Procedures

1. What is a Procedure?

We have all used procedures. They are some of the most fundamental building blocks of a program. A **Procedure** is made up of expressions, think of them as a composite of expressions.

Procedures also honor **abstraction** and to a lesser extent encapsulation.

- The purpose of a procedure is to provide a service (complete a task).
- Provide **abstraction**

2. How Do You Evaluate a Procedure?

REPL

Read-Evaluate-Print-Loop

1. **Read** the user input
2. **Evaluate** your code (to work out what you mean)
3. **Print** any results (see the computer's response)
4. **Loop** back to step 1 (continue conversation)

REPL is incredibly valuable because you don't have to go through a formal process to quickly evaluate expressions.

- Easy to evaluate expressions.
  Serving two purposes:
- Helps to learn a language quickly.
- Helps to experiment and build code.

> The computer tells you it’s waiting for instructions by presenting you with either three chevrons (`>>>`) or a numbered prompt (`In [1]:`). You just type your commands and hit return for the computer to evaluate them.
>
> Programmers use the REPL when they need to “work stuff out”. It’s a bit like a jotter where you “rough out” ideas and explore problems. Because of the instant feedback you get from a REPL it makes it easy to improvise, nose around and delve into what the computer is doing.

REPL works in many different Languages

Ruby

Ruby comes with an interactive shell called **irb**

```
> irb
>> max = 100
=> 100
>> puts max
100
>> puts max.class
Fixnum
=> nil
>> max - 10
=> 90
>> greet = "hello"
=> "hello"
>> greet.class
=> String
```

Groovy

Groovy also comes with an interactive shell for REPL, use the command `groovysh`

```
groovy:000> greet = "hello"
===> hello
groovy:000> greet.toUpperCase()
===> HELLO
```

You can experiment even more with Groovy. You can write a `metaClass`.

```
groovy:000> String.metaClass.shout = { -> delegate.toUpperCase() }
===> groovysh_evaluate$_run_closure1@6ce7ce4c
groovy:000> greet.shout()
===> HELLO
groovy:000> String.metaClass.wisper = {
groovy:001> -> delegate.toLowerCase()
groovy:002> }
===> groovysh_evaluate$_run_closure1@3d057305
groovy:000> greet2 = greet.shout()
===> HELLO
groovy:000> greet2.wisper()
```

```
===> hello
groovy:000>
```

Scala

Scala also has a REPL, just type the command `scala`.

```
scala> var greet =  "hello"
greet: java.lang.String = hello

scala> println(greet)
hello

scala> 2 * 4
res1: Int = 8

scala> def foo(value: Int) = { value * 3 }

scala> def foo2(value: Int) = { value * 3 }
foo2: (value: Int)Unit

scala> def foo3(value: Int) { value == 3 }
foo3: (value: Int)Unit

scala> def foo3(value: Int) = { value == 3 }
foo3: (value: Int)Boolean

scala> prinln(greet1)
hello there

scala> greet1 = "bye"
<console>:6: error: reassignment to val
        greet1 = "bye"
                   ^
scala> val greet1 = "hello"
greet1: java.lang.String = hello

scala> println(greet1)
hello
```

Expressions

- Expressions are evaluated and produce a result.
- A number is a primitive expression

- Combinations: You can use operators to create other expressions: 4 + 2 or in LISP (+ 4 2)

> Primitives are expressions by themselves.
> **e.g.** `"Hello"` is an expression. So is `2`, so `6` and so on.

Variables

Variables are **identified by a name** and **store values**. They are defined and assigned values.

Additional Context: Literals vs Variables

Variables are a way of referring to data, while literals are types of data.

> Literals are a synthetic representation of boolean, char, numeric, or string data. They are values that are obvious when we write them. e.g. `123` is an integer literal.

There are two reasons why a variable name are useful

1. A literal lacks abstraction, it's hard to know what the value represents.
2. You don't know if the literal you typed on one line is related to another.
    - You thus can maintain the code easier.

Notice how we initialize `max` to `1000` and then we were able to reassign the value of `max` to `2000`.

```
var max = 1000

println(max)

max = 2000
println(max)
```

A lot of times this may not be possible. Initializing variables is allowed in **all** languages. However, **reassignment** is **not** allowed in **purely functional languages**.

> Scala as a hybrid language allows for reassignment. Purely functional languages such as Erlang will not allow for assignment.

Back to Procedures...

Recall that we said earlier that procedures are **composed of expressions**.

- **Compound Procedures** - Identified by a name and represents an operation
- **Applying a function** - you evaluate a procedure by sending arguments for the parameters.

```
def doubleValue(value: Int) = value * 2

println(doubleValue(4))
```

Above you can see that we are "applying the function" by invoking it.

3. Order of Evaluation

Let's take a look at how we apply a function, and how our parameters are evaluated.

```
def doubleValue(value: Int) = value * 2

println(doubleValue(2 + 2))
```

The output here is the same as when we passed a 4 as an argument. Now the question here is, when did it evaluate the `2 + 2`

Applicative Order - Operators and parameters evaluated before procedure is valuated.
Normal Order - Operators and parameters are evaluated only after the procedure is fully expanded.

- Call happens first.

Applicative Order

Applicative order is **efficient** and can remove duplicate computations.

```
def square(x) = x * x
```

```
square(2 + 3) = square(5) = 5 * 5 = 25
```

Normal Order

```
def square(x) = x * x
```

```
square(2 + 3) = square(5) = (2 + 3) * (2 + 3) = 25
```

Result is the same as an applicative order, but **not efficient**. However lazy evaluation can come in handy at times. May eliminate the need for some computations or may push it just in time.

Decomposing Procedure

It is easier if we decompose procedures into smaller procedures.

- Easy to comprehend
- Easy to explain
- Easy to express and maintain.

> We all know that writing long functions is bad, that long functions are hard to understand, long functions are hard to maintain, long functions are hard to test, and long functions are hard to explain.

From a design point of view it is quite easy to decompose procedures. This helps a great deal from the viewpoint of **abstraction** and **reusability**.

- Rather than writing something lengthy you want to write them as calling other procedures.

Example: Say you write an algorithm and you want to implement that algorithm in code. So you write a procedure for it. Say you have three loops and inside each loop you can perform an addition of a matrix.

```
function foo():
        for i in arr:
                perform addition
                for j in arr:
                        perform addition
                        for k in arr:
                                perform addition
```

Notice if we do the addition in this function, it wouldn't stand out clearly. But if we decompose the addition, we could simply call a `addMatrix()` function/procedure. It then becomes easier to express what you are doing.

```
function foo():
        for i in arr:
                addMatrix()
                for j in arr:
                        addMatrix()
                        for k in arr:
                                addMatrix()
```

An important trait in languages is that they should not only allow you to create procedures but also they should allow you to decompose procedures. You also want a language that protects abstraction and encapsulation.

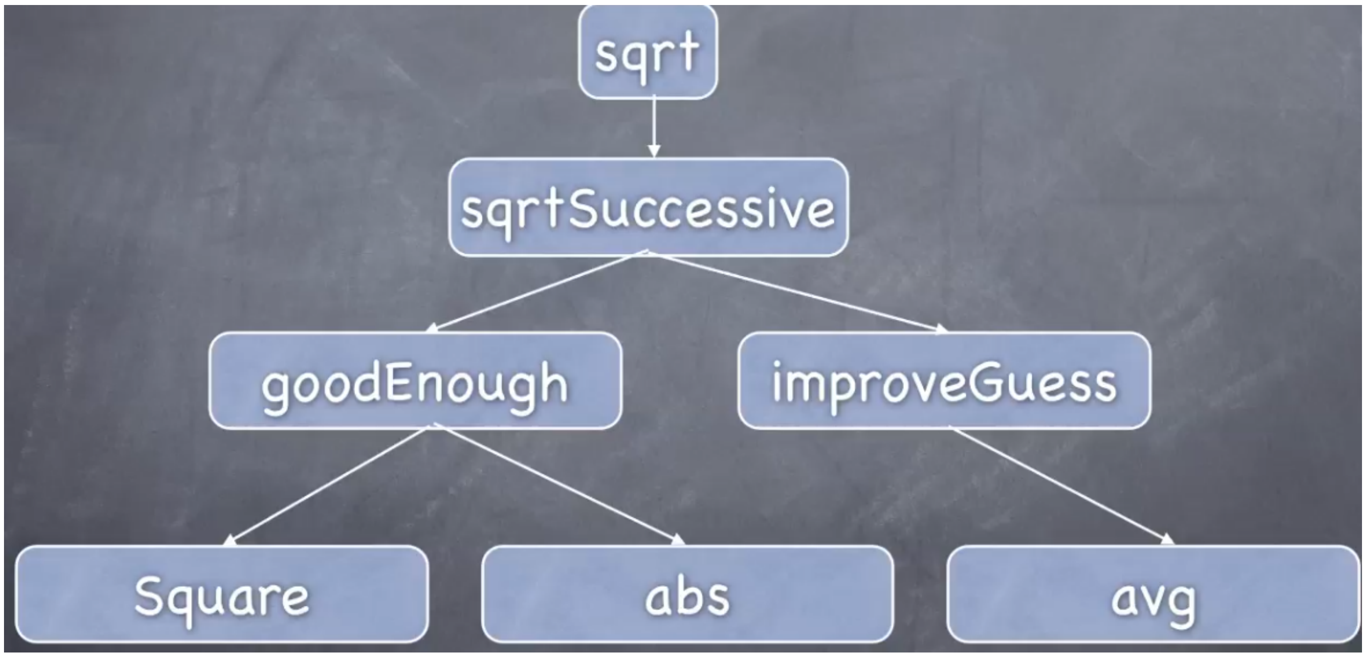
> If a language can't allow you to express and decompose procedures properly then, or protect abstraction and encapsulation then that language is not great for designing software.

Languages such as Java and C# are culprits of not fully protecting abstraction and encapsulation. Think back to a time when you've coded in either. Both indeed have encapsulation, however the only way to easily encapsulate your code is through objects. There is only encapsulation for objects in these languages and much less for procedures.

- To encapsulate procedure in these languages you need to make them private member functions of a class.

Finding the Square Root

Successive Refinement



Successive Refinement - technique for continuously improving the code while maintaining core functionality.

Let's get to an example based on our chart above.

```
println(sqrt(25))

def sqrt(candidate: Double) = {
        sqrtSuccessive(1, candidate)
}
```

First we call a function called `sqrtSuccessive` we give it a guess value and the candidate value.

```
println(sqrt(25))

def sqrt(candidate: Double) = {
        sqrtSuccessive(1, candidate)
}

def sqrtSuccessive(guess: Double, candidate: Double) = {
        if (goodEnoughGuess(guess, candidate)) {
                guess
        }
        else {
                sqrtSuccessive(improve(guess, candidate))
        }
}
```

Notice how we'll successively improve the guess and successively find the sqrt.

```
println(sqrt(25))

def sqrt(candidate: Double) = {
        sqrtSuccessive(1, candidate)
}

def sqrtSuccessive(guess: Double, candidate: Double) = {
        if (goodEnoughGuess(guess, candidate)) {
                guess
        }
        else {
                sqrtSuccessive(improve(guess, candidate))
        }
}

def goodEnoughGuess(guess: Double, candidate: Double) = {
        guess * guess == candidate.
}
```

> Notice that how we have written `goodEnoughGuess` is problematic. We should never compare two fully point numbers head on. This is because floating point arithmetic always involves some kind of int precision. These computations are unpredictable and depends on what language and class you're using.

We're going to use a tolerance value for us to get a more accurate floating point computation.

```
def goodEnoughGuess(guess: Double, candidate: Double) = {
        value TOLERANCE = 0.0000001
        Math.abs(guess * guess - candidate) < TOLERANCE
}
```

Now we'll implement our `improve` function with successive refinement.

- We want an average between the guess value and the candidate value.
  - Assume guess is right. `candidate/guess = guess`
  - Let's say the guess is not right. `candidate/guess > result`
  - `guess <= result <= candidate/guess`
  - So you can average a new guess as `(candidate/guess + guess) / 2`
  - And now you can successively refine it and improve further.

```
def improve(guess: Double, candidate: Double){
        (candidate / guess + guess) / 2
}
```

```
println(sqrt(25))

def sqrt(candidate: Double) = {
        sqrtSuccessive(1, candidate)
}

def sqrtSuccessive(guess: Double, candidate: Double) = {
        if (goodEnoughGuess(guess, candidate)) {
                guess
        }
        else {
                sqrtSuccessive(improve(guess, candidate))
        }
}

def goodEnoughGuess(guess: Double, candidate: Double) = {
        value TOLERANCE = 0.0000001
        Math.abs(guess * guess - candidate) < TOLERANCE
}

def improve(guess: Double, candidate: Double){
        (candidate / guess + guess) / 2
}
```

Look at how we took an algorithm that was decomposed into these modules and as a result it became a lot easier to express it.

- It would have been a crime if we crammed all of these procedures into one single function. It would have been messy, hard to navigate, hard to understand.

Overall we have the following modules as functions: `sqrt`, `sqrtSuccessive`, `goodEnoughGuess`, `improve`, `abs`.

We could have even abstracted an `avg` function if we wanted to.

Abstraction and Encapsulation

- Procedures should abstract and encapsulate the details.
- They should tell you what they provide and hide the details of they do it.
- User of a procedure should not be forced to know the details of the implementation.
- User should be able to conveniently use the procedure

As a user we should only be concerned with what a procedure does, and what parameters we have to send to it. We do not need to know all of the details of the procedure in order to use it!

- For example, do you know all of the code behind the `min` and `max` functions built into Python? No! But you know what these functions do, and what they need passed into them.

The reason for encapsulation is twofold:

1. If a procedure encapsulates what it does it can be changed later on. You can easily modify the implementation.
   - Flexible enough to be modified at any point.
2. **Reduces dependency.** You don't allow code to dependent on things that it should not dependent.
   - De-coupling.

The `sqrt` function fails at encapsulation in this case. I could come along and modify the `goodEnoughGuess` function and the `sqrt` function would be broken. Because by replacing that function we have have broken the encapsulation and jeopardized our procedure.

It is also a problem that our code depends on `goodEnoughGuess`. We're exposing the implementation details of the `sqrt` function which is not a good thing.

Despite our function being abstracted, it still fails at encapsulation. What would be nice is if we had everything `sqrt` needs inside of one function itself.

- We are not saying to write bad bloated code. But rather to write decomposed code that is abstracted and still **honors encapsulation**.

In the case of functional languages, functions have a higher order. You can write functions within functions.

- **You can encapsulate functions within another function.**

Encapsulated Block Structure: Nested Functions

```
def sqrt(candidate: Double) = {
        def sqrtSuccessive(guess: Double, candidate: Double) = {
                if (goodEnoughGuess(guess, candidate)) {
                        guess
                }
                else {
                        sqrtSuccessive(improve(guess, candidate))
                }
        }

        def goodEnoughGuess(guess: Double, candidate: Double) = {
                value TOLERANCE = 0.0000001
                Math.abs(guess * guess - candidate) < TOLERANCE
        }

        def improve(guess: Double, candidate: Double){
                (candidate / guess + guess) / 2
        }

        sqrtSuccessive(1, candidate)
}
```

Now `sqrt` function is both decomposing these functions but also calling these encapsulated functions.

- Everything the `sqrt` function needs is encapsulated within the function itself.

Think about achieving a greater degree of encapsulation. Notice how in Scala we are able of achieving encapsulation level at the procedure level.

Formal Parameters & Binding

The names chosen for formal parameters should not affect the choice of names by user of a procedure.

- Because you are simply passing a copy of the variable (symbol) to the parameter.

For Example, if I wrote a variable named `candidate` has nothing to do with the `candidate` named in the parameter.

```
val candidate = 25
println(sqrt(candidate))

def sqrt(candidate: Double) = {
        def sqrtSuccessive(guess: Double, candidate: Double) = {
                if (goodEnoughGuess(guess, candidate)) {
                        guess
                }
                else {
                        sqrtSuccessive(improve(guess, candidate))
                }
        }

        def goodEnoughGuess(guess: Double, candidate: Double) = {
                value TOLERANCE = 0.0000001
                scala.math.abs(guess * guess - candidate) < TOLERANCE
        }

        def improve(guess: Double, candidate: Double){
                (candidate / guess + guess) / 2
        }

        sqrtSuccessive(1, candidate)
}
```

> In languages such as JavaScript and Perl, you could have side-effects going on between variable names. Where it is less clear that the two are different.

The formal parameters are called **bound variables** and a procedure binds to its formal parameters.

- In our example, the parameter `candidate` is bounded to our symbol.

Local variables are only visible within a procedure.

- `TOLERANCE` is an example of this. It is only accessible from within `goodEnoughGuess`.

Dependency

Procedures tend to depend on other procedures

- In fact they are encouraged due to decomposition.
  Dependency increases coupling

> Coupling is the **interdependence** between modules.

If you modify the name of a procedure your procedure depends on you, you will affect your procedures implementation.

```
def good_enough_guess(guess, candidate)
        TOLERANCE = 0.00000001
        (guess * guess - candidate).abs < TOLERANCE
end

def sqrt_successive(guess, candidate)
        if good_enough_guess(guess, candidate)
                guess
        else
                sqrt_successive(improve(guess, candidate), candidate)
        end
end

def sqrt(candidate)
        sqrt_successive(1, candidate)
end

puts sqrt(25)
```

What if this was all in a separate file called `sqrt.rb` and we export that function.
And lets say that we wrote a function also called `good_enough_guess` ... what happened?

- We broke our code! Because our function `good_enough_guess` is exposed! it isn't encapsulated, so when we name another function of the same name, we end up breaking our code.

```
require 'sqrt'

puts sqrt(25)

def good_enough_guess(a, b)
        true
end

puts sqrt(25)
```

So what happens when we encapsulate our procedures. In the code below, our `good_enough_guess` function is encapsulated within our `sqrt` function. So our code is not effected by a redeclared function of the same name outside.

```
def sqrt(candidate)
	def good_enough_guess(guess, candidate)
		TOLERANCE = 0.00000001
		(guess * guess - candidate).abs < TOLERANCE
	end

	def improve(guess, candidate)
		(candidate / guess + guess) / 2
	end

	def sqrt_successive(guess, candidate)
		if good_enough_guess(guess, candidate)
			guess
		else
			sqrt_successive(improve(guess, candidate), candidate)
		end
	end

	sqrt_successive(1, candidate)
end

puts sqrt(25)
```

When we write that new function we didn't modify the encapsulated procedure.

Block Structure

Block Structure - the nesting of a procedure within another procedure.
You can eliminate problems with dependency and achieve decomposition through **block structures**.

Lexical Scoping

You don't have to pass parameters repeatedly to nested procedures.

- They can use the parameters (formal and local) defined in the nesting block.

Let's take a quick look at lexical scoping with an example from Java

1. We defined a variable `someValue` in the scope of our function `foo`.

2. We then declare new scope within this function.
3. We then use `someValue` within this new scope. (**Nested Procedure**)
   - Our code still works. Why?
     - Our code works because we are still in the same **Lexical Scope** of our variable.

```
public class Sample {
        public static void main(String[] args) {
                System.out.println(2.0 - 1.1);
        }

        public void foo(int value) {
                int someValue = 2;
                {
                        someValue * 2;
                }
        }
}
```

Okay well so what happens when we redefine that variable within that scope?

```
public class Sample {
        public static void main(String[] args) {
                System.out.println(2.0 - 1.1);
        }

        public void foo(int value) {
                int someValue = 2;
                {
                        int someValue = 54
                        int someThing = someValue * 2
                }
        }
}
```

Java doesn't allow us? Why is that?

```
Sample.java:10: someValue is already defined in foo(int)
```

Java is complaining that we already declared this variable in our lexical scope. This is something that works in other languages though, like for example in Objective C.

> Different languages treat lexical scoping in different ways. What you can define and redefine depends on the rules of the language.

Let's look at an example of lexical scoping in Scala

```
var total = 0
var list = List(1, 2, 3, 4, 5)
def totalUp(e: Int) = { total += e }
list.foreach(totalUp)
println("Total is " + total)
```

In this case, `e` is a parameter and `total` is bound to a variable outside. In lexical scoping, a procedure can use a variable in the scope of its own procedure or in the lexical scoping of a procedure of which it is contained.

There's another problem in our Scala code. We aren't making the most out of Lexical Scoping...

We have a `candidate` in every single scope of our code. This code could be way simpler if we reduced the noise in our code.
Using lexical scope could be a way to achieve just that.

- Code in the inner scope can readily access the code in the outside scope, according to lexical scoping.

```
val candidate = 25
println(sqrt(candidate))

def sqrt(candidate: Double) = {
        def sqrtSuccessive(guess: Double, candidate: Double) = {
                if (goodEnoughGuess(guess, candidate)) {
                        guess
                }
                else {
                        sqrtSuccessive(improve(guess, candidate))
                }
        }

        def goodEnoughGuess(guess: Double, candidate: Double) = {
                value TOLERANCE = 0.0000001
                scala.math.abs(guess * guess - candidate) < TOLERANCE
        }

        def improve(guess: Double, candidate: Double){
                (candidate / guess + guess) / 2
        }

        sqrtSuccessive(1, candidate)
}
```

> NOTE: Scala treats parameters as constants.

We can refactor this code and use lexical scoping to reduce the redundant declarations of `candidate` in our nested procedures.

```
val candidate = 25
println(sqrt(candidate))

def sqrt(candidate: Double) = {
        def sqrtSuccessive(guess: Double) = {
                if (goodEnoughGuess(guess)) {
                        guess
                }
                else {
                        sqrtSuccessive(improve(guess))
                }
        }

        def goodEnoughGuess(guess: Double) = {
                value TOLERANCE = 0.0000001
                scala.math.abs(guess * guess - candidate) < TOLERANCE
        }

        def improve(guess: Double){
                (candidate / guess + guess) / 2
        }

        sqrtSuccessive(1)
}
```

Notice how this becomes easier to read, and also clearer that we're using the same `candidate` variable. It helps to keep the variable as immutable, and providing safety that it isn't changing.

> It is best to try to not write code that modifies the value of a parameter. Even if a language allows you to.

How about Lexical Scoping in Ruby?

Ruby follows lexical scoping but only in code blocks or closures, but not for nested functions.

```
def sqrt(candidate)
        def good_enough_guess(guess, candidate)
                TOLERANCE = 0.00000001
                (guess * guess - candidate).abs < TOLERANCE
        end

        def improve(guess, candidate)
                (candidate / guess + guess) / 2
        end

        def sqrt_successive(guess, candidate)
                if good_enough_guess(guess, candidate)
                        guess
                else
                        sqrt_successive(improve(guess, candidate), candidate)
                end
        end

        sqrt_successive(1, candidate)
end

puts sqrt(25)
```

The `improve` function (procedure) should be called as a code block.

```
def sqrt(candidate)
        def good_enough_guess(guess, candidate)
                TOLERANCE = 0.00000001
                (guess * guess - candidate).abs < TOLERANCE
        end

        def sqrt_successive(guess, candidate)
                improve = Proc.new { |guess| (candidate / guess + guess) / 2 }
                if good_enough_guess(guess, candidate)
                        guess
                else
                        sqrt_successive(improve.call(guess))
                end
        end

        sqrt_successive(1, candidate)
end

puts sqrt(25)
```

How about we do the same with `good_enough_guess` and `sqrt_successive`?

```
def sqrt(candidate)
        sqrt_successive(guess, candidate) = Proc.new do |guess|
                improve = Proc.new { |guess| (candidate / guess + guess) / 2 }

                good_enough_guess = Proc.new do |guess|
                        tolerance = 0.0000001
                        (guess * guess - candidate)
                end
                if good_enough_guess.call(guess)
                        guess
                else
                        sqrt_successive.call(improve.call(guess))
                end
        end

        sqrt_successive(1, candidate)
end

puts sqrt(25)
```

As you can see here, we used block structures to support us in making the most of lexical scoping when it comes to nested procedures and encapsulation in Ruby.

Procedures in Different Languages

Each of these languages here are object-oriented languages on the JVM. You don't have to make the procedure as part of any class. They can be written independently.

1. Ruby
2. Groovy
3. Erlang
4. Clojure

Ruby

```
def add(*numbers)
        max_value = numbers[0]
        numbers.each do |e| # internal iterator
                max_values = e if e > max_values
        end
        max_value
```

```
end

puts max(1, 2, 4, 3, 0)
```

NOTE: Ruby is dynamically typed so you don't need to specify the data type of the parameters or the data type of what you return.

Groovy

```
def max(int[] numbers) {
        def maxValue = numbers[0]

        numbers.each { e ->
                if (e > maxValue)
                        maxValue = e
        }

        maxValue
}

println max(1, 2, 4, 3, 0)
```

NOTE: Groovy is optionally typed so you can specify the data types if you want to.

Scala

```
def max(numbers : Int*) = {
        var maxValue = num1
        for(e <- numbers) {
                if (e > maxValue)
                        maxValue = e
        }
        maxValue
}

var list = List(1, 2, 4, 3, 0))

println(max(list: _*)) // sendin this list as single parameters at a time.
```

NOTE: Scala is a statically typed language, yet it does have good type inference.

Erlang

```
main(_) ->
        io:format("~p~n", [max2(1,2, 4, 3, 0)])
max2(A, B) when A > B -> A;
max2(_, B) -> B.

max([H | []]) -> H; % base case
max([H | T]) -> % recursive bit
        max2(H, max(T)).
```

In Erlang there is no such thing as a loop. So we'll use recursion.

Notice how we treat the list. We receive the list and split it into a head and a tail. We use a recursive call to do this.

1. We recursively call `max` until we eventually the `max(T)` will end up with `T` being the only element in it. In which case we just return the head.

Clojure

Here's an example of the addition function in Clojure:

```
(defn add[op1, op2]
        (+ op1 op2)
)

(println (add 1 2))
```

Now how would the max function look like in Clojure?