# Functional Programming

Globant
we are ready

Agenda

# Definition

# Purity

# Immutability

# Function Composition

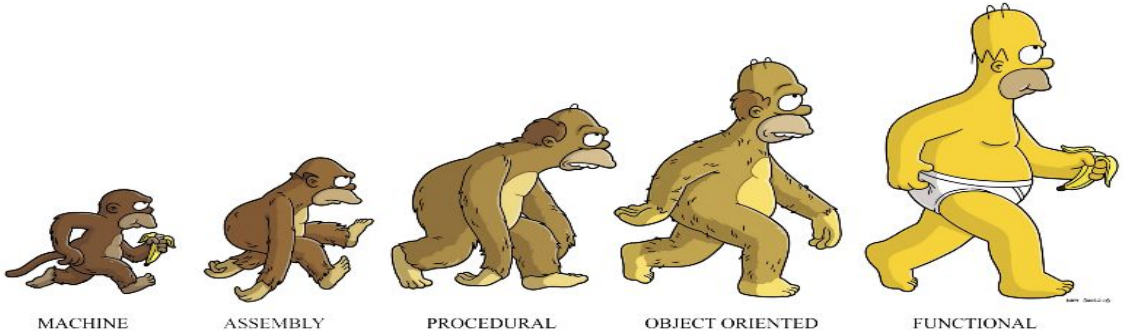
# Recursion

# Higher-Order Functions

# Closures

# Currying

# Common Functional Functions

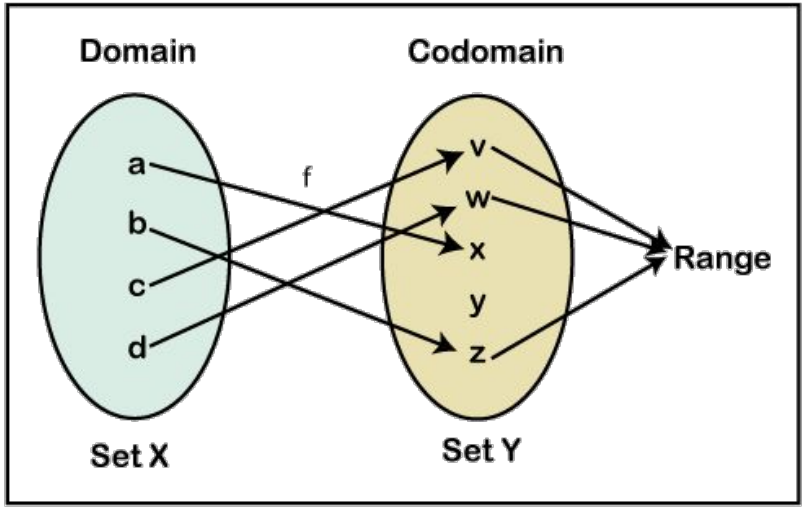



## What is Functional Programming

In computer science, functional programming is a programming paradigm —a style of building the structure and elements of computer programs— that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data.

It is a declarative programming paradigm, which means programming is done with expressions or declarations instead of statements. So It would describe what to do, rather than how to do it.

# Mathematical Functions



# Functional Programming vs Imperative

| Characteristic | Imperative | Functional |
|---|---|---|
| **Programming Style** | Perform a step-by-step task and manage changes in state | Define what the problem is and what data transformations are needed to achieve the solution |
| **State Changes** | Important | Non-existent |
| **Order of Execution** | Important | Not as important |
| **Primary Flow Control** | Loops, conditionals and function calls | Function calls and recursion |
| **Primary Manipulation Unit** | Structures, classes and objects | Function as first-class objects and data sets |

# Purity

## Purity

When Functional Programmers talk of Purity, they are referring to Pure Functions.

They only operate on their input parameters.

```
var z = 10;
function add(x, y) {
    return x + y;
}
```

A Pure Function

```
public class Pure {
  private static int value;

  public static int addOne(int input) { return input + 1; }

  public static void main(String[] args) {
    for(int i = 0; i < 3; i++) {
      System.out.println(addOne(value));
    }
  }
}
```

An Impure Function

```
public class Impure {
  private static int value;

  public static int incrementBy(int number) { return (value += number); }

  public static void main(String[] args) {
    for(int i = 0; i < 3; i++) {
      System.out.println(incrementBy(1));
    }
  }
}
```

# Purity

*Most **useful** Pure Functions must take at least one parameter.*

*All **useful** Pure Functions must return something.*

*Pure Functions will **always** produce the same output given the same inputs.*

*Pure functions have **no** side effects.*

# Immutability

# Immutability

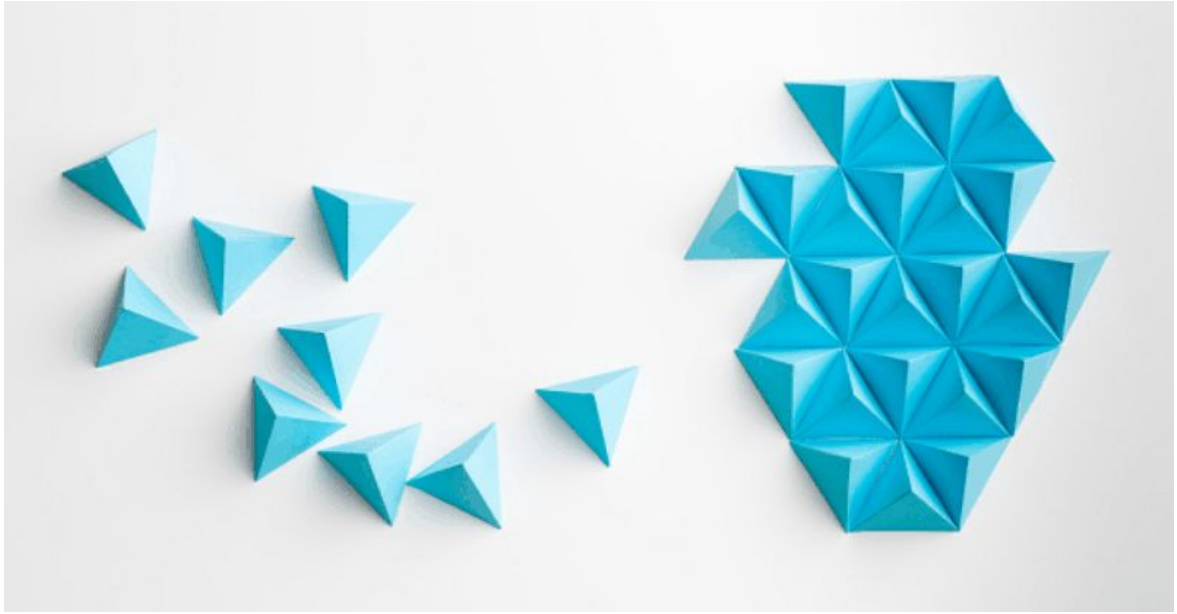
Stored values are still called variables because of history but they are constants, i.e. once *x* takes on a value, it's that value for life.

Functional Programming deals with changes to values in a record by making a copy of the record with the values changed

```
var x = 1;
x = x + 1;
```

```java
public final class ImmutableStudent {
    private final int id;
    private final String name;
    public ImmutableStudent(int id, String name) {
        this.name = name;
        this.id = id;
    }
    public int getId() {
        return id;
    }
    public String getName() {
        return name;
    }
}
```

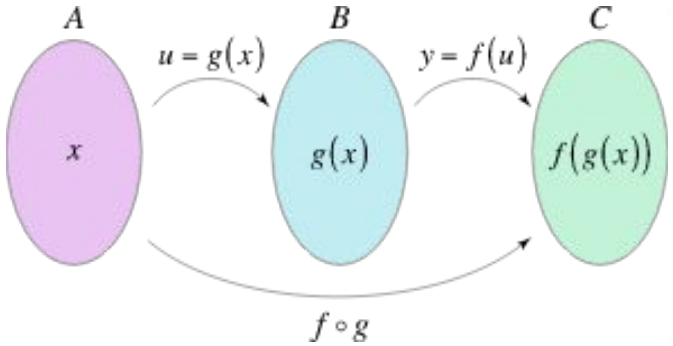
# Function Composition



## Function Composition

Refers to composing complex functions by combining simpler functions.

It's the application of a function to the output of another function.




```
Function<Double, Double> log = (value) -> Math.log(value);
Function<Double, Double> sqrt = (value) -> Math.sqrt(value);
Function<Double, Double> logThenSqrt = sqrt.compose(log);
logger.log(Level.INFO, String.valueOf(logThenSqrt.apply(3.14)));
// Output: 1.06
Function<Double, Double> sqrtThenLog = sqrt.andThen(log);
logger.log(Level.INFO, String.valueOf(sqrtThenLog.apply(3.14)));
// Output: 0.57
```

# Recursion

# Recursion

There are no specific loop constructs like *for*, *while*, *do*, *repeat*, etc.

- *Functional Programming uses recursion to do looping.*
- Non-recursive loops require Mutability, which is bad.

```
public int fib(int n) {
    if(n <= 1) {
        return n;
    } else {
        return fib(n - 1) + fib(n - 2);
    }
}
```

# Higher-Order Functions

# Higher-Order Functions

*In Functional Programming, a function is a first-class citizen of the language. In other words, a function is just another value.*

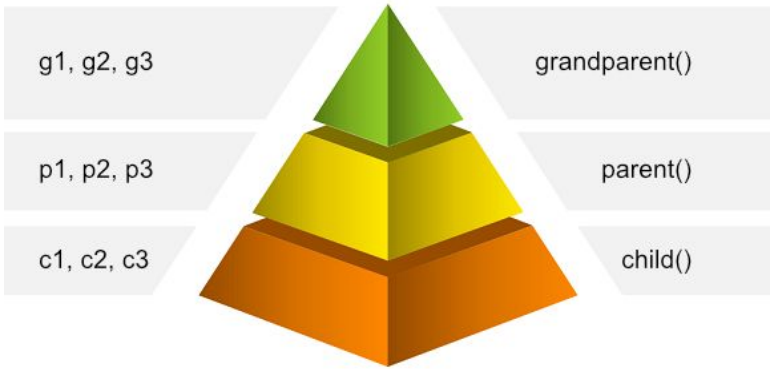
- *Higher-order Functions either take functions as parameters, return functions or both.*

```
function validateAddress(address) {
    if (parseAddress(address))
        console.log('Valid Address');
    else
        console.log('Invalid Address');
}

function validateName(name) {
    if (parseFullName(name))
        console.log('Valid Name');
    else
        console.log('Invalid Name');
}
```

```
function validateValueWithFunc(value, parseFunc, type) {
    if (parseFunc(value))
        console.log('Invalid ' + type);
    else
        console.log('Valid ' + type);
}
```

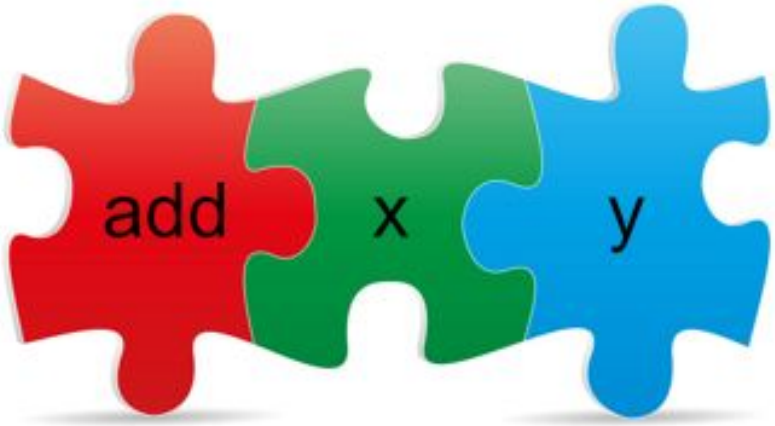
# Closures

- *child* has access to its variables, the *parent's* variables and the *grandParent's* variables.
- The *parent* has access to its variables and *grandParent's* variables.
- The *grandParent* only has access to its variables.




```
function grandParent(g1, g2) {
    var g3 = 3;
    return function parent(p1, p2) {
        var p3 = 33;
        return function child(c1, c2) {
            var c3 = 333;
            return g1 + g2 + g3 + p1 + p2 + p3 + c1 + c2 + c3;
        };
    };
}
```

# Currying



# Currying

*A Curried Function is a function that only takes a single parameter at a time*

```
var greet = function(greeting, name) {
  console.log(greeting + ", " + name);
};
greet("Hello", "Heidi"); //"Hello, Heidi"
```

```
var greetCurried = function(greeting) {
  return function(name) {
    console.log(greeting + ", " + name);
  };
};
```

```
var greetHello = greetCurried("Hello");
greetHello("Heidi"); //"Hello, Heidi"
greetHello("Eddie"); //"Hello, Eddie"
```

# Common Functional Functions

# Map

Don’t iterate over lists. Use map and reduce

- Map takes a function and a collection of items. It makes a new, empty collection, runs the function on each item in the original collection and inserts each return value into the new collection. It returns the new collection

```
name_lengths = map(len, ["Mary", "Isla", "Sam"])

print name_lengths
# => [4, 4, 3]

 squares = map(lambda x: x * x, [0, 1, 2, 3, 4])

 print squares
 # => [0, 1, 4, 9, 16]
```

## Reduce

Reduce takes a function and a collection of items. It returns a value that is created by combining the items.

```
sum = reduce(lambda a, x: a + x, [0, 1, 2, 3, 4])

print sum
# => 10
```

## Filter

Filter takes a list and a predicate (a function, returning true or false) and it return a new list which contains only the members of the original list meeting the predicate (in the same order they appeared)

```
var isOdd = x => x % 2 !== 0;
var numbers = [1, 2, 3, 4, 5];
var oddNumbers = filter(isOdd, numbers);
console.log(oddNumbers); // [1, 3, 5]
```

# Pipelines

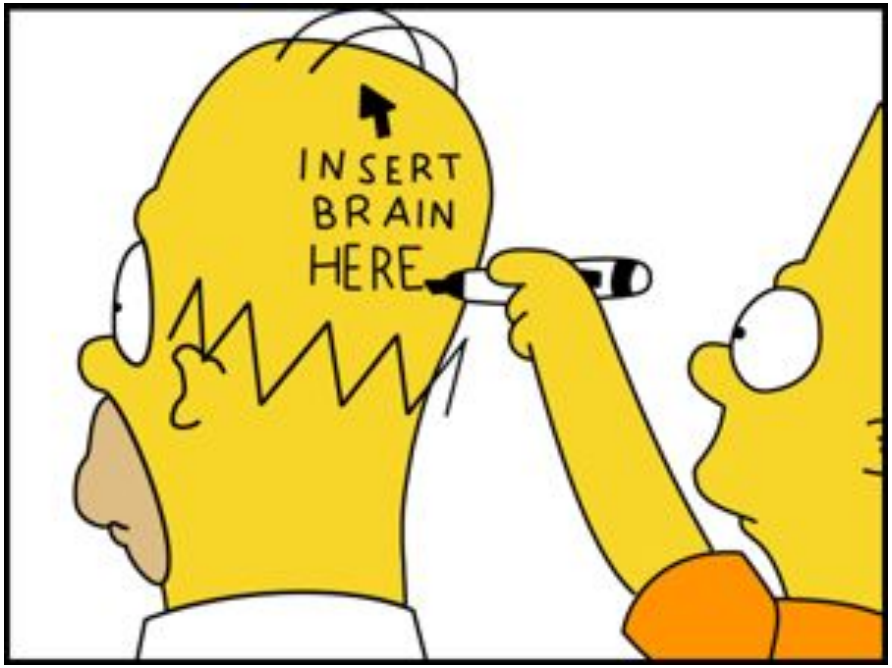
# Pipelines

arranged so that the output of each element is the input of the next

```
bands = [{'name': 'sunset rubdown', 'country': 'UK', 'active': Fals
         {'name': 'women', 'country': 'Germany', 'active': False},
         {'name': 'a silver mt. zion', 'country': 'Spain', 'active'

def format_bands(bands):
    for band in bands:
        band['country'] = 'Canada'
        band['name'] = band['name'].replace('.', '')
        band['name'] = band['name'].title()

format_bands(bands)

print bands
# => [{'name': 'Sunset Rubdown', 'active': False, 'country': 'Canad
#     {'name': 'Women', 'active': False, 'country': 'Canada' },
#     {'name': 'A Silver Mt Zion', 'active': True, 'country': 'Cana
```

---

```
pipeline_each(bands, [call(lambda x: 'Canada', 'country'),
                      call(lambda x: x.replace('.', ''), 'name'),
                      call(str.title, 'name')])
```



Exercises:

- Evaluating e^x
- Password cracker