

What's a function?

- Has a name
- Has a body
- Has parameter list
- Has return type
  Now which of these four are the most important for a function

- The function's body

Now let's look back up to our code above.

1. **Function's Name**: We have an anonymous function (no name)
2. **Function's Body**: the body of the function is `puts e`
3. **Function's Parameters**: the parameter list of the function is `|e|`
4. **Function's Return Type**: Ruby is a dynamic language so the return type is not specified.

The `each` function is an example of an ***Internal iterator***, you simply specify a body of code that is executed within the context of the elements of the collection.

Internal Iterators

Let's take a look at different internal iterators in Ruby.
To start off lets try to double each value in the collection using a functional style.

> NOTE: In Ruby everything is a an expression

Doubling each value

```
values = [1, 2, 3, 4, 5, 6]

values.map { |e| e * 2 }
```

Once again, the function passed to map is called in the context of each element in the collection. And in this case, for each element in the collection `e` is doubled.

A key difference here that is happening is that each doubled number is dumped into the `values.map` and returned back to us.

Finding Elements

```
values = [1, 2, 3, 4, 5, 6]

puts values.find { |e| e % 2 == 0 }

puts values.find_all { |e| e % 2 == 0 }
```

If the expression `e % 2 == 0` is true then you collect that value, otherwise you ignore it.

- Notice how this differs from `.map` that collects the results for all, while `.find_all` collects each element that satisfies the condition.

How About Comma Separated Prints
One form of iteration:

```
values = [1, 2, 3, 4, 5, 6]

result = values.find_all { |e| e % 2 == 0 }

# NOTE: usual for loops aren't really used in Ruby as often
for i in 0..values.count-1
        print "#{result[i]}, "
        print ', ' if i result.count-1
        print ' '
end
```

Notice how Ruby allows you to add an if statement at the end of an expression.

- The if statement is evaluated first then if it's true it evaluates the left side.
  It would be nice if you didn't have to evaluate that if statement at the end.

Using the .join function

```
values = [1, 2, 3, 4, 5, 6]

result = values.find_all { |e| e % 2 == 0 }
puts result.join(', ')
```

You may even write this on one line.

```
values = [1, 2, 3, 4, 5, 6]

result = values.find_all { |e| e % 2 == 0 }.join(', ')
```

Notice how much more clean, and intuitive that this function style is.

```
total = 0
values.each { |e| total += e } # can't be used until total is bound
```

`{ |e| total += e }` is a block of code called from `.each`
`e` is bound to parameter but `total` is bound to a variable in the scope of the caller of the function your block is called from.

More importantly, it is bound to the scope where the block is being defined.

The block has a few levels:

1. Level 1 is where total is defined.
2. Level 2 is inside each method.
3. Level 3 is inside the block.

From block level 3, `total` is bound to variable in level 1

So this block of code `{ |e| total += e }` is not able to be used until all the varibales are bounded.
But in order to bound `total` it closes over the scope of the definition of the block itself, because it closes over the scope it's called a **closure**.

Function Values vs. Closures

`{|e| e * 2 }` This is simply a block or a function value.

If you consider `{ |e| e * factor }` this is a closure because factor has to be bound or closed over to a variable outside in scope.

The difference between a function value and a closure is this:

- Function values have all bounded variables.
- Closures have unbounded variables.

> Without automatic garbage collection it would be difficult to program functional style with **immutability**!
> If you code this in a language like C++, where you have to delete everything that you don't want to use, the code becomes verbose.

Using the .inject function

```
puts values.inject(0) { |carryOver, e| carryOver + e }
```

1. carryOver is bound to 0 and e is bound to 1. The function returns 1 (carryOver + e)
2. inject now calls the function again, this time with carryOver bound to 1 and e bound to 2. The function returns 3 (carryOver + e)
3. inject now calls the function again, this time with carryOver bound to 3 and e bound to 3. The function returns 6 (carryOver + e)

So far we've look at writing in a functional style, yet not even in a functional language. Ruby is a dynamic language yet we're still able to use a functional style within it.

3. Passing functions in Groovy

Groovy

Groovy is a dynamic language built of the Java Virtual Machine (JVM), it has features of Ruby with a strong integration of Java.

- Same semantic model as Java
- Syntax is similar to Java

Take a look at the code below...

```
def totalValues(values){
        def total = 0
        values.each { total += it } //internal iterator, it refers to the param
```

```
        total // return is an optional keyword, so this line is returning total
}

def totalEvenValues(values) {
        def total = 0
        values.each {
                if (it % 2 == 0) total += it
        }
        total
}

def totalOddValues(values) {
        def total = 0
        values.each {
                if (it % 2 != 0) total += it
        }
        total
}

def totalValuesGreaterThanSix(values) {
        def total = 0
        values.each {
                if (it > 6) total += it
        }
        total
}


def listOfValues = [1, 2, 3, 6, 4, 7, 8]
println totalValues(listOfValues)
println totalEvenValues(listOfValues)
println totalOddValues(listOfValues)
println totalValuesGreaterThanSix(listOfValues)
```

Quite verbose right?

Notice how you've written almost duplicated code. Every single function here is the same format, the only differences between them are the conditions of the if blocks.

So let's change that code so that it's DRY

```
def totalSelectedValues(values, selector){
        def total = 0
        values.each {
                if (selector(it)) total += it
```

```
        }
        total
}
def listOfValues = [1, 2, 3, 6, 4, 7, 8]

println totalSelectedValues(listOfValues) { true } //send the closure on the
outside, doesn't have to be explicitly typed as a parameter

// the function becomes way more modular
println totalSelectedValues(listOfValues) { it % 2 == 0 }
println totalSelectedValues(listOfValues) { it % 2 != 0 }
println totalSelectedValues(listOfValues) { it > 6 }
```

Notice how easy it becomes to write code where functions can take functions as parameters.

4. Passing functions in Scala

Scala is a statically type language built on the JVM.

```
def totalSelectedValues(valuesL List[int], selector: Int => Boolean) = {
        var total = 0
        for(e <- values) {
                if (selector(e)) total += e
        }
        total
}

val listOfValues = List(1, 2, 4, 5, 6, 7, 8)
println(totalSelectedValues(listOfValues, { () => true }))
```

> `selector` is not an object reference, it's a reference to a function value. *What's a function value you ask?* A function is a mapping of values (input and output), an input is transformed and mapped to an output which is `Int => Boolean` represents.

Let's quickly refactor this code to use an internal iterator

```
def totalSelectedValues(valuesL List[int], selector: Int => Boolean) = {
        var total = 0
        values.foreach { e =>
                if(selector(e) total += e)
        }
        total
}
```

```
val listOfValues = List(1, 2, 4, 5, 6, 7, 8)
println(totalSelectedValues(listOfValues, { () => true }))
```

We are now using an internal iterator `foreach` which is similar to what we saw earlier.

Let's skip back down to where we're invoking our function.

```
println(totalSelectedValues(listOfValues, { e => true }))
```

Remember how in Groovy we could make things easier to read by moving our second parameter outside of the function. Unfortunately **this doesn't work in Scala... Why?**

> Scala as a statically typed language wants to enforce that two parameters are sent to the function. You cannot just provide one.

You could make this code work, but with some extra steps. Let's deviate from this code with another example:

```
foo(int a) // This function takes one parameter (int)
foo(int a, double b) // This function takes two parameters (int double)

foo() // This function takes no parameters

foo(int a, double b)(int c, char d) // This function is taking two parameter lists:
one is (int, double) another is (int, char)
```

Scala says if you're going to attach a function value outside of the parentheses then send it as a separate parameter list.

```
def totalSelectedValues(values: List[Int])(valuesL List[Int],
        selector: Int => Boolean) = {
                var total = 0
                values.foreach { e =>
                        if(selector(e) total += e)
                }
                total
}

val listOfValues = List(1, 2, 4, 5, 6, 7, 8)
println(totalSelectedValues(listOfValues) { e => true } )
```

So that's an example of writing functions that accept functions as parameters.

Not only does Scala and Groovy allow you to pass anonymous code blocks or function values, you can also assign them to variables as well.

```
def totalSelectedValues(values: List[Int])(valuesL List[Int],
        selector: Int => Boolean) = {
                var total = 0
                values.foreach { e =>
                        if(selector(e) total += e)
                }
                total
}

val listOfValues = List(1, 2, 4, 5, 6, 7, 8)
println(totalSelectedValues(listOfValues) { e => true } )

def checkEven(value : Int) = {
        value % 2 == 0
}

// There's a stark difference between these two function calls below!
println(totalSelectedValues(listOfValues) { checkEvent } )
println(totalSelectedValues(listOfValues)(checkEven))
```

What's the difference between lines 17 and 18?

```
println(totalSelectedValues(listOfValues) { checkEvent } )
println(totalSelectedValues(listOfValues)(checkEven))
```

In our first function call, the function `checkEven` is invoked within `totalSelectedValues` and the values of `checkEven` are then sent into the function.
In second function call, `checkEven` is sent as a pointer to this function to `totalSelectedValues`.

5. Mutability vs. Immutability

A pure functional language promotes immutability.

- Clear distinction between **initialization** and assignment.
  - Once you are past the initialization phase you are not allowed to modify these values, variables or objects.
- You cannot modify period.
  - All objects, data structures, must take the approach of immutability.

"A pure functional language requires learning how to deal with immutability"

6. Immutability in Scala

There are two types of declaration in Scala `var` and `val`.

- A `var` is a variable (mutable).
- A `val` is a constant value (immutable).

```
var buff1 = new StringBuilder
buff1.append("hello")
println(buff1)

buff1 = new StringBuilder
buffl1.append("there")
println(buff1)
```

In this code we have mutability at two points here as we modify the `buff1`.

1. The `StringBuilder` itself is a mutable data structure.
2. Then we also have `buff1` which is a mutable reference.

We modified the `StringBuilder`, notice how when we print `bufff1`, its output is first `hello` then it's changed to `there`.

- On line 2 we mutated the `StringBuilder`.
- On line 6 we mutated the `StringBuilder`.
- On line 5, we mutated the reference.

```
var buff1 = new StringBuilder
buff1.append("hello")
println(buff1)

buff1 = new StringBuilder
buffl1.append("there")
println(buff1)

val buff2 = new StringBuilder
buff2.append("ok")
println(buff2)
```

Notice how on line number 10 we still mutated the `StringBuilder`.
In this case `StringBuilder` will always be mutable. However, `buff2` will always be immutable unlike `buff1`.

- You can change the object you are referring but you cannot change which object you are referring to.

The following additional code will cause a compilation error.

```
val buff2 = new StringBuilder
buff2.append("ok")
println(buff2)

val buff2 = new StringBuilder
buff2.append("ok?")
println(buff2)
```

Why? Because, `buff2` is immutable and you cannot change the `buff2`.

> `var` is like a variable in Java and C#
> `val` is like `final` in Java and readonly in C# (or is it like const in C#?)

```
var greet1 = hello
println(greet1)
println(greet1.getClass())

greet1 = "howdy"
println(greet1)
```

Keep in mind that we did not modify the string, we rather took the `greet1` reference and made it now point to a completely different object in memory.

On the other hand if we create a `val`

```
val greet2 = "hello"
```

Now `greet2` will only point to this object. If we were to reassign it to another object it would result in a compilation error.

> `val` is like `const*`` in C++

7. Immutability in Erlang

Say we want to double various values from a list.

```
main (__) ->
        Values = [1,2,3,4,5,6],
        Doubled = lists:map(fun(E) -> E * 2 end, Values)
        io:format("~p", [Doubled).
```

`map` is a function that takes two parameters. The first value is a function, the second is a list of values.

- We pass an anonymous function `fun` that takes a parameter `e` that we double and return.

> Using this anonymous function is very similar to what you can do in JavaScript.

Notice that on lines 2 and 3 that we only using initialization not assignments.

```
main(__) ->
        X = 1,
        io:format("~p~n"), [X]),
        Y = 4,
        io:format("~p~n"), [Y]),
        Y = X + 3,
        io:format("~p~n", [Y]),
```

Erlang doesn't allow mutation. The equals sign (`=`) in Erlang **does not mean assignment**. It doesn't even mean initialization.

In Erlang, the equals sign means **pattern matching**.

- It is thus called the **match operator**.

> In a pattern matching, a left-hand side pattern is matched against a right-hand side term. If the matching succeeds, any unbound variables in the pattern become bound. If the matching fails, a run-time error occurs.

If we were to change line 8 to `Y = X + 2,` the program would break. Why?

- This is because the compiler is checking if the left will equal the same as the right. And because 4 does not equal 5, it is not a match.

Another Pattern Matching Example:

```
main(__) ->
{Water, "vapor"} = {"coldwater", "vapor"},
```

```
    io:format{"~p", [Water]},
    {Water, Steam} = {"coldwater", "steam"}
    io:format("~p", [Steam]),
```

In this example you can also see an unbounded variable becoming bound to an initial value by using the match operator (`=`).

So in Erlang you cannot change a variable after you have initialized it. **You bind variables to their values**.