*2 слайд*

**Функциональное программирование** – парадигма программирования, в которой процесс [вычисления](https://ru.wikipedia.org/wiki/%D0%92%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5) трактуется как вычисление значений [функций](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F_(%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) в математическом понимании последних (в отличие от [функций](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) как подпрограмм в [процедурном программировании](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D0%B4%D1%83%D1%80%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)).

*3 слайд*

**Чистые функции** - Чистыми называют функции, которые не имеют [побочных эффектов](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B1%D0%BE%D1%87%D0%BD%D1%8B%D0%B9_%D1%8D%D1%84%D1%84%D0%B5%D0%BA%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) ввода-вывода и памяти (они зависят только от своих параметров и возвращают только свой результат)

**Immutable** – Значение переменных не может изменяться в процессе выполнения функции, т.е. как только мы объявили переменную, она больше никогда не изменит своего значения (аналогично константам)

**Рекурсия** - В функциональных языках [цикл](https://ru.wikipedia.org/wiki/%D0%A6%D0%B8%D0%BA%D0%BB_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) обычно реализуется в виде рекурсии. Строго говоря, в функциональной парадигме программирования нет такого понятия, как цикл. Рекурсивные функции вызывают сами себя, позволяя операции выполняться снова и снова

**Функции высших порядков** - Функции высших порядков — это такие функции, которые могут принимать в качестве аргументов и возвращать другие функции.

*4 слайд*

Here’s an example in Javascript of a Pure Function:

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

Notice that the ***add*** function does NOT touch the ***z*** variable. It doesn’t read from ***z*** and it doesn’t write to ***z***. It only reads ***x*** and ***y***, its inputs, and returns the result of adding them together.

That’s a pure function. If the ***add*** function did access ***z***, it would no longer be pure.

Here’s another function to consider:

function justTen() {  
 return 10;  
}

If the function, ***justTen***, is pure, then it can *only* return a constant. Why?

Because we haven’t given it any inputs. And since, to be pure, it cannot access anything other than its own inputs, the only thing it can return is a constant.

Since pure functions that take no parameters do no work, they aren’t very useful. It would be better if ***justTen*** was defined as a constant.

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

Consider this function:

function addNoReturn(x, y) {  
 var z = x + y  
}

Notice how this function doesn’t return anything. It adds ***x*** and ***y*** and puts it into a variable ***z*** but doesn’t return it.

It’s a pure function since it only deals with its inputs. It does add, but since it doesn’t return the results, it’s useless.

Let’s consider the first ***add*** function again:

function add(x, y) {  
 return x + y;  
}  
console.log(add(1, 2)); *// prints 3*  
console.log(add(1, 2)); *// still prints 3*  
console.log(add(1, 2)); *// WILL ALWAYS print 3*

Notice that ***add(1, 2)*** is always ***3***. Not a huge surprise but only because the function is pure. If the ***add*** function used some outside value, then you could ***never*** predict its behavior.

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

Since Pure Functions cannot change any external variables, all of the following functions are ***impure***:

writeFile(fileName);  
updateDatabaseTable(sqlCmd);  
sendAjaxRequest(ajaxRequest);  
openSocket(ipAddress);

All of these function have what are called ***Side Effects***. When you call them, they change files and database tables, send data to a server or call the OS to get a socket. They do more than just operate on their inputs and return outputs. Therefore, you can *never* predict what these functions will return.

Pure functions have **no** side effects.

*5 слайд*

let z = 10

z <- -5

Потом x = x+1

Как обойти эту проблему? 1) Вводить новые переменные, думать об очистке мусора и памяти  
 2) Использовать mutable

*6 слайд*

One obvious benefit is that if you have access to a value in your program, you only have read access, which means that no one else can change that value. Even you. So no accidental mutations.

Also, if your program is multi-threaded, then no other thread can pull the rug out from under you. That value is constant and if another thread wants to change it, it’ll have create a new value from the old one.

*7 слайд*

let makeResult(value : string) =

value

let validateValue(value : string) =

if value.Length = 11 then makeResult("Yes")

else makeResult("No")

let main argv =

let validationResult = validateValue("123-45-5678"): string

Console.WriteLine(validationResult);