

Process

A sequence of execution of a procedure.

How a procedure consumes the computational resources.

- A procedure can be the exact same, yet the processes may vary depending on the input.

1. Simple Process

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

This process is fairly simple, there's one level in the call stack.

- `square(5) ---> 5 * 5`

```
def square(x: Int) = {  
    x * x  
}  
  
println(square(5))
```

You can look at the call stack when an exception is thrown.

```
def square(x: Int) = {  
    throw new RuntimeException  
}
```

Run this code and you'll see that the exception is from one level deep in the call stack.

- Simple example of a process being executed

Iterative vs. Recursive

When writing processes and procedures there are two different ways of implementing code: **iteratively** and **recursively**.

Iterative

A procedure or process that **loops** through a set of values to perform a certain operation.

- State is determined by fixed set of variables and rules.

Recursive

A procedure or process that calls itself to fulfill the operation.



- There is a phase of expansion with deferred evaluation followed by a phase of contraction.
- Interpreter or runtime needs to keep track of deferred computations.

Procedure vs. Process

Just because a procedure is recursive does not mean that the process is recursive.

- You can have a recursive procedure with an iterative process

Iterative vs Recursive Process

|  Iterative Processes |  Recursive Processes |
|---|---|
| Carries a series of steps | Highly Expressive |
| You can stop at anytime and resume alter even on another processor with the current state. | Requires you to carry the current chain or sequence of calls with it. |
| Linear progression | Expensive demand of resources. |
| | Deals with multiple levels of stack. Likely to cause stack overflow with large sequences. |

However, don't shy away from recursion. You can write a recursive procedure that runs as an iterative process.

An example: Way to find factorial

- You can write it iterative or recursively
- You can process it iteratively or recursively

```
def factorial(number: Int) : Int = {
    var fact = 1

    for (i <- 1 to number) {
        fact *= i
    }

    fact
}
```

```
println(factorial(1))
println(factorial(2))
println(facotrial(3))
```

You can write this function iteratively by using either an external or internal iterator.

```
def factorial(number: Int) : Int = {
    var fact = 1

    (1 to number).foreach { e => fact *= e }

    fact
}

println(factorial(1))
println(factorial(2))
println(facotrial(3))
```

Whether you write this function with an internal or external iterator, the fact remains that you're using a mutable variable to *iteratively* find the factorial value.

The Procedure and Process are Iterative

- The procedure is iterative
- The process is also iterative because when it executes you initialize a variable `fact` you loop through one value at a time and you update the value of `fact`.

Another Example:

```
def factorial(number: Int) : Int = {
    if (number <= 2)
        number
    else
        number * factorial(number - 1)
}

println(factorial(1))
println(factorial(2))
println(facotrial(3))
```

Here we have a *recursive procedure*, the function calls itself.

Now when classifying the process, pay attention to the execution order.

- We defer computation of the multiplication until we reach our base case:

1. `5 * factorial(4)` --> we have to find `factorial(4)`
2. `4 * factorial(3)` --> we have to find `factorial(3)`
3. `3 * factorial(2)` --> we have to find `factorial(2)`
4. `2`

Notice that the same sequence and number of steps as before but much more expensive in resource usage.

- We accrue computations and then there's contraction from catching up with the computations that were left behind.
- We are multiple levels deep into the stack.

What happens if we significantly increase the call stack

```
def factorial(number: Int) : Int = {  
    if (number <= 2)  
        number  
    else  
        number * factorial(number - 1)  
}  
println(factorial(5000))
```

We end up with a **Stack Overflow** exception. This happens because we've ran out of space to store the stack. The procedure is so expensive that it crashes our program.

- The stack has a steep demand on our resources.

We can write an iterative process that can handle large numbers to avoid a stack overflow, as seen below:

```
import java.math.BigInteger  
  
def factorial(number: Int) : BigInteger = {  
    var fact = BigInteger.ONE  
    (1 to number).foreach { e =>  
        fact = fact.multiply(new BigInteger(e.toString))  
    }  
    fact  
}  
  
println(factorial(500))
```

So far we've talked about how an iterative process is derived from an iterative procedure. And a recursive process is derived from a recursive procedure.

While recursive procedures are highly expressive, the recursive process is quite costly for resources.

- Cannot be migrated between processors as easily as iterative processes can.

Iterative procedures compute values in a loop with linear progression. Iterative processes can be moved or paused at anytime.

- Can be moved between processors.

While recursive procedures are attractive, recursive processes may not be as desirable.

Best of Both Worlds: Recursive Procedure and an Iterative Process

Imagine code that is as expressive as a recursive procedure but can be ran as an iterative process at runtime.

How could we possibly write this? Let's take a quick look....

```
def f1(number: Int) : Int = {  
    if (number == 1)  
        number  
    else  
        number + f1(number - 1)  
}  
  
println(f1(5))
```

This code above will recursively add the numbers together. If you look at the stack itself, we are using 5 levels of the stack. (5 calls to `f1`).

- This will certainly lead to a stack overflow if we pass in a number that's too large.

The culprit of this inefficiency is on line 6. `number + f1(number-1)` .

- Let's break down this line:
 - We're adding the number to the result of our function call.

- On the stack for `f(5)`, you are saying add 5 to result of `f1(4)`. But now you have to wait for `f1(4)` to be available. So you leave the current stack to the stack of `f1(4)`. ---> `f1(4) : return 4 + f1(3)`

How about rather than doing this, we do something slightly different:

- In `f1(5)`:
Hey here is 5, take it with you, perform your computation `f1(4)` and then send the result to my caller.
 You take the partial result go on and perform your computation and send it onto the caller.
 So let's say we add a parameter `sum` to our recursive procedure.

```
def f1(sum: Int, number: Int) : Int = {
  if (number == 1)
    sum
  else
    f1(sum + number, number - 1)
}
```

Notice that line 7 does not perform any computations past the recursive call.

- Contrast this to before, when we had to come back and compute an addition after the recursive call.
- The computation is completed before the function is called.

This is an example of ***Tail Recursion***.

Tail Recursion

Tail recursion is when the last operation performed is a recursive call in a function. So nothing happens after the recursive call.

When the recursive call is the last expression to be evaluated, it can roll the call into a simple iteration instead of a jump.

- This leads to a **recursive procedure w/ an iterative process**

NOTE: Not all languages support tail recursion.

Compilers that support tail recursion or **tail call optimization (TCO)** convert our recursive procedure down into an iterative process.

- This grants us the best of both worlds.

In this case when we use tail recursion, our call stack has been converted to an iterative process thus, our call stack is now 1 level deep as opposed to 5 levels like before.

```
def f1(sum: Int, number: Int) : Int = {  
    if (number == 1)  
        sum  
    else  
        f1(sum + number, number - 1)  
}  
  
println(f1(0, 5))
```

Let's look at them side by side:

```
// This is a recursive procedure and a recursive process.  
def addRecursive_Recursive(number: Int) : Int = {  
    if (number == 1)  
        sum  
    else  
        number + addRecursive_Recursive(number - 1)  
}  
  
// This is a recursive procedure and an iterative process.  
def addRecursive_Iterative(sum: Int, number: Int) : Int = {  
    if (number == 1)  
        sum  
    else  
        addRecursive_Iterative(sum + number, number - 1)  
}  
  
println(addRecursive_Recursive(5))  
println(addRecursive_Iterative(5))
```

While they look similar at the code level, at the lower leveled bytecode level these two are quite distinct.

1. `addRecursive_Recursive` contains a call to itself at the bytecode level
2. `addRecursive_Iterative` will appear iterative at the bytecode level

This is because Scala's compiler supports **tail call optimization** and will convert a tail recursive algorithm into an iterative process under the hood.

Let's return back to our factorial function and look at it through the lens of differing procedures and processes.

```

import java.math.BigInteger

// Iterative Procedure - Iterative Process
def factorial_Iterative_Iterative(number : Int) : BigInteger = {
    var factorial = BigInteger.ONE
    (1 to number).foreach { e =>
        factorial = factorial.multiply(new BigInteger(e.toString))
    }
    factorial
}

def factorial_Recursive_Recursive(number : Int) : BigInteger = {
    if (number == 1)
        BigInteger.ONE
    else
        new BigInteger(number.toString).multiply(
            factorial_Recursive_Recursive(number - 1)
        )
}

def factorial_Recursive_Iterative(factorial : BigInteger, number : Int) :
BigInteger = {
    if (number == 1)
        factorial
    else
        factorial_Recursive_Iterative(
            factorial.multiply(new BigInteger(number.toString)),
            number - 1
        )
}

println(factorial_Iterative_Iterative(20))
println(factorial_Iterative_Iterative(50000))
println(factorial_Recursive_Recursive(20))
println(factorial_Recursive_Iterative(BigInteger.ONE, 50000))

```

We have a problem in our `factorial_Recursive_Iterative` we've muddled the interface of our function. *How can we fix this?*

- Rather than passing an extra parameter how about we internally implement `factorial` as a variable alongside an inner function.

```

def factorial_Recursive_Iterative(number : Int) : BigInteger = {
    def factorialImpl(factorial : BigInteger, number : Int) : BigInteger {
        if (number == 1)

```



```

        factorial
    else
        factorialImpl(
            factorial.multiply(new
BigInteger(number.toString)),
            number - 1
        )
    }
    factorialImpl(BigInteger.ONE, number)
}

```

To avoid polluting the interface, we added an inner function (encapsulation) to create a clear recursive procedure that runs an iterative process in languages that support tail recursions.

How about an example in Erlang

Let's toy around and start of with some code:

- So far this isn't going to scale well, and it certainly isn't DRY.

```

main(_) ->
    io:format("~p~n", [factorial(4)]).
    io:format("~p~n", [factorial(3)]).
    io:format("~p~n", [factorial(2)]).
    io:format("~p~n", [factorial(1)]).

factorial(4) -> 24
factorial(3) -> 6;
factorial(N) ->
    N.

```

Let's reel back and recall what we know:

- The factorial of 1 is equal to 1
- The factorial of n is $factorial(n-1)$

```

main(_) ->
    io:format("~p~n", [factorial(4)]).
    io:format("~p~n", [factorial(3)]).
    io:format("~p~n", [factorial(2)]).
    io:format("~p~n", [factorial(1)]).

factorial(1) -> 1
factorial(N) -> N * factorialImpl(N-1).

```

This however does not utilize tail recursion. There's still an operation that comes after our recursive call. So let's rewrite this.

```
main(_) ->
    io:format("~p~n", [factorial(4)]).
    io:format("~p~n", [factorial(3)]).
    io:format("~p~n", [factorial(2)]).
    io:format("~p~n", [factorial(1)]).

factorial(N) -> factorialImpl(1, N).

factorialImpl(Factorial, 1) -> Factorial; % base case
factorialImpl(Factorial, N) ->
    factorialImpl(Factorial * N, N - 1).
```

Here we finally have tail recursion!

Notice how Erlang uses pattern matching to divide. We have one function however with multiple entry points.

- If `N` is a `1`, return factorial
- If `N` is another number we run our recursive call. `factorialImpl(Factorial * N, N - 1)`

If you normally code in a certain language, you are used to a certain paradigm, and a certain set of idioms. Once you code in other languages you can see different things that we can do and different ways things can be done based on what the language supports and provides.

Recap:

Recall what we said about writing Procedures. Your procedures need to be providing a good abstraction. You need to make sure procedures have proper encapsulation. Good encapsulation means less dependencies, better modifications, and easier refactoring. Procedures should be fairly small.

Some algorithms are easier to express with mutability and iterative code, while others may be easier to express using recursive algorithms.

We also talked about the consequences of these decisions at runtime.

- Iterative processes occupy less resources. Give us flexibility of snapshotting, coming and resuming back.
- Recursive process demands more resources. We call on stacks, jumping through on call

stacks. Taking more computational resources on hardware.

You can get the best of both worlds by using **tail recursion**. And you know you're using tail recursion, when the recursive call is at the end of your operations.

- You have the benefit that the procedure can be recursive but the process can be iterative. If you have a tail recursion you can encapsulate a function so you don't burden your code with extra details. Highly expressing your code.