Bank Kata

- Deposits and withdrawals can be made into an account
- Each deposit or withdrawal will have the amount of the operation and the date of the operation
- You should be able to transfer between accounts. A transfer will appear as a withdrawal in the account of the transfer or and as a deposit on the account of the transferee.
- A statement can be requested at any time. The statement will contain for each entry the date, the amount of deposition or withdrawal (by using a + or a -), and the balance of the account after the entry.
- Headers should be shown on the statement.
- You should be able to filter the statement (only deposits, only withdrawals, date)

```
Date         Amount  Balance
24.12.2015    +500      500
23.08.2016    -100      400
24.09.2017    -200      200
```

https://www.codurance.com/publications/2017/11/16/katas-for-functional-calisthenics
https://github.com/sandromancuso/Bank-kata

Name everything - All functions need to be named, you shouldn't use lambdas
No mutable state - You shouldn't use any variable of any type that can mutate
Exhaustive conditionals - No if without an else, switches and pattern matching always have all paths considered
Do not use intermediate variables - No variables declared in the body of a function, only parameters and other functions appear on the body
Expressions not statements - All lines are be expressions. All lines return a value
No Explicit recursion - Do not use explicit recursion
Generic building blocks - Use generic types for your functions, outside of the boundaries of your application
Side effects at the boundaries - Side effects appear only on the boundaries of your application
Infinite Sequences - Don't use collections that have a fixed size specified.
One argument functions - Each function has a single parameter.

Name Everything

All functions need to be named. Which means that you shouldn't use lambdas or anonymous functions. Also, name all your variables as per standard clean code rules.

Expected Outcome

Learn to recognize patterns on your code. It will help recognizing signatures that repeat, and the applying of DRY on your code. Also, it will clearly express the intention of the action, rather than just show the implementation.

```
array.filter(x => x.age > 10)
```

```
function personsOlderThan10(Person p)
{
      return x > 10;
}
array.filter(personsOlderThan10);
```

No mutable state

You shouldn't use any variable of any type that can mutate.

Expected Outcome

Learn how to create code around immutable variables.

Two main benefits:

- FP is about immutability. Most of its benefits comes from the fact that all your functions are (or should be) referencially transparent.
- Use of recursion as your main looping technique. No for or while loops in your code. For comprehensions abide by this rule.

```
var sum = 0
for(var person in persons){
   sum += person.age;
}
```

```
sum = persons.sum(x => x.age)
```

Exhaustive Conditionals

There can not be an if without an else. Switches and pattern matching should always have all paths considered (either through default paths or because all options have been considered).

Expected Outcome

Complete functions. A function should know how to deal with all possible values passed as arguments.

```
if(person.age > 10)
   console.log("older");
```

```
if(person.age > 10)
   console.log("older");
else
   console.log("younger");
```

Do Not Use Intermediate Variables

There shouldn't be any variable declared in the body of a function. Only parameters and other functions should appear on the body.

Expected Outcome

Use and understanding of functional pipelines. Starting on the path of functional composition. Functions are the building blocks of functional languages and the combination of functions create the logic of your application. Not having intermediate variables means changing the way that you think about your functions, and how the body of a function (composed of other functions) is, so the code remains legible and easy to follow.

```
function averageAge(Person[] persons){
  const totalAge = persons.sum(p ->
p.age);
  return totalAge / persons.length;
}
```

```
function averageAge(Person[] persons){
  return persons.sum(p -> p.age)/ persons.length;
}
```

Expressions, not Statements

All lines should be expressions. That's it, all lines should return a value.

Expected Outcome

Full purity on the code. Statements that are not expression are there for their side-effects. Nothing like that should be in the code.

```
function printAverageAge(){
  console.log(
    "Average age: ${averageAge(persons)}
  );
}
```

```
function printAverageAge(){
  return "Average age: ${averageAge(persons)}";
}

console.log(printAverageAge());
```

No Explicit Recursion

You shouldn't use explicit recursion like Clojure's java loop/recur forms or F# java let rec form.

Expected Outcome

Learning the power of map and reduce and use of High Order Functions. The idea is powerful enough to be the basis of some systems (Apache Hadoop) and has appeared on non-FP languages.

```
function fibonacci(n) {
  return n <= 1
      ? 1
      : fibonacci(n -1) + fibonacci(n - 2);
}
```

```
function fibonacci(n) {
  return Array
    .from(
      {length: n - 1},
      (_, i) => i + 2
    )
    .reduce((acc) =>
      ({ a: acc.b, b: acc.a + acc.b }),
      { a: 1, b: 1 }
    )
    .b;
```

Generic Building Blocks

Try to use a much as possible generic types for your functions, outside of the boundaries of your application.
Don't use a list, use a collection; Don't use an int when you can use a number; so forth and so on.

Expected Outcome

Easily composable code.
Using existing functionality and High Order Function provided by your language stack becomes much easier.

```
function calculateAverage(Person[] persons){
  var sum = 0
  for(var p in persons.values())
    sum += p.age;
  return sum / persons.length
}
```

```
interface Collection<T> {
  length(): number;
  values(): T;
}

function calculateAverage(Collection<Person> persons){
  var sum = 0
  for(var p in persons.values())
    sum += p.age;
  return sum / persons.length
}
```

Side Effects at the Boundaries

Side effects should only appear on the boundaries of your application. The guts of your application should have no side effects.

Expected Outcome

Limit the amount of code that is influenced by side effects. The main outcome is to create code on which you have tight control over side effects, when are the executed and to limit the effect on them on your logic. All your code logic should depend exclusively on parameters provided.

```
function printAverageAge(){
  console.log(
    "Average age: ${averageAge(persons)}
  );
}
```

```
function printAverageAge(){
  return "Average age: ${averageAge(persons)}";
}

function main(String[] args) {
   console.log(printAverageAge());
}
```

Infinite Sequences

All sequences, collections used need to be infinite collections. You can't use collections that need to have a fixed size specified.

Expected Outcome

Lazy Evaluation of Sequences Lazyness has some considerations regarding perfomance and not performing difficult calculations when they are not needed. Furthermore, reading from a file, database or network can be consideded an infinite sequence.

```
long nonRecursiveIterator(int n){
  return Enumerable
    .Range(2, n - 1)
    .Aggregate(
      new { a = 1, b = 1},
      (acc, i) => new { a = acc.b, b = acc.a +
acc.b},
      acc => acc.b
    );
}
```

```
IEnumerable<long> InfiniteSequence(int start) {
  while(true)
    yield return start++;
}

long nonRecursiveInfinite(int n) {
  return InfiniteSequence(2)
    .Take(n - 1)
    .Aggregate(
       new { a = 1, b = 1},
       (acc, i) => new { a = acc.b, b = acc.a + acc.b},
       acc => acc.b
     );
}
```

One argument functions

Each function should have a single parameter. You need to be explicit, just using the fact that the language works by automatically applying currying is not enough (Haskel, F#, ...) The parameter can be a structure/map or some other complex type (don't think this restricts to primitives).

Expected Outcome

Use of functional composition. Currying and partial function usage.

```
const person = new Person("John Doe", 15);

function concat(String separator, Person[] persons){
   return persons
      .reduce(
        "",
        (acc, p) => acc + separator + p
      );
}

concat("\n", persons);
```

```
var person = Person
      .ofName("John Doe")
      .withAge(15)
      .build();

function concat(String separator){
  persons => persons
      .reduce("", (acc, p) => acc + separator + p;
}

const newLineConcat = concat("\n");
newLineConcat(persons);
```