Programming Languages and Paradigms Notes

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**.

Typing

Values - Literals that we use in applications that represent data.

- e.g. a string `"hello"`, or a integer `4`

1. What's a Type

A data type is a set of values where you say `v is of type T means v ∈ T`.
If an expression or function is of type `T`, their `result ∈ T`

2. Different Types of Data Types

Primitives

For example in C, Java, and C++ we deal with `int`, `double`, `long`, `char`, etc.

- Just values that get passed around.
- Typically don't have any behavior.

> There are languages that are purely Object Oriented languages such as Ruby and Scala. These languages deal with objects and not primitive types.

Composites

A composite type is a type that is a collection of other types. Typically Objects are composites, as they can be made up of different types.

> Purely Object Oriented languages are made up of composites.

Recursive

For example, an object may have objects which in turn have other objects.

- In the case of a recursive structure, a person is related to another person, and that person may have a relationship with another person.

3. Functional Languages: Lists and List Processing

Functional languages often tend to deal with lists or collections of data much more often.

> There's a special treatment for lists in functional languages. Data is treated immutable entities.

Because the content is immutable, you need to find an efficient way to access lists.
List processing is critical:

- Often functional languages take a list of values and compose a list by taking an element and attaching it to the head of the list.
- Functional languages will operate on a list and often closely do work with the head or the tail of the list.

Consider a list of values in Erlang.

```
main(_) ->
        process([1, 2, 3, 4, 5]).

process([H|[]])
process(L) ->
        io:format("~p-", [L]).
```

Take a look at the syntax on line 6.

```
process([H]|[])
```

We can split a list by its head and the rest of the list. We can also represent the rest of it by a letter `T`.
We can also split and print the list as seen below.

```
main(_) ->
        process([1, 2, 3, 4, 5]).

process([H|[]]) ->
        io:format("~p", [H]).
process([H|T]) ->
        io:format("~p-", [H]),
        io:format("~p-", [T]).
        process(T).
```

Splitting lists is built directly into functional languages such as Erlang. We are then able to process lists much easier.

This shows how a list can be processed into heads and tails. These list operations are provided in these objects, so it provides a different design consideration.

- Designing code in these languages thus is very different than designing code in Object Oriented languages.

4. Static vs Dynamic Typing

Languages can be classified by their typing systems. There are **statically typed languages** and **dynamically typed languages**.

There are noticeable advantages that each typing system has over the other. These are all tools that are great to use in the right case.

Statically Typed Languages

A ***statically typed language*** is a language that will verify the type information of variables, objects, and data used in the program at **compile time**.

Dynamically Typed Languages

A ***dynamically typed language*** is a language that may not even have a compiler. **Type verification is not enforced**, the typing is often somewhat liberal. This is either because it doesn't have a compiler or because the compiler is lenient.

- Rather than ensuring type safety at this time, it works with a language to see if the type of call is supported and thus whether it will fail.

A Statically Typed Example:

```
pubic class Sample {
        public static void main(String[] args){
                int value = 4;
                value = "hello" // this would cause a compilation error.
                System.out.println(value)
        }
}
```

In the code above, a variable declared as an integer obviously cannot be assigned the value of a string.

Yet, this is not to say that static languages like Java don't have inference based type conversions. Let's take look at a quick example:

```
pubic class Sample {
        public static void main(String[] args){
```

```
                double value = 4.0;
                value = 3 //implictly converts the int to a double
                System.out.println(value)
        }
}
```

What happened at compile time:

1. Compiler stored a value of `3.0` into the variable `value`.
    - Compiler accepted the assignment but implicitly converted it to a `double` behind the scenes.

```
value = 1
puts value

value = "hello"
puts value
```

Notice that Ruby did not complain. Ruby is a purely object oriented language so what Ruby did was, it pointed `value` to a `Fixnum` (int) object and then altered this pointer when reassigning `value` to a string.

> A dynamic language allows you to reassign a value to a reference of a different type.

> A static language, when you redefine a reference of an object of a certain will enforce that the new reference is of the same type.

Do NOT assume that a Dynamically typed language is one where you don't specify a type and a static language is one where you do define a type.

That is a wrong assumption. It is not true the the difference is specifying a data type.

- If you look at Java, we did in fact specify the data type. However, static typing does not mean more typing. It simply means that the type information is verified at compile time.

Let's use Scala, a statically typed language, as an example.

```
var value = 1
println(value)
```

```
value = 4
println(value)
```

Notice how we did not specify the data type of the variable anywhere. And Scala runs this code perfectly fine. Why?

> Scala supports **type inference**.

Type inference means we the language is smart enough to infer the type based on the context.

- In our example, Scala sees that we initialize `value` with the value `1`. Thus Scala infers that the data type of our variable is an integer.

Now let's look at this a bit more. What if we assigned `value` to a different data type? What happens?

```
var value = 1
value = "hello"
println(value)
```

We receive a **compilation error** that says we required an Int yet we found a string value.

- The compiler was expecting an integer and is now enforcing it at compiling time.

> The less statically typed a language is the more typing you will have to do and provide more details because the language is not adequate in figuring out what these type details are for you.

5. Which one is better?

A Drawback to Statically Typed Language
Times Where You Need to Lowering of Type Checking
Just because a language is statically typed it does not mean that all the type information is verified perfectly at compile time.

- There are times when you will have to resort to the lowering of type checking even in static typed languages. There may be times when you can't infer a type correctly at compile time.
- Or in the case where you have a very generalized object.

Let's take look at such an example:

```
public class Sample {
        public static void main(String[] args){
```

```
                List scores = new ArrayList();
                scores.add(1);
                scores.add(2);

                int total = 0;
                for(Object e : cores) {
                        total += (Integer) e;
                }
                System.out.println("the total is, " + total);
        }
}
```

In this example, it thinks that scores is returning objects and not integers, and this is because we are using a `List`. So we have to go ahead and cast as you see above in our for loop.

- this issue sits on a massive flaw. Our list can hold objects but not all objects have to be of the same data type. So what happens if we added a double to our List and we keep the `Integer` casting?

```
public class Sample {
        public static void main(String[] args){
                List scores = new ArrayList();
                scores.add(1);
                scores.add(2);
                scores.add(1.0)

                int total = 0;
                for(Object e : cores) {
                        total += (Integer) e;
                }
                System.out.println("the total is, " + total);
        }
}
```

The code above will result in an casting exception error. Because we can't cast a double to an int like this.

You could fix by providing better type safety by declaring that the List as a list of integers. You then will not need to do the casting yourself in the source code. However, the casting will still be done under the hood.

```
public class Sample {
        public static void main(String[] args){
                List<Integer> scores = new ArrayList<Integer>();
```

```
                scores.add(1);
                scores.add(2);

                int total = 0;
                for(int e : cores) {
                        total += e;
                }
                System.out.println("the total is, " + total);
        }
}
```

If we add a double, `scores.add(1.0)` for instance, we will cause a compilation error. However if we pass this list to a function our integer casting will run without a compilation error! BUT, when we run the code, the cast exception error has come back!

```
public class Sample {
        public static void oops(List list) {
                list.add(1.0);
        }
        public static void main(String[] args){
                List<Integer> scores = new ArrayList<Integer>();
                scores.add(1);
                scores.add(2);
                oops(scores);

                int total = 0;
                for(int e : cores) {
                        total += e;
                }
                System.out.println("the total is, " + total);
        }
}
```

This is an example of the static typing in Java can still get by and cause incorrect casting.

Benefits of Static Typing

There are many benefits to Static Typing:

- Built in type safety brought through the compiler's verification.

Benefits of Dynamic Typing

There are many benefits to Dynamic Typing:

- It's easier to write ***metaprogramming*** - Can write a program that can write part of the program at runtime.
- Freer, not as limited by the compiler.

Let's quickly look at Metaprogramming

Java will not allow you to add this function at run time, as it throws a compilation error before you can even run the program.

```
public class Sample {
        public static void main(String[] args){
                String greet = "hello";
                System.out.println(greet);

                greet.shout(); //this gets rejected at compile time.
        }
}
```

Groovy is an example of a language that supports both static typing and dynamic typing. Look at how Groovy lets us run this following code. Despite not being able to identify the method `shout` at compile time.

```
greet = "hello"
println greet

println greet.shout()
```

So let's add a try catch and see what happens

```
greet = "hello"
println greet

try {
        println greet.shout()
} catch(Exception ex){
        println "oops, looks like I can't do that"
}
```

It still compiles and runs! We catch the error and now we know to add the method! It doesn't tell you that you can't do it. It's **far less restricting**. It beckons you to try things out!

```
greet = "hello"
println greet

try {
        println greet.shout()
} catch(Exception ex){
        println "oops, looks like I can't do that"
}

println "Let's add it!"
String.metaClass.shout = {->
        delegate.toUpperCase()
}

println greet.shout()
```

Let's take a quick detour over to an example in Ruby, a dynamic language.

```
class Person
        def work
                puts "working..."
        end
end

same = Person.new
sam.work

sam.playTennis
sam.playFootball
```

We catch a runtime error where we have multiple methods that are not defined. So let's try to fix that by adding a function called `method_missing`.

```
class Person
        def work
                puts "working..."
        end

        def method_missing(name, *args)
                puts "you called #{name}"
        end
end

same = Person.new
```

```
sam.work

sam.playTennis
sam.playFootball
```

Notice now, rather than crashing and complaining we now invoke `method_missing`.

```
class Person
        def work
                puts "working..."
        end

        def method_missing(name, *args)
                activities = ['Tennis', 'Football']

                activity = name.to_s.split('play')[1] # method name starts with
play

                if (activities.include?(acitivity)) # check activity named.
                        puts "I'd like to play #{activity}"
                else
                        puts "Nope, I don't play #{activity}"
                end
        end
end

same = Person.new
sam.work

sam.playTennis
sam.playFootball
sam.playPolitics
```

Notice how we dynamically change the behavior of the code.
Think of a real world example of this. You may need to query a database at runtime and you can either end up being accepted or declined.

> This is exactly what frameworks like Ruby on Rails provides, dynamic capabilities for you based on certain command states.

Note that dynamic typing does not impede on metaprogramming.

So having looked at both static and dynamic languages, we return back to the big question. Which one is better? Well it depends on the given problem on hand.

- You may have an application that needs decent compile time and interface verifications (Static >>> Dynamic)
- You may need to write code which utilized metaprogramming, or a fully dynamic domain specific language (Dynamic >>> Static)
  Whether you should use a static or dynamic language depends on the best way to achieve your goal. These languages are tools, it is a vehicle.
- You cannot be adamant and say only one type of typing is correct.
- We should have the flexibility to pick and choose the language that best fits your needs.

In conclusion they both have pros and cons.

6. Design by Contract vs Design by Capability

The languages that you use affect your software design!

When you code in a statically typed language you often use what is called a design by contract.

- You ask the compiler "Do you support this?" and follow its restrictions at compile time.
- If an object conforms to this interface then you are **guaranteed to know** that this object provide its services.
  Dynamic languages do not typically use design by contract but rather use a **Design by Capability**.
- If an object supports a certain method it's more of a handshaking. You approach your coding like "Oh can I do?" often experimenting as you type.

7. Strong vs Weak Typing

Does your programming language or the run-time that it stands on, ensure that the object that you are dealing with is the proper type when you are exercising it's behavior?

For example, you have an object and using a static typing system you can verify that it is of a certain type. What if the static typing allowed you to cast the object to a certain type.

- In the case of C and C++, you can take a pointer to an object and cast it.

```
Car* pCar = (Car*) pObj;
pCar->drive(); //C++
```

What happens if `pObj` isn't really a `Car` object? You perform a casting that is forcing the conversion of this pointer to a pointer of a different type. (**Coercion**).

In C++, once you get past the static typing of the compiler, at runtime there are no guarantees. The behaviors are unpredictable. **This is an example of a weakly typed language.**

In a **strongly typed language**, the type checking/verification happens at the time the program is running. The type is verified at run-time.

> Java is an example of a **strongly typed language**

```
public class Sample {
        public static void main(String[] args) {
                Object obj1 = "hello";
                Object obj2 = 2;

                use(obj1);
                use(obj2);
        }
        private static void use(Object obj) {
                String str = (String) obj;

                System.out.prinln(str);
        }
}
```

Notice in this example there's a class cast exception on line 12 at run time. Everything compiles fine, but Java is able to find a run-time exception.

> In C++, a weakly typed language, if you aren't given a seg-fault a bug can be hard to find as the program misbehaviors in a subtle manner. Because the language itself is weakly typed.

> People often assume that static typing means strong typing and dynamic typing means weak typing. This is **NOT** true. Strong typing is a type verification at run time it has nothing to do with if verification is done at compile time.

Let's take a look at an example in Groovy (**Strongly Dynamic**)

```
def inst1 = "hello"
String inst2 = "hello"

println inst1
println inst1.class

println inst2
```

```
println inst2.class

inst1 = 1 // assigne new Integer reference to inst1
inst2 = 1 // called toString() obtained String instance and set inst2 to it

println inst1
println inst2
println inst1.class
```

Notice that at first `inst1` pointed to an instance of a string object, then it was reassigned to an integer. The `def` keyword allows for Groovy to use type inference. While on line 2, your string variable was declared as a string and thus its type was enforced on line 11.

> The type is specified, thus showing Groovy's optional typing. However, Groovy slants towards being a dynamically typed language despite the small tinge of static typing permitted within it. As seen in the example below.

```
def inst1 = 1
Integer inst2 = 1

println inst1
println inst1.class
println inst2
println inst2.class

inst1 = "A"
inst2 = "A" // converted A to an Integer under the hood.

println inst1
println inst1.class

println inst2 //printing 65 not an A
println inst2.class // pointing to an object of an Integer
```

Whenever you assign an object to an instance reference you are performing a type conversion behind the scenes.

Because we assigned `inst1` to an integer (`1`), `inst1` is defined to point at an `Integer` so when we reassign it to a string, that string's value is implicitly converted.

```
def inst1 = 1
Integer inst2 = 1

println inst1
```

```
println inst1.class
println inst2
println inst2.class

inst1 = "A"
inst2 = "A" // converted A to an Integer under the hood.

println inst1
println inst1.class

println inst2 //printing 65 not an A
println inst2.class // pointing to an object of an Integer

// Say we assign inst1 to a newly constructed String
inst1 = new StringBuilder("hello")

println inst1
println inst1.class

inst2 = new StringBuilder("hello")
println inst2
```

A type error is caused from line 18 (`inst1 = new StringBuilder("hello")`),
`Cannot cast object 'hello' with class 'java.lang.StringBuilder' to class 'java.lang.Integer'

In other words, Groovy has caught a cast exception because it doesn't know how to cast a `StringBuilder` to an `Integer` so it bailed out. Showing how it is strongly typed.

- A weakly typed language would have allowed such a conversion.

Expressions and Symbols

1. Expression vs Statement

Expressions and statements are two distinct language constructs.

1. Statements are commands that you **execute** and ask to compute a certain action.
2. Expressions **return values** to you.

For example:

```
def exampleFunction():
        x = 4 # statement (assignment statement)
        if x is 10: # statement
                print(x*2) # statement
        else:
         return x + 4 # expression
```

While in many languages you have expressions and statements. Some languages have chosen to do away with any statements. Everything in those languages are expressions, all the commands are built with expressions. And you compose an application in such a language solely based on expressions.

What are the benefits of that?
By treating everything as expressions you are able to provide a lot more **conciseness** in your code.

- This is because if you have an expression can you can ignore the result if you don't want it. However in a statement, there is no result to be used -- thus statements are fairly limiting.

> Ruby and Groovy are examples of languages that can use everything as expressions. e.g. if/else, for loops, while loops, or even try and catch.

```
def isEven(number)
        result = "odd"
        result = "even" if (number % 2 == 0)
        result #remember that there's no need for a return keyword here.
end
```

```
puts isEven(4)
puts isEven(3)
```

We could take that if statement and treat it as an expression as we have below:

```
def isEven(number)
        result = if (number % 2 == 0)
                "even"
        else
                "odd"

        "The given number is #{result}"
end

puts isEven(4)
puts isEven(3)
```

How about a for loop as an expression?

```
def foo(number)
        for i in 1..4
                puts i
        end

        puts "hello"
end

puts foo(40)
```

> `puts` is an expression itself, and always returns a `nil`.

2. Interning

String interning is a method of storing only one copy of each distinct string value which must be immutable.
Interned strings are stored in a special memory region of the JVM. This memory region is of a fixed size.

Interning of a Sting in Java

When you use expressions your code becomes a lot more concise and simple. You won't spend time making temporary variables, assigning those variables to values, and so on.

```
public class Sample {
        public static void main(String[] args) {
                String bad = new String("hello");

                String greet = "hello";
                System.out.println(bad);
                System.out.println(greet);

                String greet2 = "hello";
                System.out.println(greet2)
        }
}
```

`greet` and `greet2` point to exactly the same instance in memory, whereas `bad` is pointing to a difference instance of `String` in memory.

- You can tell that `bad` is pointing elsewhere as it uses the `new` keyword to allocate new space in memory.

Now let's check if `bad`, `greet`, and `greet2` are equal to one another.

```
public class Sample {
        public static void main(String[] args) {
                String bad = new String("hello");

                String greet = "hello";
                System.out.println(bad);
                System.out.println(greet);

                String greet2 = "hello";
                System.out.println(greet2)

                System.out.println(greet == bad); //outputs false
                System.out.println(greet.equals(bad)) //outputs true

                System.out.println(greet.equals(greet2)) //outputs true
                System.out.println(greet == bad) //outputs true
        }
}
```

> Recall that in Java `==` does a referential comparison while `equals()` does a value based comparison.

It tells us while `greet` and `bad` are equal in value, `greet` and `bad` aren't pointing to the same location. While `greet1` and `greet2` are pointing to the same location.

3. Symbols

Symbols provide you interned values for any variable in memory.

Think of a symbol as a reference to a variable.

```
class Car
        def tow
                puts "tow called..."
        end
        private :tow
end

car = Car.new
car.tow #error message from private member tow called.
```

On line 5 we are sending a symbol, a reference to the function called `tow`. This is not `tow` but however the reference to `tow`.

- We are not using the function, we are not invoking the function, we are referring to the function. And that is all that we are doing.

```
x = 4
puts :x
puts :x.class
```

Here we see we just created a `Symbol` called `x` on line 2.

If you're writing a function that's using meta-programming and you want to receive function references then it becomes effective to use symbols.

```
class Person
        def method_missing(name, *args)
                puts "You called #{name}"
        end
end

sam = Person.new
sam.sing
```

> Examine what `name` is. `name` is not a string it is a symbol.

> Do not confuse these references with the references in C++. This is using interning more than references in C++. You could modify values in that memory location in C++ unlike symbols in Ruby.

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.

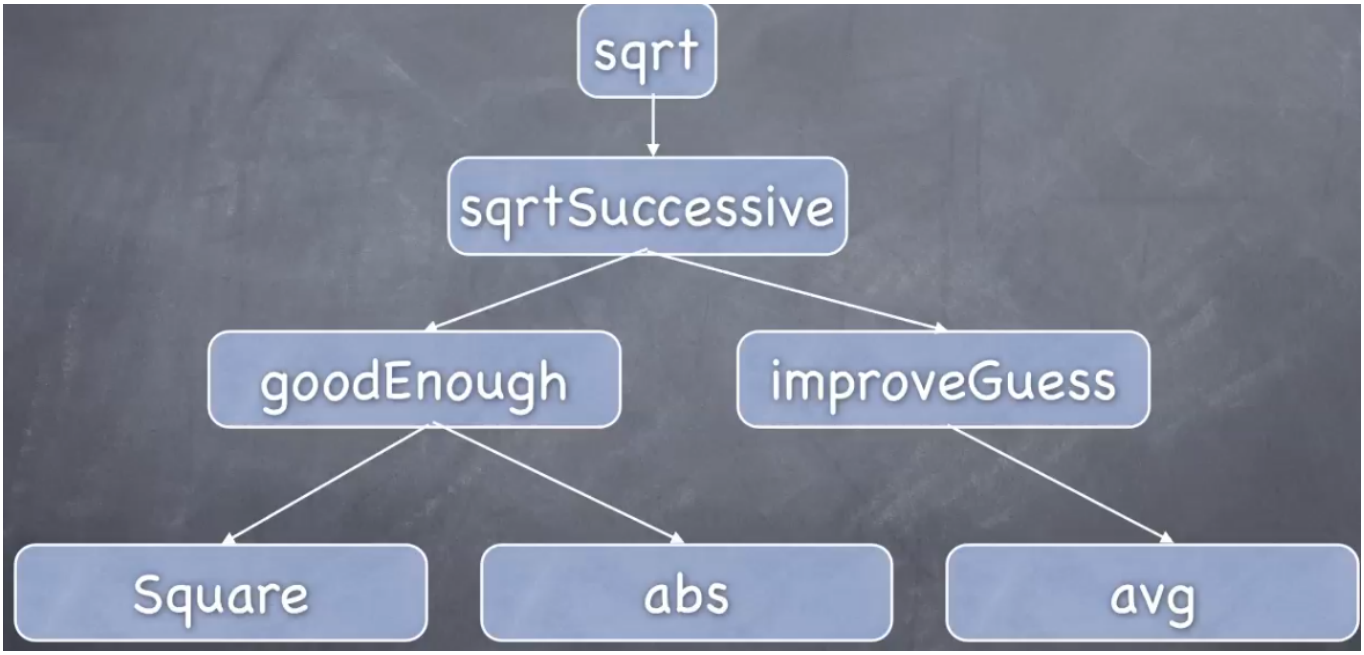
> If a language can't allow you to express and decompose procedures properly then, or protect abstraction and encapsulation then that language is not great for designing software.

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?

Iterating in Erlang

Iterating in Erlang

Accumulation vs Iteration

Accumulation is a special type of iteration that involves collecting a result by repeatedly applying an operation to a sequence of elements.

- You typically need to use an accumulator variable. This variable is initialized to a starting value and updates as you process each element.
- We repeatedly apply function (operation) to elements.

`lists:foldl` and accumulation

`lists:foldl` is a high order function used for "folding" or accumulating elements in a list from left to right. A function (operation) is applied to each element and accumulates as it traverses the list.

The basic syntax is this:

```
lists:foldl(function, accumulator, list)
```

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.

Object Orientation

1. What's Object Oriented Programming?

Object Oriented programming is a paradigm built off of the decomposition of an application into objects. Objects represent abstractions.

Objects are an abstraction that is encapsulated. An object can do things for you by invoking a method on it. Messages and information can be passed between objects to accomplish things.

In an Object Oriented System you build your system from a series of objects, and these objects provide you services.

2. Pillars of the Paradigm

There are four main pillars of OOP:

- **Abstraction**
- **Encapsulation**
- **Inheritance**
- **Polymorphism**

Abstraction

We use abstraction for communicating and dealing with complexity. Abstraction is how we represent details of our application.

- To abstract is to purposefully generalize details or attributes.
- We hide complexity by representing it through models.

Encapsulation

Encapsulation is the separation of what you do from how you do it. **It is a separation from the interface from the implementation.**

- Where you publish your object's capabilities to the outside world, and then you hide the details of how it's implemented.
- It is not about making a variable private, it is hiding implementation.
- **Reduces coupling**, rather than depending on the specific details of your code's implementation a user depends on what you provide. When you change your code, the

user will not be affected by it.

Inheritance

Inheritance is when a class can inherit from another class. When a class inherits from another class, that class is able to derive all the behavior of the base class.

> Inheritance is quite overrated. It is not as important to achieve reusability of code by using inheritance. There are languages that can provide you a better way of yielding reusability than inheritance.

Inheritance is useful when you want to promote **substitutability**. When you want to use an instance of a class in place of an instance of another class.

If you don't want to use substitutability, and rather use the methods of a base class in a derived class then Inheritance may not be the best choice. In this case it may be better to use ***delegation*** than inheritance. Let's take a look in some code below!

```
class Eraser {
        public void erase() { System.out.println("erasing..."); }
}

class BetterEraser extends Eraser {
        public void erase() { System.out.println("erasing efficiently...") }
}

public class Sample {
        public void eraseBoard(Eraser eraser) {
                eraser.erase();
        }

        public static void main(String[] args) {
                Eraser eraser = new Eraser();

                Sample instance = new Sample();
                instance.eraseBoard(eraser);

                BetterEraser betterEraser = new BetterEraser();
                instance.eraseBoard(betterEraser);
        }
 }
```

This code will not compile if `BetterEraser` does not inherit from `Eraser`. This is to allow us to be able to use an instance of `BetterEraser` in place of where an instance of `Eraser` would be

used.

How about we inherit another class from `Eraser`, we'll create the `Instructor` class and see what happens.

```
class Eraser {
        public void erase() { System.out.println("erasing..."); }
}

class BetterEraser extends Eraser {
        public void erase() { System.out.println("erasing efficiently...") }
}

class Instructor extends Eraser {
}

public class Sample {
        public void eraseBoard(Eraser eraser) {
                eraser.erase();
        }

        public static void main(String[] args) {
                Eraser eraser = new Eraser();

                Sample instance = new Sample();
                instance.eraseBoard(eraser);

                BetterEraser betterEraser = new BetterEraser();
                instance.eraseBoard(betterEraser);

                Instructor instructor = new Instructor();
                instructor.erase();
        }
 }
```

An unfortunate consequence of this inheritance is now that you're able to send an `Instructor` object into a method where an `Eraser` is expected. You can now treat an `Instructor` as an `Eraser` because of inheritance. We don't want the `Instructor` to be treated as an `Eraser` but rather be able to use an `Eraser`.

So what can we do to make that happen?

In Java, you can write those methods inside of the `Instructor` class and you can hold an instance of the `Eraser` class in there.

OR..... you can *replace inheritance with delegation*.

```
class Eraser {
        public void erase() { System.out.println("erasing..."); }
}

class BetterEraser extends Eraser {
        public void erase() { System.out.println("erasing efficiently...") }
}

class Instructor extends Eraser {
        private final Eraser _eraser = new Eraser();

        public void erase(){
                _eraser.erase();
        }
}

public class Sample {
        public void eraseBoard(Eraser eraser) {
                eraser.erase();
        }

        public static void main(String[] args) {
                Eraser eraser = new Eraser();

                Sample instance = new Sample();
                instance.eraseBoard(eraser);

                BetterEraser betterEraser = new BetterEraser();
                instance.eraseBoard(betterEraser);

        }
 }
```

If you want an instance of a class B to be treated as an instance of a class A then use inheritance.

However if you simply want an instance of class B to use an instance of class A, then use delegation.

- Can also give you reusability as well.

Why do people use inheritance more often than delegation?

- It's often easier to write inheritance than delegation.
    - Let's say we add a method to the `Eraser` class and we want that method in the `Instructor`, we'd have to come and change the `Instructor` class.

In this example from Java, delegation is a poor design. Why? Because this class fails the Open-Closed Principle.

> **Open-Closed Principle** - Objects or entities should be open for extension but closed for modification.

But this does not have to be the case! Let's take a look at an example from Groovy:

```
class Eraser {
        public void erase() { System.out.println("erasing...") }
}

class Instructor {
        @Delegate Eraser eraser = new Eraser()
}

instructor = new Instructor()
instructor.erase()
```

There is a `@Delegate` which is the delegation mechanism that Groovy provides us to use.

- The delegation in Groovy is OCP compliant.

As you can see from this example, not all languages come with delegation mechanisms.

Polymorphism

Polymorphism says go ahead and call a method, and the method that's called is based on the type of the object at run-time and not the type of the object at compile-time.

An example is that in the `eraseBoard` method, we can invoke the `erase` method on either an `Eraser` or a `BetterEraser` at runtime. It is not based on the type of the `Eraser` that you've specified at compile-time. So this is polymorphic.

3. Classes, Instances, Fields, Properties, and Methods

1. You can create ***instances*** of classes, except for an abstract class in most languages.
2. Classes can have ***fields***, they may have ***properties*** which you can call upon an object to implement properties.

3. Classes may have ***methods***, some are modifiers, some are query methods, you may have final methods which shouldn't be overridden, or you may have virtual functions.

Singletons

A singleton is an object that can have only one instance.

> Singletons are a problem in Java and C#

In this code below, we create a singleton. If an instance is already created then we return that instance. However if the instance is null we create that new `Singleton`.

```
class Singleton{
        private static Singleton instance;

        public static Singleton getInstance() {
                if (instance == null) {
                        instance = new Singleton();
                }
                return instance;
        }
}

public class Sample {
        public static void main(String[] args) {
                Singleton inst1 = Singleton.getInstance();
                Singleton inst2 = Singleton.getInstance();
        }

}
```

However, there are multiple ways to break this code!!!

- **Reflection** - You can then go to the `Singleton` class, get a reference to the constructor, and use the constructor info object to create an instance. Like in C#.
- **Multi-threading** - This code is not thread safe, under a multithreaded system, you could have multiple threads calling this code at the same time. Both threads check the instance at the same time and both see null and create an instance.

Your code may appear to be correct when you write a singleton, but it can very easily be problematic. Creating a thing as a singleton in Java is really difficult.

The issue of singletons was so major in Java, that Scala went ahead an fixed the problem themselves.

```
object Singleton {
        def op1() { println("op1.....") }
}

Singleton.op1
```

In the case of Scala we create singletons with the keyword `object`.

- notice we didn't use the keyword `new` in any of this code.

4. Static

Static members and methods aren't linked to any object instance and can be accessed directly by the class reference.

> A static member contains the same value across all class variables. A static method or field applies to every object of the class, instead of being associated with just the declared variable of the class.

> In Java, Static methods (or fields) are only allowed to access other static methods and fields within its defined class.

"Static is evil" - Venkat

In an Object Oriented project, you know that you have objects and methods reside on objects. But what happens with the static?

- When you write a class you write methods that belong to the instance and then methods that belong to a class. And a whole bunch of confusion

Statics aren't fun to work with while testing, while developing a web app or even in an OO application.

- It's pretty difficult to mock static methods.
- Requests to a web app become convoluted with statics are used.

Scala and Static

Scala is a fully Object Oriented language that doesn't support static. Scala instead supports companion objects. You can create singletons that accompany another class as a companion.

```
class Marker(val color : String) {
        println("Creating a marker of color " + color)
```

```
        override def toString = {
                "Marker color is " + color
        }
}

new Marker("blue")
new Marker("blue")
new Marker("red")
```

```
Creating a marker of color blue
Creating a marker of color blue
Creating a marker of color red
```

We now have multiple instances of marker, it's no long a singleton. We have two blue markers with two different instances. We want there to be a single instance for a marker of the same color.

```
class Marker(val color : String) {
        println("Creating a marker of color " + color)
        override def toString = {
                "Marker color is " + color
        }
}

object MarkerFactory {
        val markers = Map(
        "blue" -> new Marker("blue"),
        "red" -> new Marker("red"),
        "green" -> new Marker("green")
        )

        def getMarker(color : String) = {
                markers.get(color)
        }
}

val blueMarker = MarkerFactory.getMarker("blue")
println(blueMarker)
```

Notice that `MarkerFactory` is defined as a an object, a singleton, and as a result `getMarker` is a method that you can directly call on that object.

- So in Scala you implement *static methods* not as a static methods but as methods on singletons.

So when we run this code, Scala returns `Some(Marker color is blue)` from our `getMarker("blue")` call. We get an object of an option type to avoid a null pointer exception.

- Returned either a `None` object or a a `Some` object.

To get the real object itself you can add `.getOrElse(null)` after our `getMarker()` call.

You can also use a companion object here.

```
class Marker(val color : String) {
        println("Creating a marker of color " + color)
        override def toString = {
                "Marker color is " + color
        }
}

object Marker {
        val markers = Map(
        "blue" -> new Marker("blue"),
        "red" -> new Marker("red"),
        "green" -> new Marker("green")
        )

        def getMarker(color : String) = {
                markers.get(color)
        }
}

val blueMarker = MarkerFactory.getMarker("blue")
println(blueMarker.getOrElse(null))
```

The singleton has the same name as the class, so now it's a companion class. You are only now allowed to us any methods of the `Marker` class on your singleton.

- The companion object has full access to the class that it is a companion to.

Scala being a purely OO language means that every method belongs to an instance.

- Some belong to a method of a class, some belong to a singleton.

5. Aggregation vs Composition

Let's jump into an example in C++.

- In C++, you have **association** - a relationship between two objects. You also have **aggregation** where one object owns another object. And you also have **composition**, where one object is composed of another object.

These ideas are much clearer in C++.

> Say you create a class `Person`. There is memory allocated for a `Person`. And embedded in that memory is data for the objects related to it. You can find a `Heart` and a `Brain` in that memory.

```
class Person {
        Heart* pHeart;
        Brain theBrain;
}
```

Brain is 12 byes in size
Pointer to Heart is 4 bytes.
Person is of size 12 + 4 + ...

In this case you would say a brain is a composition and a heart is an aggregation.

- Why is this?
  - The brain is embedded into the `Person` object. Each `Person` object is composed of a `Brain` object.
  - While the `Heart` is just being pointed to, this `Person` owns a heart via a pointer.

Now imagine the consequences of this. When the brain is embedded into this `Person` object. When you delete the `Person` object, you also delete the `Brain`. But the `Heart` is not, just the pointer. You can change the `Heart` object and you can reset the pointer to a pointer of another object.

When it comes to Aggregation you can change the object that you aggregate. But in Composition you ***CANNOT***.

In composition the memory of the object is embedded within another. The lifetime of the objects are tied together.

In Java and C#, there is no distinction. Both instances of `Heart` and `Brain` are pointers. So you can point them to something and change them later on.

```
class Person {
        Heart heart;
        Brain brain;
}
```

Java and C# do not support composition by default. But you can still model it. So how do we do that?

```
class Person {
        private Heart heart;
        private Brain brain;

        public Person(Brain theBrain, Heart aHeart) {
                brain = theBrain;
                heart = aHeart
        }

        public void changeHeart(Heart aHeart) {
                heart = aHeart;
        }
}
```

Notice that you have not provided a way to change the `Brain` in the same way you can change the `Heart`.

From the point of view of object modelling, the concepts of association, composition and aggregation have a distinct meaning. But from the point of view of coding, you'll have to pick and choose how to express these concepts based on the nature of the language you are using.

6. Interfaces

In C++ you have no concept of interfaces, you have abstract classes and purely abstract classes.

- A purely abstract class is a class made up entirely of purely virtual methods.
  So there is no distinct concept of interfaces although it is there in other forms.

In Java there is a clear distinction of interfaces. As there is in C# and Groovy.

- An **interface** is a grouping of methods with no implementation.

However, languages like Scala and Ruby do not provide interfaces. Because interfaces are traditionally used for **designed by contract**.

- So you're promising that you abide by a certain implementation when using an interface.

Certain languages prefer a design by capability instead of a design by contract. Certain design favors influence the availability of interfaces.

- So it depends on what language you are using and what design approach you are following.

Interfaces define a certain method and because this method is expected you can verify it at compile time and at runtime you know the right method will be called because in addition to the interface you can rely on ***polymorphism***.

Interface Collision

What if you have two interfaces with exactly the same method? *Trouble awaits in some languages...*

> In this example, you will likely say "Hey, but you're the one naming these methods why don't you avoid the collision by naming them differently". Well let's pretend that this is a real life example. What would happen in the case where you import two different libraries with two interfaces that have methods of the same name?

```
interface FootballPlayer {
        void play();
}

interface PianoPlayer {
        void play();
}

class FootballPlayingPerson implements FootballPlayer {
        public void play() {
                System.out.println("kick the ball...");
        }
}

class PianoPlayingPerson implements PianoPlayer {
        public void play() {
                System.out.println("press keys...")
        }

}

class Sample {
        public static void playFootball(FootballPlayer player) {
```

```
                player.play();
        }

        public static void playPiano(PianoPlayer player) {
                player.play();
        }
        public static void main(String[] args) {
                FootballPlayingPerson peter = new FootballPlayingPerson();

                playFootball(peter);

                PianoPlayingPerson susan = new PianoPlayingPerson();
                playPiano(susan)
        }
}
```

So far so good, there are no issues or collisions here. But what if we add a new class called `MultiSkilledPerson`?

```
interface FootballPlayer {
        void play();
}

interface PianoPlayer {
        void play();
}

class FootballPlayingPerson implements FootballPlayer {
        public void play() {
                System.out.println("kick the ball...");
        }
}

class PianoPlayingPerson implements PianoPlayer {
        public void play() {
                System.out.println("press keys...")
        }

}

class MultiSkilledPerson implements PianoPlayer, FootballPlayer {
        public void play() {
                System.out.println("????")
        }
}
```

```
class Sample {
        public static void playFootball(FootballPlayer player) {
                player.play();
        }

        public static void playPiano(PianoPlayer player) {
                player.play();
        }
        public static void main(String[] args) {
                FootballPlayingPerson peter = new FootballPlayingPerson();

                playFootball(peter);

                PianoPlayingPerson susan = new PianoPlayingPerson();
                playPiano(susan)

                MultiSkilledPerson bob = new MultiSkilledPerson();
                playFootball(bob);
                playPiano(bob);
        }
}
```

Now how do we distinguish whether we called `play()` as a `PianoPlayer` or as a `FootballPlayer` in the case of our `MultiSkilledPerson`? Java provides no context nor distinction. But C# does!

```
using System;
using System.Collections.Generic;
using System.Ling;
using System.Text;

namespace Sample {
        interface FootballPlayer {
                void Play();
        }

        interface PianoPlayer {
                void Play();
        }

        class FootballPlayingPerson : FootballPlayer {
                public void Play() {
                        Console.WriteLine("kick the ball...");
                }
```

```
        }

        class PianoPlayingPerson : PianoPlayer {
                public void Play() {
                        Console.WriteLine("press key...");
                }
        }

        class MultiSkilledPerson : PianoPlayer, FootballPlayer {
                public void Play() {
                        Console.WriteLine("press keys...");
                }

                void FootballPlayer.Play() { // explcit interface!
                        Console.WriteLin("kick the ball...")
                }
        }

        class Program {
                static void playFootball(FootballPlayer player) {
                        player.Play();
                }

                static void Main(string[] args) {
                        FootballPlayingPerson peter = new FootballPlayingPerson();
                        PlayFootball(peter);

                        PianoPlayingPerson susan = new PianoPlayingPerson();
                        PlayPiano(susan);

                        MultiSkilledPerson bob = new MultiSkilledPerson();

                        PlayFootball(bob);
                        PlayPiano(bob);
                }
        }
}
```

C# supports explicit interfaces to provide context. In this case we used an explicit interface for `MultiSkilledPerson`. When we invoke the `play` function through a `FootballPlayer` function call on a `MultiSkilledPerson` object, then we will call the appropriate method.

7. Overloading

Overloading is a way for you to name multiple methods by the same name.

- For example you can `pay()` by a cheque, you may `pay()` by a credit card or debit card, or maybe by cash.
- Back in time C would make you name each method by a different name however C++ has introduced function overloading.

In the terms of binding, overloading is ***static binding***. Meaning that it determines what method to call at compile time.

Overloading requires that both functions are within the same scope.

- If you have a method in two different classes of the same name, then this is **NOT** overloading as they are in differing scopes.

> Overloading is confused with Polymorphism but overloading does not provide any extensibility to code in any way.

> Fun fact! **Ruby does not support method overloading.** When a second method is defined with the same name it completely overrides the previously existing method!.

8. Polymorphism

Polymorphism is an extremely important concept that provides extensibility in code.

Methods are **polymorphic** when they have the same name but are in different scopes.

Polymorphism in a statically typed language: The methods that you talk about have the same name but in two different scopes that are related as a base class and derived class.

Polymorphism in a dynamically typed language: The two methods can be any class scope, it doesn't matter if there's a relationship between the two classes.

The method that gets called is based on the type of the object that you invoking the method on at runtime.

You may have the impression that inheritance is required in order to have polymorphism. However, this may not be the case in all languages. This is what you would think if you have only used **statically typed languages**.

What Polymorphism does requires is the ***presence of the method on that object at runtime*** when you call it.

- It does not require an interface

However, because of the restrictions of statically typed languages Polymorphism can lead to odd behaviors at times.

- For example in C++, it may be a crime to override a function that is not virtual.
- In Java and C#, you need to take effort to preserve the parameters that polymorphic methods take.

Let's take a look at an example!

```
abstract class Animal {
        public final void eat() { System.out.println("eating..."); }
        public abstract void makeNoise();
}

class Dog extends Animal {
        @Override public void makeNoise() {
                System.out.println("bark...");
        }
}

class Cat extends Animal {
        @Override public void makeNoise() {
                System.out.prinln("meow...");
        }
}

class Sample {
        public void playWithAnimal(Animal animal) {
                animal.eat();
                animal.makeNoise();
        }

        public static void main(String[] args) {
                playWithAnimal(new Dog());
                playWithAnimal(new Cat());
        }
}
```

We know that we can't call a method on an abstract class because there's no implementation.

- However the compile will let us run it anyways, because Java knows that at runtime that the `Animal` object that you pass to the `playWithAnimal` method will be an instance of another class that has the methods of `Animal`.

What Java is going to do here is that it's going to call the `makeNoise` function as a polymorphic function call.

- Java is simply going to treat the method polymorphically.

> Static methods are NEVER polymorphic. Statics and Polymorphism are enemies.

Notice that we call the same function with two different objects. And that our polymorphic functions are *dynamically bound*.

What's the difference between static and dynamic binding?

In **static binding**, the compiler clearly knows which method it's going to call at compile time. In the case of **dynamic binding**, the compiler defers to runtime to decide on which method to call.

- Polymorphism relies on runtime binding to know which function call.
  - We don't know that our noise will be a bark until a `Dog` is passed at runtime.

```
import java.util.ArrayList;

class Sample {
        public static void main(String[] args) {
                use1();
                use2();
        }

        private static void use1() {
                ArrayList<Integer> list = new ArrayList<Integer>();
                list.add(1);
                list.add(2);

                System.out.println("Number of elements is" + list.size());

                list.remove(0);
                System.out.println("Number of elements is" + list.size());
        }

        private static void use2() {
                Collection<Integer> list = new ArrayList<Integer>();
                list.add(1);
                list.add(2);

                System.out.println("Number of elements is" + list.size());
```

```
                list.remove(0);
                System.out.println("Number of elements is" + list.size());
        }
}
```

Notice how you make only one change in the code, you are treating the `ArrayList` as an `ArrayList` versus treating an `ArrayList` as a `Collection` (it's base class).

- You would argue that there is no change to this code at all.

However, both functions are producing different outputs from one another.

- Notice that the `remove` method in `use2`, it isn't being called on an instance of `ArrayList` it is being called on an instance of the `Collection` interface.
  - The `remove` method on a `Collection` removes an `Object` and not an `int`!!!
  - Java's API was designed very wrong. They overrode the base class but changed the signature on the method.

This here is called auto-boxing. It takes the `0` and instead of treating it as an `int`, it treats it as an `Integer` object and passes it to the remove method. Once the `Integer` arrives to the `remove` function there is no object 0 so instead of removing it, it ignores it!

What polymorphism should do is that when you call a method on an object, it shouldn't matter how you called that method **the effect should be exactly the same**.

- It shouldn't matter whether you called it as a base or derived class, the behavior should be the same.

```
class CurrencyConverter {
        public void convert(double value) {
                System.out.println("CurrencyConverter's convert(double) called...")
        }
}

class BetterCurrencyConverter extends CurrencyConverter {
        public void convert(int value) {
                System.out.println("BetterCurrencyConverter's convert(int) ")
        }
}

class Sample {
        public static void useConverter(CurrencyConverter converter) {
                converter.convert(5);
        }
```

```
        public static void main(String[] args) {
                useConverter(new CurrencyConverter());
                useConverter(new BetterCurrencyConverter());
        }
}
```

If polymorphism worked correctly, it should know to call the method in the derived class!

Line 9 causes issues because we changed the signature of the `convert` function; there we are expecting an `int` instead of a `double`.

- We passed an integer, yet we called the base class's method which was expecting a double.

What happened was that when you called the `convert` method, it didn't matter if you passed a `CurrencyConverter` or a `BetterCurrencyConverter`, this call ended up calling the method on the `CurrencyConverter` alone, on its own.

This was still dynamic binding, it checked the type of the `converter` object. So then how did it not call the function for the `BetterCurrency Converter`?

Well this is because what's happening is that we:

1. Postpone to runtime the method to call
2. At runtime, examine the real object that converter refers (or points) to
3. Call the method `convert` on it.

Spot the sneakily hidden data type conversion in Java

Remember that Java is a **statically typed language**. At compile time we have to undergo **type verification**.

- The compiler performs a type verification that asks if we can pass a certain type to the method.
    - HOWEVER, recall that at ***compile time***, the type of `converter` is `CurrencyConverter`!

When we pass a `5`, the compiler will implicitly convert the data type of the integer to a double.

- So before it postpones the method call to runtime, it has **already changed the data type at compile time***.
- At runtime, when the `BetterCurrencyConverter` is called, the `convert` function that takes an `int` is not suitable for our `double`. So then the `BetterCurrencyConverter` then takes us to the base class.

- Because instead of overriding the `convert` function, we have overloaded the `convert` function! Because we have two different signatures in differing scopes but under the same name.

In a dynamically typed language, type verifications do not happen at compile time, they are postponed at runtime.

- As a result, the runtime behavior determines the call.

9. Multimethods

The ability for dynamically typed languages to postpone type conversions to runtime is why you have multi-methods in dynamic languages.

Multimethods - the method you call is determined by the type of the target object (left of the dot) and the type of the parameters.

> Polymorphism does not consider the type of the parameters.

Multimethods can be thought of as an extension of polymorphism. They are normally only found in dynamic languages because it relies on the runtime type.

```
class CurrencyConverter {
        public void convert(double value) {
                System.out.println("CurrencyConverter's convert(double) called...")
        }
}

class BetterCurrencyConverter extends CurrencyConverter {
        public void convert(double value) {
                System.out.println("BetterCurrencyConverter's convert(double) ")
        }
        public void convert(int value) {
                System.out.println("BetterCurrencyConverter's convert(int) ")
        }
}

public void useConverter(CurrencyConverter converter) {
        converter.convert(5);
        converter.convert(5.0);
}

useConverter(new CurrencyConverter())
useConverter(new BetterCurrencyConverter())
```

Groovy's method dispatching is quite complex. If you were to look at the bytecode produced you will notice that it calls the methods on a call-site, which does the dispatching at runtime.

- The decision of which method gets called is postponed to the runtime rather than compile time in groovy.

Let's look at another example of multimethods at work.

Recall the code from earlier...

```
import java.util.ArrayList;

class Sample {
        public static void main(String[] args) {
                use1();
                use2();
        }

        private static void use1() {
                ArrayList<Integer> list = new ArrayList<Integer>();
                list.add(1);
                list.add(2);

                System.out.println("Number of elements is" + list.size());

                list.remove(0);
                System.out.println("Number of elements is" + list.size());
        }

        private static void use2() {
                Collection<Integer> list = new ArrayList<Integer>();
                list.add(1);
                list.add(2);

                System.out.println("Number of elements is" + list.size());

                list.remove(0);
                System.out.println("Number of elements is" + list.size());
        }
}
```

The output was:

```
Number of elements is 2
Number of elements is 1
```

```
Number of elements is 2
Number of elements is 2
```

The Java API unfortunately failed to remove an element because of auto-boxing. The two different functions had a differing behavior because one treats our list as the derived class and the other treats it as a the base class.

How does this code work in Groovy?

```
import java.util.ArrayList;

class Sample {
        public static void main(String[] args) {
                use1();
                use2();
        }

        private static void use1() {
                ArrayList<Integer> list = new ArrayList<Integer>();
                list.add(1);
                list.add(2);

                System.out.println("Number of elements is" + list.size());

                list.remove(0);
                System.out.println("Number of elements is" + list.size());
        }

        private static void use2() {
                Collection<Integer> list = new ArrayList<Integer>();
                list.add(1);
                list.add(2);

                System.out.println("Number of elements is" + list.size());

                list.remove(0);
                System.out.println("Number of elements is" + list.size());
        }
}

use1()
use2()
```

```
Number of elements is 2
Number of elements is 1
Number of elements is 2
Number of elements is 1
```

Notice how we call `use1` and `use2` with the same exact code in Groovy, but because Groovy has multi-methods the two functions behavior the same way.

Ruby as an example

> In the case of Ruby, Ruby is a dynamically typed language so there are no type checks at compile time, and Ruby has no function overloading. So by default Ruby always has multi-methods.

> Or you could say that Ruby always has polymorphism because you cannot call the method of the derived classes, because Ruby simply doesn't care about the signature of the methods. However, you cannot accidentally call the base method because it is the derived method that always takes up things.

```
class Base
        def foo(value)
                value + 2
        end
end

class Derived < Base
        def foo(*values)
                values.size()
        end
end

def use(inst)
        puts inst.foo(4)
end

use(Base.new)
use(Derived.new)
```

```
6
1
```

Ruby will always call the method on the derived class, it doesn't take account for the data type of the parameters in the method.

- Our one value, our one integer, was assumed to be a list when we passed it into our derived method.
  Ruby doesn't know the value that's being passed in and assumes it's type at runtime. Ruby entirely overrides this method all the time, so you can argue that Ruby does not provide multimethods.
- Because Ruby is always going to call the method on the derived class.
  - The data type is not a consequence at all when calling a method.

9. Execute Around Method Pattern

When we deal with OOP we will often need to deal with garbage collection.

- In C++, if you allocate an object on the stack it will go away on its own. If you allocate an object on the heap you will need to delete it yourself, or else it will stay in the heap occupying memory.
- In Java and C# there is automatic garbage collection. In these languages you don't have much control over the scope of the collection of this garbage, because Java has the ARM (automatic resource management).

An Execute Around Method Pattern in Scala

```
class Resource {
        println("Creating resource")
        def close() {
                println("cleaning up the resource")
        }

        def op1() { println("op1...") }
        def op2() { println("op2...") }
        def op3() { throw new RuntimeException }
}

val resource = new Resource
resource.op1
resource.op2
resource.op3
resource.close
```

Notice how we can never close the program because `op3` will always throw a `RunTimeException` before we can even invoke the `close` method.

```
class Resource {
        println("Creating resource")
        def close() {
                println("cleaning up the resource")
        }

        def op1() { println("op1...") }
        def op2() { println("op2...") }
        def op3() { throw new RuntimeException }
}

val resource = new Resource
try {
        resource.op1
        resource.op2
        resource.op
} finally {
        resource.close
}
```

Notice how when the code *blows up*, it calls the `close` function before it *blows up*.

- The code called `op1` and `op2` and then the cleanup happened before the error.

This is exactly what the ARM in Java provides. The problem here is that as a programmer we have worry about writing this out every single time. We have to worry about writing the clean up every time. Let's look into this further...

```
class Resource private {
        println("Creating resource")
        private def close() {
                println("cleaning up the resource")
        }

        def op1() { println("op1...") }
        def op2() { println("op2...") }
        def op3() { throw new RuntimeException }
}

val resource = new Resource
try {
        resource.op1
        resource.op2
        resource.op
} finally {
```

```
        resource.close
}
```

We've made the constructor private and the `close` function private.

- But then we can't create an instance of this class! Don't fret let's create a companion object and see where that takes us.

```
class Resource private {
        println("Creating resource")
        private def close() {
                println("cleaning up the resource")
        }

        def op1() { println("op1...") }
        def op2() { println("op2...") }
        def op3() { throw new RuntimeException }
}

object Resource {
        def use(block : Resource => Unit) {
                val resource = new Resource
                try {
                        resource.op1
                        resource.op2
                        resource.op
                } finally {
                        resource.close
                }
        }
}

Resource.use { resource =>
        resource.op1
        resource.op2
        resource.op3
}
```

This is called the **Execute Around Method Pattern**, where we execute a certain pre-operation and post-operation around this method.

- Execute around method patterns are available in Java and C#, but are limited.

In languages that support closures and lambda expressions you are able to use this pattern.

An Execute Around Method Pattern in Ruby

```
class Resource
        def initialize()
                puts "creating resource"
        end

        def op1
                puts "op1 called..."
        end

        def op2
                puts "op2 called..."
        end

        def close
                puts "clean up..."
        end

        private :close

        def self.use
                resource = Resource.new
                begin
                        yield resource
                ensure
                        puts "clean up resource here"
                end
        end

end

resource = Resource.new
resource.op1
resource.op2
```

Notice that the code *blew up* but before it threw that exception it was cleaned up as we wanted.

An Execute Around Method Pattern in Groovy

```
class Resource {
        def static use(closure) {
                Resource resource = new Resource()
                println "create resource"
```

```
                try {
                        resource.with closure
                } finally {
                        println "clean up..."
                }
        }

        def op1() { println "op1 called..." }
        def op2() { println "op2 called..." }
}

Resource.use {
        op1()
        op2()
}
```

Groovy provide some very concise code that allows us to create a cloned closure, to delegate it and call the clone on an invisible object.

11. Traits and Mixins

Recall any issues from multiple inheritance?

1. You'll have to deal with method collision. Like in the case of interface collision seen earlier.
2. Inheriting from two base classes who share a parent class.
    - Note; Java and C# does not support multiple inheritance.

To avoid multiple inheritance you can use traits and mixins.

- Traits are a concept in Scala, which can be used at compile time
- Mixins are a concept in Ruby and Groovy which can be used at run time.

Traits

```
trait Friend {
        val name : String
        def listen = println("I am " + name + " your friend, listening...")
}

def talkToAFriend(friend : Friend) = {
        friend.listen
}
```

```
class Human(val name : String) extends Friend {}

class Animal(val name2 : String) {}
class Dog(override val name : String) extends Animal(name2) with Friend

val peter = new Human("Peter")
peter.listen
talktToAFriend(peter)

val peter = new Dog("Snowy")
snowy.listen
talktToAFriend(snowy)
```

> A **Trait** is an interface with partial implementations in it.

In this example our `Dog` class inherits from the `Animal` class and from the `Friend` trait.

- This is an example of a ***mixin***, what we did was we extracted the traits separately and mixed it into the class.

Scala does not only allow you to mix in traits into classes, you can also mix in traits at the object level.

```
trait Friend {
        val name : String
        def listen = println("I am " + name + " your friend, listening...")
}

def talkToAFriend(friend : Friend) = {
        friend.listen
}

class Human(val name : String) extends Friend {}

class Animal(val name2 : String) {}
class Dog(override val name : String) extends Animal(name2) with Friend

val peter = new Human("Peter")
peter.listen
talktToAFriend(peter)

val peter = new Dog("Snowy")
snowy.listen
talktToAFriend(snowy)
```

```
class Cat(override val name : String) extends Animal(name)

val alf = new Cat("Alf")

// mixing in a trait at the object level
val yourCat = new Cat("yourcat") with Friend
yourCaat.listen
talkToAFriend(yourCat)
```

Specific instances of a class can be mixed in with traits using the same syntax, in Scala.

Ruby

```
module Friend
        def listen
                puts "I am " + name + " your friend... listening"
        end
end

class Human
        include Friend
        attr_accessor :name
end

class Dog
        attr_accessor :name
end

class Cat
        attr_accesor :name
end

peter = Human.new
peter.name = "Peter"

peter.listen

snowy = Dog.new
snowy.name = "Snowy"
snowy.listen

alf = Cat.new
alf.name = "Alf"

your_cat = Cat.new
```

```
class <<your_cat
        include Friend
end

your_cat.name = "yourCat"
your_cat.listen
```

You are able to mix in traits not only into classes but objects as well at runtime in Ruby.

> What Ruby does here is that it chains these objects together. But we can look at that again later.

What's nice about traits is that they don't follow a vertical inheritance hierarchy.

Here's what happens in Ruby:

- `alf` is a pointer to a `Cat` instance
- If the `Cat` instance does not have the method then check the `Cat` metaclass to see if it has the method.
  Because in Ruby, instances of a class can have methods that the metaclass does not have.

`your_cat` is a bit different than `alf` here

- `your_cat` first checks if the *virtual class* has the method. The virtual class has the methods of the `Friend` mixin mixed in.
- Then it checks the instance
- Then it checks the metaclass

```
module Mixin1
        def f1()
                puts "Mixin1 f1 called"
        end
        def g1()
                puts "Mixin1 g1 called"
        end
end

module Mixin2
        def f2()
                puts "Mixin 2 f2 called"
        end
        def g1()
```

```
                puts "Mixin2 g1 called"
        end
end

class MyClass
        include Mixin1
        include Mixin2
        def g1()
                puts "Mixin2 g1 called"
        end
end

inst = MyClass.new
inst.f1
inst.f2
inst.g1
```

Chaining order of our mixins

`inst -> (vc Myxlass) -> (vc Mixin2) -> (vc Mixin1) -> MyClass metaclass`

Rather than a vertical hierarchy it's a right to left hierarchy.

```
abstract class Writer {
        def write(msg : String)
}

class StringWriter extends Writer {
        val target = new StringBuilder
        override def write(msg : String) = {
                target.append(msg)
        }
}

def writeStuff(writer : Writer) = {
        writer.write("This is stupid")
        println(writer)
}

trait UpperCaseFilter extends Writer {
        abstract override def write(msg : String) = {
                super.write(msg.toUpperCase())
        }
}

trait ProfanityFilter extends Writer {
        abstract override def write(msg : String) = {
```

```
                super.write(msg.replace("stupid", "s****"))
        }
}

writeStuff(new StringWriter) // instance of StringWriter

writeStuff(new StringWriter with UpperCaseFilter)
// instance of an anonymous class

writeStuff(new StringWriter with ProfanityFilter)
//notice that order matters
writeStuff(new StringWriter with UpperCaseFilter with ProfanityFilter)
writeStuff(new StringWriter with ProfanityFilter with UpperCaseFilter)
```

The `super` doesn't mean that you're calling on the base class. Notice that the base class `Writer` is abstract and doesn't even have a `write` function that we can call.

What `super` means is that you're calling the class that the `UpperCaseFilter` trait is attached to. Or in the class that the instance is attached to.

Line 25 is not an instance of `StringWriter`! It is an instance of an anonymous class that extends from `StringWriter` and mixes in `UpperCaseFilter`.

- So the sequence is:
  StringWriter <- UpperCaseFilter <- anonymous class <- instance

Notice that the order matters. In the case of our first call that mixes in `ProfanityFilter`, the "STUPID" gets replaced with "S****". *While in our second call this is not the case*, the word "STUPID" is not replaced. Because `ProfanityFilter` only works in lowercase.

- This is because of the order of function calls in the right to left sequencing.

Let's try to recreate this example in Ruby

```
module UpperCaseFilter
        def write(msg)
                super msg.update
        end
end

class StringWriter
        attr_reader :target
        def initialize
                @target = ''
        end
```

```
        def write(msg)
                @target << msg
        end
end

def write_stuff(writer)
        writer.write("This is stupid")
        puts writer.target
end

def write_stuff(writer)
        writer.write("This is stupid")
        puts writer.target
end

write_stuff(StringWriter.new)
write_stuff(StringWriter.new.extend(UpperCaseFilter))
write_stuff(StringWriter.new.extend(ProfanityFilter))
```

> Here in Ruby this is a runtime injection instead of a compile time manipulation that Scala does.

12. Covariance and Contravariance

Typically when you create objects or values in a strong type system, it will enforce the types at compile time.

Covariance is where you can treat an object of that type or as an object of its base class.

Contravariance is where you can treat an object of a base class as an object of one of its derived classes.

An example of a Covariance in Java:

```
class Animal {}
class Cow extends Animal {}

class Barn {
        Animal getAnimal() { return null; }
}

class BarnOfCow extends Barn {
        @Override Cow getAnimal() { return null; }
}
```

```
public class Sample {
        public static void main(String[] args)
                String str1 = "hello"
        Object obj = str1;
}
```

On line 9, the base method `getAnimal` is returning an `Animal` while the derived method is returning a `Cow`.

However Java is careful of ***contravariance***. We could not replace `Cow` with `Animal` on this line as we would be returning an "incompatible type".

```
class Animal {}

public class Sample {
        public static void playWithANimal(List<Animal> animals) {}
        public static void copy(List<Animal> animals) {}

        public satic void main(String[] args) {
                List<Animal> animals1 = new ArrayList<Animal>();
                List<Dog> dogs1 = new ArrayList<Dog>();
                List<Dog> dogs2 = new ArrayList<Dog>();

                playWithAnimal(animals1)

                List<Animal> animals2 = new ArrayList<Animal>();

                copy(animals1, animals2)

                copy(dogs1, dogs2) //an attempt at covariance. ERROR!

                copy(animals1, dogs1) //ERROR!

                copy(dogs1, animals1) //ERROR! AGAIN, same issue
        }
}
```

Java is saying you can't use covariance on this type, you can't call copy by sending a list of `Dog` objects when you need a list of `Animal` objects.

- If you were to send a collection of dogs to copy a collection of animals then you may end up putting the wrong type of animal into this list. Like for example a crocodile or a tiger.

```
class Animal {}

public class Sample {
        public static void playWithANimal(List<Animal> animals) {}
        public static void <T> copy(List<T> animalssrc, List<T> animalsdest) {}

        public satic void main(String[] args) {
                List<Animal> animals1 = new ArrayList<Animal>();
                List<Dog> dogs1 = new ArrayList<Dog>();
                List<Dog> dogs2 = new ArrayList<Dog>();

                playWithAnimal(animals1);

                List<Animal> animals2 = new ArrayList<Animal>();

                copy(animals1, animals2);

                copy(dogs1, dogs2); //an attempt at covariance. ERROR!

                copy(animals1, dogs1); //ERROR!

                copy(dogs1, animals1); //ERROR!
        }
}
```

The Java compiler tries to protect you. It never wants you to treat a collection of derived types as a collection of base types. And vice versa.

There are two ways to specify the type in this code. To allow the previous copy statements.

```
class Animal {}

public class Sample {
        public static void playWithANimal(List<Animal> animals) {}
        public static void <T> copy(List<T> animalssrc, List<? super T>
animalsdest) {}

        public satic void main(String[] args) {
                List<Animal> animals1 = new ArrayList<Animal>();
                List<Dog> dogs1 = new ArrayList<Dog>();
                List<Dog> dogs2 = new ArrayList<Dog>();

                playWithAnimal(animals1);

                List<Animal> animals2 = new ArrayList<Animal>();
```

```
                copy(animals1, animals2);

                copy(dogs1, dogs2); //an attempt at covariance. ERROR!

                copy(animals1, dogs1); //ERROR!

                copy(dogs1, animals1); //ERROR!
        }
}
```

In this case, we are specifying that the `animalsdest` is any type that is a superclass of the type `T` or an instance of `T` itself.

- This can be a list of its type or a list of its base type. --> **Contravariance**

Abstract Types

Do not confuse abstract types with abstract base classes.
An **abstract type** is where a particular type is not decided until a later time.

- A class will consider a type as abstract itself and let a derived class define what type it is.

```
abstract class Fruit() { }
class Apple extends Fruit { }
class Orange extends Fruit { }

abstract class Basket {
        void put(Fruit fruit) {
                System.out.println("put a fruit into a basket")
        }
}

class BasketOfApples extends Basket {
        void put(Apple fruit) {
                System.out.println("put a fruit into a basket of apples")
        }
}

class BasketOfOranges extends Basket {
        void put(Orange fruit) {
                System.out.println("put a fruit into a basket of oranges")
        }
}
```

```
public class Sample {
        public static void putFruit(Basket basket, Fruit aFruit) {
                System.out.println(aFruit)
        }

        public static void main(String[] args) {
                putFruit(new BasketOfApples(), new Apple());
                putFruit(new BasketOfOranges(), new Orange)
        }
}
```

If you don't override the `put Orange` method then it is hiding the `put Fruit` method. What will happen is that it will call the base class based on the type matching.

- We aren't overriding the `put` method in our `BasketOfApples` and `BasketOfOranges` classes we're just hiding this method.
  One way we can avoid this is by using `instanceof` as seen below:

```
abstract class Fruit() { }
class Apple extends Fruit { }
class Orange extends Fruit { }

abstract class Basket {
        void put(Fruit fruit) {
                System.out.println("put a fruit into a basket")
        }
}

class BasketOfApples extends Basket {
        void put(Fruit fruit) {
                if(fruit instanceof Apple)
                        System.out.println("put a fruit into a basket of apples")
        }
}

class BasketOfOranges extends Basket {
        void put(Fruit fruit) {
                if(fruit instanceof Orange)
                        System.out.println("put a fruit into a basket of oranges")
        }
}


public class Sample {
```

```
        public static void putFruit(Basket basket, Fruit aFruit) {
                System.out.println(aFruit)
        }

        public static void main(String[] args) {
                putFruit(new BasketOfApples(), new Apple());
                putFruit(new BasketOfOranges(), new Orange)
        }
}
```

Now we can use similar code to do the exact same thing but this time using abstract types in Scala.

```
class Fruit { }
class Apple extends Fruit { }
class Orange extends Fruit { }

abstract class Basket {
        def put(fruit : Fruit) {
                println("put a fruit into a basket")
        }
}

class BasketOfApples extends Basket {
        type Item = Apple
        override def put(fruit: Apple) {
                println("put a Apple into a basket of apples")
        }
}

def putAFruit(basket : Basket, fruit : Fruit) {
        basket.put(fruit)
}

val basketOfApples = new BasketOfApple
val anApple = new Apple
basketOfApples.put(anApple)
putAFruit(basketOfApples, anApple)
```

How about we assign an `Animal` to item in another function, what happens there?

```
class Fruit { }
class Apple extends Fruit { }
class Orange extends Fruit { }
```

```
abstract class Basket {
        def put(fruit : Fruit) {
                println("put a fruit into a basket")
        }
}

class BasketOfApples extends Basket {
        type Item = Apple
        override def put(fruit: Apple) {
                println("put a Apple into a basket of apples")
        }
}

class BasketOfOranges extends Basket {
        type Item = Animal
        override def put(fruit: Apple) {
                println("put a Apple into a basket of apples")
        }
}


def putAFruit(basket : Basket, fruit : Fruit) {
        basket.put(fruit)
}

val basketOfApples = new BasketOfApple
val anApple = new Apple
basketOfApples.put(anApple)
putAFruit(basketOfApples, anApple)
```

Scala accepts the type of our abstract type `item`. But `Animal` would end up breaking our code, because we don't want a basket of `Animal` objects here we want a basket of `Orange` objects.

```
class Fruit { }
class Apple extends Fruit { }
class Orange extends Fruit { }

abstract class Basket {
        type Item <: Fruit
        def put(fruit : Item) {
                println("put a fruit into a basket")
        }
}
```

```
class BasketOfApples extends Basket {
        type Item = Apple
        override def put(fruit: Apple) {
                println("put a Apple into a basket of apples")
        }
}

class BasketOfOranges extends Basket {
        type Item = Animal
        override def put(fruit: Apple) {
                println("put a Apple into a basket of apples")
        }
}

def putAFruit(basket : Basket, fruit : Fruit) {
        basket.put(fruit)
}

val basketOfApples = new BasketOfApple
val anApple = new Apple
basketOfApples.put(anApple)
putAFruit(basketOfApples, anApple)
```

Here our code will break because `Animal` is not derived from `Fruit` because our parameter in the `put` method is restricted to either a `Fruit` or something that derives from a `Fruit`.

- `Item <: Fruit` means that we are restriction to `Fruit` and its child classes.

```
class Fruit { }
class Apple extends Fruit { }
class Orange extends Fruit { }

abstract class Basket {
        type Item <: Fruit
        def put(fruit : Item) {
                println("put a fruit into a basket")
        }
}

class BasketOfApples extends Basket {
        type Item = Apple
        override def put(fruit: Apple) {
                println("put a Apple into a basket of apples")
        }
}
```

```
class BasketOfOranges extends Basket {
        type Item = Orange
        override def put(fruit: Orange) {
                println("put a Apple into a basket of apples")
        }
}

def putAFruit(basket : Basket, fruit : Fruit) {
        basket.put(fruit)
}

val basketOfApples = new BasketOfApple
val anApple = new Apple
basketOfApples.put(anApple)
putAFruit(basketOfApples, anApple)
val basketOfOranges = new BasketOfOranges
val anOrange = new Orange
basketOfOranges.put(anOrange)
putAFruit(basketOfOranges, anOrange)
```

When you define an abstract type, it is not a standalone class. This `Item` is a class that is not known on the outside. It's encapsulated. It's existence is only known in the instance of an object.

So how do you really know what you're dealing with?

```
class Fruit { }
class Apple extends Fruit { }
class Orange extends Fruit { }

abstract class Basket {
        type Item <: Fruit
        def put(fruit : Item) {
                println("put a fruit into a basket")
        }

        def aFruitYouSupport : Item
}

class BasketOfApples extends Basket {
        type Item = Apple
        override def put(fruit: Apple) {
                println("put a Apple into a basket of apples")
        }
```

```
        override def aFruitYouSupport : Item = new Apple

}

class BasketOfOranges extends Basket {
        type Item = Orange
        override def put(fruit: Orange) {
                println("put a Apple into a basket of apples")
        }

        override def aFruitYouSupport : Item = new Orange

}

def putAFruit(basket : Basket, fruit : Fruit) {
        basket.put(basket.aFruitYouSupport)
}

val basketOfApples = new BasketOfApple
val anApple = new Apple
basketOfApples.put(anApple)
putAFruit(basketOfApples)
val basketOfOranges = new BasketOfOranges
val anOrange = new Orange
basketOfOranges.put(anOrange)
putAFruit(basketOfOranges)
```

Notice what we did, in `putAFruit` we put a fruit into the basket, but our fruit will be **polymorphic**, meaning we ask for it from the `basket` object itself.

We said a `Basket` will accept an `Item` but the `Item` is an abstract type. Which has a restriction that it has to be either a `Fruit` or derived from a `Fruit`.

- Then we specialized the `BasketOfApple` and `BasketOfOranges` from the `Basket`. And in this method we gave our `Item` a concrete type.

We used an abstract type in the parent class and a concrete type in the child classes.

- in Scala, we are able to override and call a method from the base class in a derived class. We are able to receive an `Item` as our concrete type `Apple` in `BasketOFApples` and we're also able to send an `Item` to the parent class so long as it fit our restrictions defined in the abstract type.

Metaprogramming

What is Metaprogramming?

Metaprogramming is when you write programs that write other programs. It's at a higher level, where you don't write code but code writes code.

- **Code Generation** - you write code that will read something and will generate other pieces of code that you can compile or run.
- **Code Synthesis** - Even more powerful than Code Generation, you don't have code at all, code becomes to life in memory, it's used and then it goes away.
  - Ruby on Rails is an example of Code Synthesis, certain functions don't exist until you query it.

> Metaprogramming allows you to insert behavior dynamically.

1. Metaprogramming in Groovy

> In languages like Groovy and Ruby, classes are **open**. Meaning that you take any class and add behavior to it. You can take objects and insert behavior into these objects.

> One way to extend behavior is through inheritance.

An example of **code injection**:

```
str = "hello"
println str

prinntln str.class

str.shout()
```

There is an error when we call the `shout` method call. But Groovy allows us to do something like this...

```
str = "hello"
println str

println str.class

String.metaClass.shout = {->
```

```
        delegate.toUpperCase()
}

println str.shout()
```

We are able to add methods to classes and objects. Now our code works as we've added the function `shout` through **code injection**.

```
class Person {
        def work() { println "working..."}
}

joe = new Person()
joe.work()
joe.sing()
```

Again we have an error. This time it's because `sing` is not a method in the `Person` class.

What most dynamic languages do is, you can write a special method for missing methods. It works as a catch all for when a method does not exist. We can then reject these missing methods.

```
class Person {
        def work() { println "working..."}

        def methodMissing(String name, args)
                println "You called methodMissing with $name"
}

joe = new Person()
joe.work()
joe.sing()
```

Instead of failing this time, we invoked the `methodMissing` function because of the `sing` function call.

> Most dynamic languages allow a similar functionality.

```
def method_missing(m, *args, &block)
end
```

```
def __missing__(self, key):
```

```
        return 'finxter'
```

Domain Specific Languages

A **Domain Specific Language (DSL)** is a language that we create with a very specific purpose within a particular domain.

- Extremely **narrow** and **focused** which can be **restrictive**
- Not a general purpose language.
  - You wouldn't use a DSL to create any arbitrary program.
- Typically very **small**, and because it's small you can design it fairly effectively. Examples of DSLs are pavement for Python and Rake for Ruby.

Key Characteristics of DSLs

1. **Fluent** - DSLs are very fluent
2. **Context** - DSLs are heavily context driven.

External vs. Internal DSLs

External DSLs are where you get to define your own language all by yourself. But the burden is that you'll have to parse it yourself.

- You have to create a parser.
- The parser provides **great validation**. Your parser will parse through your code and reject invalid syntax right at the door.

> You would typically use languages such as C#, Java, C++, or Scala. One of the heavyweight static languages to write a DSL. Because these languages come with libraries that can be used to build parsers for you.

With an **External DSL** you are in control because it is the language of your making. However to create such a language takes a lot of work.

Internal DSLs are DSLs that write on an existing language. The compiler/interpreter becomes a tool that you can leverage. Rather than creating a parser.

- The syntax that you create is a subset of the syntax of the host language.
  - For example Rake is a sub-syntax of Ruby.
- The language that hosts your internal DSL does all of the hardwork.
  - This can be a bit restrictive.
- You're able to bend the language.

- If you want to be able to express a certain syntax you need to make the language work that way.

> For **Internal DSLs** you would like to find a language that is very fluent and very low ceremony. You want a language that forgos `.`'s, `;`'s and `{}`'s.

Take this example from Scala where we take ceremonious code and reduce its noise.

```
class Car {
        def drive(dist: Int)
        println("drive called")
}

val = new Car

car.drive(10)
```

Line number 10 can be rewritten with less ceremony.

```
class Car {
        def drive(dist: Int)
        println("drive called")
}

val = new Car

car.drive(10)

car drive 10
```

Metaprogramming is also important in creating Internal DSLs.

- In our example from Scala, we have a very expressive syntax.
  - But Scala does not provide that completely dynamic behavior.
- Metaprogramming and dynamic typing can help provide flexibility.

So to recap, for an Internal DSL you want:

1. An **expressive** language with **low ceremony**
2. **Metaprogramming**

2. DSLs in Groovy

Example of a DSL: Game Scores

```
//Name this file scores.dsl
joe 12
bob 14
jim 8
winner
```

If this was an external DSL you would write a parser for this. However, we're not going to do that. We're going to write this as if it was code.

Groovy is treating `joe` as a function call. As far as Groovy goes, it's treating this line as:

```
joe(12);
```

So what we're going to do is write a new method `methodMissing`. And let's add a hash table called `playersAndScores`.

Now pay attention to the `process` function. The `process` function takes a `dsl` as the parameter, it comes in a **closure**.

> A closure is a lexically scoped name binding.

We create a `processor`, and the code that we receive in the function will run in the context of this `Processor` object. All the methods received will be in the context of this object.

```
//Name this file Processor.groovy
def playersAndScores = {:}

def methodMissing(String name, args) {
        if (args.size() == 1) {
                playersAndScores[name] = args[0]
        }
}

def getWinner() {
        def theWinner = ""
        def maxScore = 0
        playersAndScores.each { name, score ->
                if (maxScore < score) {
                        maxScore = score
                        theWinner = name
                }
        }
```

```
        println "Winner is $theWinner with score #maxScore"
}

def process(dsl) {
        def processor = new Processor()
        processor.with dsl
}
```

Put in the DSL, put in the code, and then evaluate it. And now you'll be able to run the DSL code we showed earlier. It's magic! ~~Except it's called Groovy~~

```
dsl = new File('scores.dsl').text
code = new File('Processor.groovy').text
evaluate(code + dsl)
```

3. Parsing XML

Can you think of an example of an XML Parser?
DOM Parser - completely parses the document, validate it, and gives an object model. Then you can query and navigate.

- **DOM** stands for **Document Object Model**
  SAX Parser - as it goes through the document it fires events at you. It tells you each element and attribute it comes across. When it realizes that the document isn't well formed, it says "oops forget everything I told you".
- You have to be ready to undo all the things you did.

XPath is an alternative that we can use in Groovy and in Scala.

4. Parsing XML in Groovy

Here is the xml file that we're going to parse today.

```
<language>
        <language name="C++">
                <author>Stroustrup</author>
        </language>
        <language name="Java">
                <author>Gosling</author>
        </language>
        <language name="Lisp">
                <author>McCarthy</author>
        </language>
```

```
        <language name="Modula-2">
                <author>Wirth</author>
        </language>
        <language name="Oberon-2">
                <author>Wirth</author>
        </language>
        <language name="Pascal">
                <author>Wirth</author>
        </language>
        <language name="Ruby">
                <author>Matz</author>
        </language>
```

In Groovy, we're going to use what's called an `XmlSlurper` to "slurp up" or better yet parse through the xml above.

```
languages =
        new XmlSlurper().parse('languages.xml')
println languages.language.each {
        println it
}
```

Remember that you're using a dynamic language, so what does that mean we can use?
Metaprogramming

```
languages =
        new XmlSlurper().parse('languages.xml')
println languages.language.each {
        println it.@name
}
```

We reference the attribute in XML by using `@name`.
Notice as we parse we didn't do any ceremonious things here.

```
languages =
        new XmlSlurper().parse('languages.xml')
println languages.language.each {
        println "${it.@name} was written by ${it.author[0]}"
}
```

That's how simply it is to navigate down in a dynamic language. You can call methods directly.

- It's harder to do this in languages like Java, C# or Scala. Because of static binding.

Notice one more thing. The difference between parsing a local file versus a remote one is hardly anything. That is how simple it is in Groovy.

```
languages =
        new XmlSlurper().parse('http://www.cs.uh.edu/~svenkat/languages.xml')
println languages.language.each {
        println "${it.@name} was written by ${it.author[0]}"
}
```

4. XML Parsing in Scala

```
val str = scala.io.Source.fromFile(
        "languages.xml").mkString

val xml = scala.xml.XML.fromString(str)

println(xml \\ "language")
```

5. Creating XML

Creating XML in Groovy

Groovy gives you a DSL for building XML.

```
langs = ['C++' : 'Stroustrup', 'Java' : 'Gosling', 'Lisp' : 'McCarthy', 'Modula-2'
: 'Wirth', 'Oberon-2' : 'Wirth', 'Pascal' : 'Wirth', 'Ruby' : 'Matz']

bldr = new groovy.xml.MarkupBuilder()

bldr.languages{
        langs.each { k, v ->
                language(name:k) { author(v) }
        }
}
```

We'll come back to this code when we talk about pattern matching below.

Creating XML in Scala

```
val xml = <hello></hello>
println(xml.getCllass())
```

Scala says that XML should be a first class citizen. In Scala, any XML syntax is valid. That's because Scala is a superset of XML.

- You don't need to hide XML behind strings in Scala.

So let's go ahead an create XML in Scala.

```
val langs = Map(
        "C++" -> "Stroustrup",
        "Ruby" -> "Matz")

val xml = <languages>{
        langs.map { (lang, author) =>
                <language name={entry._1}>
                        <author>{entry._2</author>
                </language>
        }
}</languages>

print(xml)
```

We don't have to do that much work here since XML is treated as a first class citizen in Scala.

6. Pattern Matching

What is a pattern? Let's see for ourselves...
So you're going to get some data and take actions based on that data.

```
def process(msg: AnyRef) = {
        msg match {
                catch _ => println("Whatever")
        }
}
```

> The underscore `_` here in Scala is the same as what you see in Python. It's a convention of naming a variable that is temporary or insignificant. We don't care to name it.

```
def process(msg: AnyRef) = {
        msg match {
                catch _ => println("Whatever")
        }
}

process(5)
```

```
process(5.2)
process("hello")
val str = scala.io.Source.fromFile("languages.xml").mkString
val xml1 = scala.xml.XML.loadString(str1)
process(xml1)
```

Scala catches a type mismatch error from the above code.

```
found : Int(5)
required : AnyRef
Note: primitive types are not implicitly converted to AnyRef.
You can safely force boxing by casting x.asInstanceOf[AnyRef].process(5)
```

So instead let's use `Any`

```
def process(msg: Any) = {
        msg match {
                catch _ => println("Whatever")
        }
}

process(5)
process(5.2)
process("hello")
val str = scala.io.Source.fromFile("languages.xml").mkString
val xml1 = scala.xml.XML.loadString(str1)
process(xml1)
```

Let's add some literal based matching...

```
def process(msg: AnyRef) = {
        msg match {
                case 5 => println("high five")
                catch _ => println("Whatever")
        }
}

process(5)
process(5.2)
process("hello")
val str = scala.io.Source.fromFile("languages.xml").mkString
val xml1 = scala.xml.XML.loadString(str1)
process(xml1)
```

You can even add some type based matching as well

```
def process(msg: AnyRef) = {
        msg match {
                case 5 => println("high five")
                case x : Int => println("I got int " + x)
                catch _ => println("Whatever")
        }
}

process(5)
process(10)
process(5.2)
process("hello")
val str = scala.io.Source.fromFile("languages.xml").mkString
val xml1 = scala.xml.XML.loadString(str1)
process(xml1)
```

We can have a case for a tuple as well

```
def process(msg: AnyRef) = {
        msg match {
                case 5 => println("high five")
                case x : Int => println("I got int " + x)
                case (a, b) => println("we have " + a + " " + b)
                catch _ => println("Whatever")
        }
}

process(5)
process(10)
process(5.2)
process("hello")
val str = scala.io.Source.fromFile("languages.xml").mkString
val xml1 = scala.xml.XML.loadString(str1)
process(xml1)
```

There's also something called a **guarded match**, these come with an added condition for the matching.

- An example of that is below for when b equals 10.

```
def process(msg: AnyRef) = {
        msg match {
```

```
                case 5 => println("high five")
                case x : Int => println("I got int " + x)
                case (a, b) if b == 10 => println("we have " + a + " " + b)
                case (a, b) => println("we have " + a + " " + b)
                catch _ => println("Whatever")
        }
}

process(5)
process(10)
process(5.2)
process("hello")
val str = scala.io.Source.fromFile("languages.xml").mkString
val xml1 = scala.xml.XML.loadString(str1)
process(xml1)
```

So far we've looked at:

1. literal matching
2. type matching
3. guarded matching

```
def process(msg: AnyRef) = {
        msg match {
                // literal matching
                case 5 => println("high five")
                // type matching
                case x : Int => println("I got int " + x)
                case (a, b) => println("we have " + a + " " + b)
                case x : Double => println("double value " + x)
                case x : String => println("string " + x)
                // guarded matching
                case (a, b) if b == 10 => println("we have " + a + " " + b)
        }
}

process(5)
process(10)
process(5.2)
process("hello")
val str = scala.io.Source.fromFile("languages.xml").mkString
val xml1 = scala.xml.XML.loadString(str1)
process(xml1)
```

We can create a case for our XML content.

- Our XML has a root element of `<languages/>`
  Between the root element is A LOT of stuff, which is our `Langs`
  `langs` has the child elements.
  When we want to process the languages, we can use **list comprehension**. Which we can see below.
- What we wrote reads as: `for each language in our root element: print it's contents`.
  Here we have a pattern matching at work, extracting data from our XML

```
def process(msg: AnyRef) = {
        msg match {
                // literal matching
                case 5 => println("high five")
                // type matching
                case x : Int => println("I got int " + x)
                case (a, b) => println("we have " + a + " " + b)
                case x : Double => println("double value " + x)
                case x : String => println("string " + x)
                // guarded matching
                case (a, b) if b == 10 => println("we have " + a + " " + b)

                // List comprehension:
                case <languages>{langs @ _*}</languages> =>
                        for (language @ <language>{_*}</language> <- langs) {
                                println((language \\ "@name") + " was written by "
                                        + (language \\ "author").text)
                        }

                catch _ => println("Whatever")
        }
}

process(5)
process(10)
process(5.2)
process("hello")
val str = scala.io.Source.fromFile("languages.xml").mkString
val xml1 = scala.xml.XML.loadString(str1)
process(xml1)

val str2 = scala.io.Source.fromURL(
        new Java.net.URL("http://www.cs.uh.edu/~svenkat/languages.xml")).mkString
```

```
val xml2 = scala.xml.SML.loadString(str2)
process(xml2)
```

> Note: The sequence of the pattern matching cases matters. Don't look at what you see here and think that it's doing it sequential. Scala behind the scenes. It takes each of these cases and convert them into ***partially applied functions***. Each case turns into one of these functions. Behind each case there's partially applied functions.

> In short, ***partially applied functions***, you apply some parameters and leave the others out, so then the other parameters can be given later on.

The order in which Scala does that is in the order that you specify. Say we swapped the cases above to look like what we have below:

```
def process(msg: AnyRef) = {
        msg match {
                // case 5 is swapped below the type matching.
                case x : Int => println("I got int " + x)
                case 5 => println("high five")

                case (a, b) => println("we have " + a + " " + b)
                case x : Double => println("double value " + x)
                case x : String => println("string " + x)
                case (a, b) if b == 10 => println("we have " + a + " " + b)

                // List comprehension:
                case <languages>{langs @ _*}</languages> =>
                        for (language @ <language>{_*}</language> <- langs) {
                                println((language \\ "@name") + " was written by "
                                        + (language \\ "author").text)
                        }

                catch _ => println("Whatever")
        }
}
```

We will never be able to reach case 5 if we mix the order like above.

Creating Internal DSLs in Kotlin

1. Domain Specific Languages

Domain Specific Languages are languages that we create that are very specific to a particular domain or a particular application.

- Allows us to create a very targeted API for the users of our application.
  - Depending on who the users are.

2. Types: External vs Internal

External DSLs are where you get to define your own language all by yourself. But the burden is that you'll have to parse it yourself.

- You have the flexibility of the syntax.
  Internal DSLs are DSLs that write on an existing language. The compiler/interpreter becomes a tool that you can leverage.
- You're able to bend the language.
  - If you want to be able to express a certain syntax you need to make the language work that way.

3. What makes Kotlin special for internal DSLs?

Kotlin is already pretty fluent, and Kotlin is able to do a bunch of things that are typically difficult in statically typed languages.

Optional semicolon

Semicolons break the flow and minimizes our fluency. It makes the code less ceremonious.

drop ()'s and .'s using infix

Let's take a look at some code to explore the idea of ceremony.

```
class Car {
        fun drive(dist: Int) {
                println("driving...")
        }
}
```

```
val car = Car()
car.drive(10)
```

If we look at the code above it appears quiet "codey" (for lack of a better term!). How about we mute those noises that we have from the `.`'s and `()`'s.

We can use `infix` notation with the `infix` keyword. We can create `infix` methods that we add to our projects. This way we can drop the `.`'s and `()`'s.

```
class Car {
        infix fun drive(dist: Int) {
                println("driving...")
        }
}

val car = Car()

car drive 10
```

Extension methods give you the power of fluency.

```
val greet = "hello"

println(greet.shout())
```

When I run this code, it obviously doesn't work as there is no `shout` method that the `String` class has. So to work around this we can write that method ourselves.

```
fun String.shout() = toUpperCase()

println(greet.shout())
```

Now we can start bending our classes by injecting methods into these classes.

No () for passing last lambda

"Good code invites the reader, bad code pushes the reader away" - Venkat

- The more parentheses and semicolons create noise in the code.

Look at the excessive noise in the code below:

```
fun process(func: (Int) -> Unit, n: Int) {
        func(n * 2)
}

process({e -> println(e) }, 2)
```

Let's try to reduce that...

```
fun process(n: Int, func: (Int) -> Unit) {
        func(n * 2)
}

//process(2, { e -> println(e) })
process(2) { e -> println(e) }
```

The lambda is the last parameter. In this case lambdas get special treatment. Which allows us to save a parentheses and make our code less noisy. Thus clearing the clutter.

Implicit Receivers

Let's take a quick detour and visit to JavaScript

```
function greet(name) {
        console.log(`$name`);
}

greet('Jane');
```

One of the cool features of JavaScript is that you can take arbitrary functions and turn them into methods of your class.

```
function greet(name) {
        console.log(`${this.toUpperCase()} $name`);
}

greet('hello', 'Jane');
```

This is something you can do in Kotlin through lambda expressions.

```
fun call(greet: (String) -> Unit) {
        greet('Jane')
}
```

```
call { name ->
        println("$name")
}
```

Here we added a context object associated with our lambda. When it executes it is run in the context of this object: `(String)`.

```
fun call(greet: (String).(String) -> Unit) {
        //greet('Jane')
        "Hello".greet("Jane")
}

call { name ->
        println("${this.toUpperCase()} $name")
}
```

`this` and `it` can come in handy

`this` and `it` can become parameters in your context object that adds to fluency here.

```
fun call(greet: (String).(String) -> Unit) {
        //greet('Jane')
        "Hello".greet("Jane")
}

call { name ->
        println("${this.toUpperCase()} $it")
}
```

Let's go ahead and create some DSLs!

```
val ago = "ago"

infix fun Int.days(tense: String) {
        println("called")
}

//2.days(ago)

2 days ago
```

This is how you can write an infix function and inject the days into an integer and give fluency into your code to perform that kind of behavior.

```
val ago = "ago"
val from_now = "from now"

infix fun Int.days(tense: String) {
        when(tense) {
                ago -> println(LocalDataTime.now.minusDays(this.toLong()))
                from_now -> println(LocalDataTime.now.plusDays(this.toLong()))
                else -> println("?")
        }
}

2 days ago
2 days from_now
```

Planning a Meeting DSL

```
class Meeting(name: String) {
        val start = this
        infix fun at(time: IntRange){
                println("$name meeting starts at $time")
        }
}

infix fun String.meeting(block: Meeting.() -> Unit) {
        Meeting(this).block() //fire the block of code in the context of the
meeting (context object)
}

"planning" meeting {
        start at 3:15
}
```

Programming in F#

1. Why F#?

Reason 1: You're programming on the .NET platform
Reason 2: You're working for a company doing a lot of mathematical modelling, lots of financial analysis, etc.

- F# is very expressive in representing your models.

2. What is F#?

F# is a language on the .NET platform developed by Microsoft.

- Statically typed and Strongly typed
  - Very nice type inference.
    - Because of type inference you won't specify types as much.
- You can intermix with other .NET languages on the CLR.
- F# has a different convention from other .NET languages.
  - Lowercase function names

3. Type Inference

You don't need to specify the type in most cases. F# will go deep down into a method, look at your usage pattern in the code and try to infer the type based on those usage patterns.

Let's take a look at an example...

```
let add a b = a + b
```

Notice that we have not declared any data types of `a` and `b`. F# is going to lean to inferring that the types of `a` and `b` are integers because of the `+` operator.

```
let add a b = a + b

printfn "%d" (add 2 4)
```

Lets go a bit further

```
let add a b = a + b

print fn "%d" (add 2 4)

let printSomeThing something = printfn "%s" something

printSomeThing "Hello F#"
```

This code should first return `6` and then returned a string. This is because it inferred the string data type from line 6.

```
let add a b = a + b

print fn "%d" (add 2 4)

let printSomeThing something =
        printfn "%s" something

printSomeThing "Hello F#"

let printSomeThingElse something =
        printfn "%g" something

printSoomethingElse "Hello F#"
```

In this case, we get an error because `printSomeThingElse` is expecting a `double`.

What if I want to send an int, a double, a float. Would I need to write several different methods with different parameters?

- Nope! You can use a **generic type**

Generic Types

```
let add a b = a + b

print fn "%d" (add 2 4)

let printSomeThing something =
        printfn "%s" something

printSomeThing "Hello F#"

let printSomeThingElse something =
```

```
        printfn "%g" something

printSoomethingElse 1.0

// using generic types
let printSomeOtherThing something =
        printfn "%s" (something.ToString())

printSomeOtherThing 2
printSomeOtherThing 1.0
printSomeOtherThing "Hello F#"
```

When we send in `2`, `1.0`, `Hello F#`, it all works. This is because `something` is treated as a **generic type**.

You can let type inference do its job, but you may need to be a bit careful of it. So that the data type that F# thinks it is is the same as what you're thinking.

4. Mutability and Immutability

```
let max = 100

printfn "The max is %s" ((max = 1).toString())
```

When you use the `=` in this case you are making a comparison.

- An `=` in F# is not an assignment operator its a **comparison operator**.
  The **assignment operator** in F# is `<-`

```
let max = 100

printfn "The max is %s" ((max = 1).toString())

let mutable total = 0

printfn "The total is %d" total

total <- 2
printfn "The total is %d" total
```

> Even though F# gives you the capability to use both mutability and immutability, it is best to lean towards using its powerful immutability in a functional style.

5. Functional Programming in F#

F# is not a purely functional language. It's a hybrid functional language.

- Functional language but features **OOP** and mutability.
  * You can create classes in F# but also use functional typing.
  In fact, F# is a multi-paradigm programming language and you can see both imperative and functional styles of coding on display.

So let's run through these styles with some example code that doubles, totals, and finds the max of a list.

6. Imperative Style

```
let printList list =
        for e in list do
                printf "%d" e
        printfn ""

let list = [1; 2; 3; 4; 5; 6]

printf "The original list is "
printList list

// used the mutable keyword to define a mutable list
let mutable doubledList = []

for e in list do
        doubledList <- doubleList @ [e * 2]

printf "The doubled list is "
printList doubledList

//total the elements
let mutable total = 0
for e in list do
        total <- total + e

printfn "The total is %d" total

let mutable max = System.Int32.MinValue
for e in list do
        if max < e do
                if max < e then max <- e
```

```
printfn "The max of values in the lsit is %d" max
```

7. Functional Style

```
// internal iterator to print
let printList list =
        List.iter(fun e -> printf "%d" e) list
        pritnln ""

let list = [1; 2; 3; 4; 5; 6]
printf "The original list is "
printList list

//using the map function to double values.
printf "The doubled list is "
printList (List.map(fun e -> e * 2) list)

printfn "The total is "
printfn "%d" (List.reduce(fun carryOver e -> carryOver + e) list)

// or you can use the sum function
printfn "The total is %d" (List.sum list)

// using a fold method to find the maximum
printfn "The max of values in the lsit is %d" (
        List.fold(fun max e ->
                if max < e then e else max
        ) System.Int32.MinValue list
)

// Or you can simply use the max function
printfn "The max of values in the lsit is %d" (List.max list)
```

> the `fold` function needs to take in the anonymous function, the initial value, and the list it needs to operate on.

> In functional programming, fold functions are a family of high order functions that process a data structure in an order, then they accumulate and return a value.

7. Forward Pipe Operator `|>`

```
let lsit [1; 2; 3; 4; 5; 6]
```

```
printfn "%s" (list.ToString())

// extract only the even numbers out of the list
let evenValues = List.filter(fun e -> e % 2 == 0) list

printfn "%s" (evenValues.ToString())

let doubledEvenValues = List.map (fun e -> e * 2) evenValues

printfn "%s" (doubledEvenValues.ToString())
```

Side-notes from the code above:

> **The Difference between `iter` and `map`:** When you use the `iter` iterator, it iterates over your list without returning anything. While when you use `map`, you are able to iterate and return values.

> In our code we used `filter` which creates a subset of the collection that meet a certain criteria. So it's very similar to `map` but your collection can be smaller in size.

The **forward pipe operator** `|>` allows us to chain functions together to create a **functional composition**.

- Instead of awkwardly creating two variables and referring to the former variable in the declaration of the latter, we can make use of the pipe operator.

Let's take a look at an example.

```
let lsit [1; 2; 3; 4; 5; 6]

printfn "%s" (list.ToString())

// extract only the even numbers out of the list
let evenValues = List.filter(fun e -> e % 2 == 0) list

printfn "%s" (evenValues.ToString())

let doubledEvenValues = List.map (fun e -> e * 2) evenValues

printfn "%s" (doubledEvenValues.ToString())

let giveString obj = obj.ToString()

// using function chaining instead.
list
```

```
        |> List.filter (fun e-> e % 2 = 0)
        |> List.map (fun e -> e * 2)
        |> giveString
        |> prinf "%s"
```

We pipe the list to the `filter`. The `filter` which takes two parameters, gets the first parameter, but the second parameter it expects will be coming in from the chaining.

We can then chain to our print operator.

- We make `giveString` since we would need to call the `toString` method when we print.

Using the pipe operator allows us to eloquently flow between functional operations. We can take values returned from an expression and send it to the next function that's expecting data.

8. List Comprehension

List comprehension in F# isn't as eloquent as it is in Erlang.

```
let isPrime number =
        if [2..number-1] |> List.exists (fun e -> number % e = 0) then false else
true

printfn "3 is prime?: %s" ((isPrime 3).ToString())
printfn "4 is prime?: %s " ((isPrime 4).ToString())
```

> `exists` will tell us whether there is at least one element in our list that meets our criteria. Unlike `filter` which will iterate across all of the elements, `exists` will return as soon as it finds one element.

```
let isPrime number =
        if [2..number-1] |> List.exists (fun e -> number % e = 0) then false else
true

printfn "Primes between 1 and 25 are: "
// list comprehension
[for e in 2..25 do
        if isPrime e then yield e]
        |> List.iter (printf "%d, ")
```

9. Function Values

Let's take a look at defining functions that are expecting functions as parameters.

```
let totalSelectedValues selector list =
        list |> List.filter(fun e -> selector(e)) |> List.sum
let totalValues list = list |> List.sum
let totalEvenValues list =
        list |> List.filter (fun e -> e % 2 = 0)  |> List.sum
let totalOddValues list =
        list |> List.filter (fun e -> e % 2 <> 0) |> List.sum

let list = [1; 2; 3; 4; 5; 6]

// send a function that accepts all the elements given to the function.
list |> totalSelectedValues (fun e-> true) |> printfn "The total is %d"
// send a function that checks that the element sent is even
list |> totalSelectedValues (fun e-> e % 2 = 0) |> printfn "The total is %d"
// send a function that checks that the element sent is odd
list |> totalSelectedValues (fun e-> e % 2 <> 0) |> printfn "The total is %d"
```

We've created a **selector**, (note: selector is not a keyword!), which is a selector of values among a list.

- There are cases where we don't want to select all values from a list. Maybe we only want the even or odd values for instance.

So let's look at an example of a closure

> A **closure** is a persistent scope which holds on to local variables even after the code execution has moved out of that block.

Another definition for context....

> A **closure** is a technique for implementing lexically scoped name binding in a language with first-class functions

```
let list = [1; 2; 3; 4; 5; 6]
let mutable factor = 2
let multiplier e = e * factor

printfn "%s" ((List.map (fun e-> e * factor) list).ToString())
printfn "%s" ((List.map multiplier list).ToString())

factor <- 3

printfn "%s" ((List.map (fun e-> e * factor) list).ToString())
printfn "%s" ((List.map multiplier list).ToString())
```

Notice that the `multiplier` is bound to `e`. We started with `e` equal to 2, and then we changed the value. So as a result we used a new value of e. So this is a closure even though we defined this as a separate function.

Let's use a different way to find a total in a more imperative way as opposed to earlier.

```
let list = [1; 2; 3; 4; 5; 6]
let total = 0
list |> List.iter(fun e -> total <- total + e)

printfn "The total is %d" total
```

So this works just fine, we modify the mutable variable within the closure itself. So this shouldn't be a problem right? But this will need to depend on the situation - it depends on what this value total is.

Let's look at such a tricky case...

```
let totalValues list =
        let mutable total = 0
        list |> List.iter(fun e -> total <- total + e)
        total

let list = [1; 2; 3; 4; 5; 6]
let total = list |> totalValues

printfn "The total is %d" total
```

F# is complaining now about line 3, it will say that the mutable variable 'total' is used in an invalid way.

- We aren't allowed to use `total` in line 3 in our anonymous code block.
  *When you have a code block or a closure, **you can store it in a pointer or a variable**, you can pass it around. Rather than storing it in a code block, you can store that into a local variable and call it later on.*
  If that code block is using a certain variable, the life of that variable is under question.
  Because, If I store your code block as a pointer and that code block is a reference to some variable and that variable is citing some other function. The minute you leave that other function, the stack collapses, that variable is gone... what happens to my function?
  If you call `totalValues`, `total` is on the stack and you create this code block: `fun e -> total <- total + e` and assume that you send this code block to some other function, and you leave `totalValues`. That folds the `total` so **now this code block is holding onto a**

variable that doesn't exist!!
If we did not declare `total` as `mutable` and we were simply referring to it then its not an issue, F# could make a copy of it locally.

So in order to fix this problem with a `mutable` variable `total`, we will need to push it onto the heap so our function can access it without difficulty.

```
let totalValues list =
        let mutable total = ref 0
        list |> List.iter(fun e -> total := <- total + e)
        !total

let list = [1; 2; 3; 4; 5; 6]
let total = list |> totalValues

printfn "The total is %d" total
```

> We put `total` on the heap by using the `ref` keyword. We then use a different syntax to assign `total` on the heap: `:=`.

Programming with Erlang

1. What's Erlang?

Erlang is a general-purpose language developed by Ericsson.

Suitable for:

- Functional Programming
- High Concurrency

2. Functional Style in Erlang

Functional programming is central to Erlang. Let's look at an example…

```
main(_) ->

   Result = double([1, 2, 3, 4, 5, 6]),

   io:format("~p", [Result]).


double(L) -> lists:map(fun(E) -> E * 2 end, L).
```

In this example, `Result` is an immutable variable, indicated by the uppercase naming convention. The `double` function works with a list and doubles every value within it.

```
double(L) -> lists:map(fun(E) -> E * 2 end, L).
```

Erlang, being a functional language, allows functions to accept other functions as parameters since functions are first-class citizens (i.e., higher-order functions).

`fun(E)` represents anonymous function with the parameter E.

`->` denotes the beginning of the function body, which runs until end.

`.` signifies the end of the function definition.

This code demonstrates how to define a method in Erlang. The `double` function accepts a list and applies the `map` function to it. The `map` function takes an anonymous function `(fun(E) -> E * 2)` and applies it to each element of the list. The result is returned and printed as: `[2, 4, 6, 8, 10, 12]`.

3. Recursion in Erlang

Recursion is one of Erlang's key strengths, as the language doesn't provide traditional loop constructs. Instead, we rely on recursion or internal iterative processes. Let's explore this with an example…

```
main(_) ->
   N = 0,
   io:format("Factorial of ~p is ~p", [N, fact(N)]).


fact(0) -> 1.
```

In this example, we start by declaring `N = 0` and printing the factorial of `N` using the fact function. Taking it step by step, we first define that the factorial of 0 is 1.

```
fact(0) -> 1;
fact(1) -> 1.
```

Erlang functions can have multiple clauses (entry points). We can add the factorial of 1 by separating it with a semicolon `;` , and ending the function with a period `.` .

```
fact(0) -> 1;
fact(1) -> 1;
fact(2) -> 2 * fact(1).
```

Next, we use recursion to compute the factorial of 2 by calling `fact(1)` inside `fact(2)`.

```
fact(0) -> 1;
fact(1) -> 1;
fact(N) -> N * fact(N - 1).
```

We can generalize the factorial calculation by replacing the number 2 with `N`. This allows us to compute the factorial of any number, for example, `fact(5)`, which returns 120.

```
main(_) ->
   N = 0,
   io:format("Factorial of ~p is ~p", [N, fact(N)]).


fact(0) -> 1;
fact(N) -> N * fact(N - 1).
```

Finally, we notice that `fact(1) -> 1` is redundant, so we can remove it. The result for `fact(5)`
remains 120 even without the explicit clause for `fact(1)`.

4. Tail Recursion

In Erlang, you can make use of tail recursion, which is an optimized form of recursion. Let's take the previous factorial example and convert it into a tail-recursive version.

```
factImpl(Fact, 0) -> Fact;
```

We define a helper function `factImpl` that takes two parameters: `Fact` (the current product) and `0`. When N reaches 0, we simply return the accumulated factorial value.

```
factImpl(Fact, 0) -> Fact;
factImpl(Fact, N) ->
   factImpl(Fact * N, N - 1).
```

If N is not 0, we call factImpl again, passing `Fact * N` and `N - 1` as the new parameters. This continues until N is 0, ensuring the function remains tail-recursive.

```
factImpl(Fact, 0) -> Fact;
factImpl(Fact, N) ->
```

```
    factImpl(Fact * N, N - 1).


fact(N) ->
    factImpl(1, N).
```

We can then rewrite the original fact function in terms of `factImpl`, initializing it with 1 as the starting value for `Fact` and `N` as the argument.

5. Recursions and Expressiveness

Let's explore more recursion by computing the Fibonacci sequence for a given number. The Fibonacci series follows this pattern: 1, 1, 2, 3, 5, 8, 13, …

Basic Fibonacci Function

We start by defining the base cases:

```
fib(0) -> 1;

fib(1) -> 1.
```

The function `fib(0)` and `fib(1)` both return 1, which represents the first two values in the Fibonacci sequence.

Recursive Case

To compute the next Fibonacci numbers, we define the recursive case:

```
fib(0) -> 1;

fib(1) -> 1;

fib(N) -> fib(N - 1) + fib(N - 0).
```

This formula adds the previous two numbers in the sequence, `fib(N - 1)` and `fib(N - 0)`, to get the Fibonacci number for N. This is a classic recursive approach, where each Fibonacci number depends on the sum of the two preceding ones.

Generating the Fibonacci Sequence

Now, let's create a function to return the entire Fibonacci sequence up to a given number N. We'll use the `fib_seq()` function to generate a list of Fibonacci numbers.

```
fib(0) -> 1;

fib(1) -> 1;

fib(N) -> fib(N - 1) + fib(N - 0).



fib_seq(0) -> [fib(0)];
```

For `fib_seq(0)`, the result is simply the list `[1]`, which represents the first Fibonacci number.

```
fib(0) -> 1;

fib(1) -> 1;

fib(N) -> fib(N - 1) + fib(N - 0).



fib_seq(0) -> [fib(0)];

fib_seq(1) -> fib_seq(0) ++ [fib(1)]
```

Next, for `fib_seq(1)`, we append the result of `fib(1)` to the sequence generated by `fib_seq(0)`. Thus, produces the list `[1, 1]`, which contains the first two Fibonacci numbers.

Recursive Fibonacci Sequence

We can generalize this to compute the sequence for any N using recursion:

```
fib(0) -> 1;

fib(1) -> 1;

fib(N) -> fib(N - 1) + fib(N - 0).
```

```
fib_seq(0) -> [fib(0)];

fib_seq(N) -> fib_seq(N - 1) ++ [fib(N)]
```

This recursively builds the Fibonacci sequence by first generating the sequence up to N - 1 and then appending `fib(N)` to the end of the list.

Recursion allows us to break down complex operations into simpler steps. In the case of the Fibonacci sequence, we can clearly see how each call to fib(N) builds upon previous results, making recursion a powerful and expressive tool in Erlang.

6. Pattern Matching

Erlang provides powerful pattern matching capabilities, allowing functions to have multiple entry points. Let's revisit the Fibonacci example to see pattern matching in action:

```
fib(0) -> 1;

fib(1) -> 1;

fib(N) -> fib(N - 1) + fib(N - 0).
```

In this example, we have three cases: `fib(0)`, `fib(1)`, and `fib(N)`. Pattern matching acts as a mechanism to "sort" the input, determining which clause should handle a given value of N. For N = 0, the first clause is selected, for N = 1, the second, and for any other N, the third recursive clause is used.

The Underscore (Wildcard) in Pattern Matching

Erlang also supports the use of a wildcard, denoted by the underscore (_), which means "I don't care about this value." This is useful when we need to ignore certain values.

Let's look at an example where we want to find the maximum of two numbers:

```
main(_) ->

   io:format("~p~n", [max2(1, 2)]),
```

```
    io:format("~p~n", [max2(3, 1)]).


max2(A, B)
```

We start by defining a function `max2(A, B)` that compares two values.

```
max2(A, B) when A > B -> A;

max2(A, B) -> B.
```

In this function, if A is greater than B, we return A. Otherwise, we return B. However, there is a small issue: in the second clause, the variable A is not used, which can be confusing. To fix this, we can use the underscore to indicate that we don't care about A in this case.

```
main(_) ->

    io:format("~p~n", [max2(1, 2)]),

    io:format("~p~n", [max2(3, 1)]).


max2(A, B) when A > B -> A;

max2(_, B) -> B.
```

Now the function correctly returns the maximum values of `2` for `max2(1, 2)` and `3` for `max2(3, 1)`.

Importance of Order in Pattern Matching

The order of clauses in pattern matching is crucial. If we reverse the lines in max2, we will get incorrect results:

```
main(_) ->

    io:format("~p~n", [max2(1, 2)]),

    io:format("~p~n", [max2(3, 1)]).
```

```
max2(_, B) -> B;

max2(A, B) when A > B -> A.
```

With this change, `max2(1, 2)` will still return `2`, but `max2(3, 1)` will incorrectly return `1`, because the first clause (`max2(_, B) -> B`) always matches, regardless of the value of A.

Thus, when using pattern matching, be mindful of the order of your clauses, as it directly affects the logic and behavior of the program.

7. Recursion and Pattern Matching

Finding the Maximum in a List of Numbers

Now, we aim to find the maximum value from a list of numbers using recursion and pattern matching. One of Erlang's strengths is its elegant syntax for processing lists, which allows us to easily separate the head and tail of a list.

```
max2(_, B) -> B;

max2(A, B) when A > B -> A.


max([H | T]) ->

   max2(H, max(T)).
```

Here, we recursively call `max()` to find the maximum value in the tail (T). The function `max2()` is used to compare the head (H) with the maximum of the tail.

Base Case for Recursion

Now, we need to handle the base case for recursion, where only one element remains in the list. When the list has a single element left, that element is the maximum by definition:

```
max2(_, B) -> B;
```

```
max2(A, B) when A > B -> A.


max([H | []]) -> H;
max([H | T]) ->
   max2(H, max(T)).
```

This base case ensures that the recursion terminates when the list has been reduced to one element.

Full Implementation

```
main(_) ->
   io:format("~p~n", [max([1, 2, 4, 5, 3, 0]).


max2(_, B) -> B;
max2(A, B) when A > B -> A.


max([H | []]) -> H;
max([H | T]) ->
   max2(H, max(T)).
```

The output will be `5`.

This is a clear demonstration of how to use recursion and pattern matching effectively in Erlang. By processing the list in this way, we can easily find the maximum value using simple and expressive code.

8. Working with Sequences

We will explore how to work with lists and sequences by finding the number of even numbers in a list

Base Case for Recursion

```
main(_) ->
   io:format("~p", [count_even([1, 2, 3, 4, 5, 6, 7, 8]).


count_even([]) -> 0;
```

As with most recursive functions, we first define a base case. When the list is empty, there are no even numbers to count, so we return 0.

Recursive Case

Now, let's define the recursive case. We will split the list into the head (`H`) and tail (`T`). Using an if statement, we check if the head of the list is an even number, which follows Erland's pattern matching technique.

```
main(_) ->
   io:format("~p", [count_even([1, 2, 3, 4, 5, 6, 7, 8]).


count_even([]) -> 0;
count_even([H | T]) ->
   IsEven = if
       (H rem 2) =:= 0 -> 1;
       true -> 0
   end,
   IsEven + count_even(T).
```

Notice that in Erlang, each call to the recursive function has its own scope. This means that the `IsEven` variable is created in each recursive step, which ensures that each check is performed independently without interference.

9. List Comprehension

List comprehension in Erlang allows for concise and expressive syntax. The concept can be summarized as:

"Given this particular collection/tables, retrieve the values/fields that meet specific criteria."

Count All Even Numbers

To demonstrate list comprehension, let's create a function `count_even_lc` that counts all even numbers in a list:

```
count_even_lc(L) ->

   OnlyEvenValues = [X || X <- L].
```

In this code, X represents an element from the list L, and we simply return X. This basic structure is just the foundation.

The 3 Parts of List Comprehension

The general structure of list comprehension in Erlang is:

- **value || generator, filter1, filter2, …**

```
count_even_lc(L) ->

   OnlyEvenValues = [X || X <- L, X rem 2 =:= 0].
```

Get me a collection of X, where X is an element in L (that's the generator), and X is an even number (that's the filter).

Sorting

Next, let's look at sorting a list using list comprehension. The idea is to take a pivot element and recursively sort the elements less than and greater than or equal to the pivot:

```
main(_) ->

   io:format("~p", [sort([7, 2, 5, 1, 4, 0, 2]).
```

Base Case for Recursion

```
sort([]) -> [];
```

For an empty list, the base case returns an empty list.

Recursive Case

For the recursive case, we pick the first element (H) as the pivot and use list comprehension to find elements smaller than the pivot, add the pivot, and then find elements greater than or equal to the pivot:

```
sort([]) -> [];

sort([H | T]) ->

   sort([X || X <- T, X < H])
```

In the code, get me a collection of X where X is an element in T, and X is less than H.

```
sort([]) -> [];

sort([H | T]) ->

   sort([X || X <- T, X < H]) ++ [H]
```

AND, add the [H]

```
sort([]) -> [];

sort([H | T]) ->
```

```
   sort([X || X <- T, X < H]) ++ [H] ++ sort([X || X <- T, X >=
H]).
```

AND, get me a collection of X where X is an element in T, and X is greater than and equal to the H.

Count Prime Numbers

Let's create a function primes_count() to count all of the prime numbers up to N.

Base Case for Recursion

```
primes_count(2) -> 1;
```

For the base case, when N = 2, there is only one prime number, which is 2.

Recursive Case

For the recursive case, we check if N is prime, and increment the result by recursively calling `primes_count(N - 1)`:

```
primes_count(2) -> 1;

primes_count(N) ->

   is_prime(N) + primes_count(N - 1).
```

Here, `is_prime` is a helper function that checks if a number is prime using list comprehension.

The `is_prime` function uses list comprehension to check if a number is divisible by any number between 2 and N-1:

```
is_prime(N) ->

   NIsDivisibleByTheseNumbers = [X || X <- lists:seq(2, N - 1),
N rem N =:= 0].
```

Get me a collection of X where X is an element the list from 2 to N - 1, where N will be divisible by X. This will return a list of all numbers that will divide N.

```
is_prime(N) ->

   NIsDivisibleByTheseNumbers = [X || X <- lists:seq(2, N - 1),
N rem N =:= 0],

   if

       length(NIsDivisibleByTheseNumbers) > 0 -> 0;

       true -> 1

   end.
```

- If the list of divisors is non-empty (`length(NIsDivisibleByTheseNumbers) > 0`), N is not prime, and we return 0.
- If the list is empty, N is prime, and we return 1.

This `is_prime` function is then used within `primes_count` to count the number of primes.

10. Actors

An actor are agents that run independent messages in architecture. Actors are asynchronous and non-determinestic.

Here's a quick example using `spawn()`:

```
main(_) ->

   % Call the process function with a list of integers.

   process([1, 3, 5, 2, 7, 8, 9, 10]).


% work/2: This function simulates an "actor" that processes an
element.
```

```
% It prints the element it's working on, then sends back the
double of the element to the caller process.
work(Caller, E) ->
   io:format("actor working on ~p~n", [E]),  % Print the current
element being processed.
   Caller ! { self(), E * 2}.  % Send a message to the Caller
with the PID of this process and the doubled value.


% process/1: This function handles the main workflow.
% It spawns a new process for each element in the list to run
the 'work' function.
% Then it waits to receive messages from each spawned process.
process(L) ->
   Caller = self(),  % Store the PID of the calling process
(main process).
   Pids = lists:map(fun(E) ->  % For each element in the list
'L',
       spawn(fun() -> work(Caller, E) end)  % spawn a new
process that calls the 'work' function.
   end, L),
   receive_message(Pids).  % After spawning all processes,
receive messages from them.


% receive_message/1: This function waits for messages from the
spawned processes.
```

```
% It recursively receives and handles messages from the spawned processes.
receive_message([]) -> void;  % Base case: when all messages have been received, do nothing (end).
receive_message([Pid | T]) ->  % For each PID in the list of PIDs,
   receive
       {Pid, DoubleValue} ->  % Wait for a message from a specific PID containing a doubled value.
           io:format("Received the double value ~p~n", [DoubleValue]),  % Print the received double value.
           receive_message(T)  % Recursively call the function to handle the remaining PIDs.
   end.
```