

# Functional\* UI JS

Dan Menssen | MN.js February 2015

(\* Philosophically, at least)

# Functional\* UI JS

Dan Menssen | MN.js February 2015

The Olson logo consists of a solid red rectangle with the word "OLSON" in white, uppercase, sans-serif font centered within it.

**OLSON**

Solutions Architect

# Functional\* UI JS

Dan Menssen | MN.js February 2015

The Olson logo consists of a solid red rectangle with the word "OLSON" in white, uppercase, sans-serif font centered within it.

**OLSON**

Solutions Architect (?)



@menssen



[dan@menssen.org](mailto:dan@menssen.org)

# Functional programming

From Wikipedia, the free encyclopedia

*For subroutine-oriented programming, see [Procedural 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](#). In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function  $f$  twice with the same value for an argument  $x$  will produce the same result  $f(x)$  each time. Eliminating [side effects](#), i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

Functional programming has its roots in [lambda calculus](#), a [formal system](#) developed in the 1930s to investigate [computability](#), the [Entscheidungsproblem](#), function definition, function application, and [recursion](#). Many functional [programming languages](#) can be viewed as elaborations on the lambda calculus. In the other well-known declarative [programming paradigm](#), [logic programming](#), [relations](#) are at the base of respective languages.<sup>[1]</sup>

# Functional programming

From Wikipedia, the free encyclopedia

*For subroutine-oriented programming, see [Procedural 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](#). In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function  $f$  twice with the same value for an argument  $x$  will produce the same result  $f(x)$  each time. Eliminating [side effects](#), i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

Functional programming has its roots in [lambda calculus](#), a [formal system](#) developed in the 1930s to investigate [computability](#), the [Entscheidungsproblem](#), function definition, function application, and [recursion](#).

Many functional [programming languages](#) can be viewed as elaborations on the lambda calculus. In the other well-known declarative [programming paradigm](#), [logic programming](#), [relations](#) are at the base of respective languages.<sup>[1]</sup>



# Functional programming

From Wikipedia, the free encyclopedia

*For subroutine-oriented programming, see [Procedural 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](#). In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function  $f$  twice with the same value for an argument  $x$  will produce the same result  $f(x)$  each time. Eliminating [side effects](#), i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

Functional programming has its roots in [lambda calculus](#), a [formal system](#) developed in the 1930s to investigate [computability](#), the [Entscheidungsproblem](#), function definition, function application, and [recursion](#). Many functional [programming languages](#) can be viewed as elaborations on the lambda calculus. In the other well-known declarative [programming paradigm](#), [logic programming](#), [relations](#) are at the base of respective languages.<sup>[1]</sup>

( paradigms  
procedural  
OO  
functional  
logic )

# Functional programming

From Wikipedia, the free encyclopedia

*For subroutine-oriented programming, see [Procedural 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](#). In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function  $f$  twice with the same value for an argument  $x$  will produce the same result  $f(x)$  each time. Eliminating [side effects](#), i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

Functional programming has its roots in [lambda calculus](#), a [formal system](#) developed in the 1930s to investigate [computability](#), the [Entscheidungsproblem](#), function definition, function application, and [recursion](#). Many functional [programming languages](#) can be viewed as elaborations on the lambda calculus. In the other well-known declarative [programming paradigm](#), [logic programming](#), [relations](#) are at the base of respective languages.<sup>[1]</sup>



# Functional programming

From Wikipedia, the free encyclopedia

*For subroutine-oriented programming, see [Procedural 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](#). In functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function  $f$  twice with the same value for an argument  $x$  will produce the same result  $f(x)$  each time. Eliminating [side effects](#), i.e. changes in state that do not depend on the function inputs, can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.

Functional programming has its roots in [lambda calculus](#), a [formal system](#) developed in the 1930s to investigate [computability](#), the [Entscheidungsproblem](#), function definition, function application, and [recursion](#). Many functional [programming languages](#) can be viewed as elaborations on the lambda calculus. In the other well-known declarative [programming paradigm](#), [logic programming](#), [relations](#) are at the base of respective languages.<sup>[1]</sup>

( procedural )

```
var x = getUserInput()
function increment() {
  ++x
}
```

( OO )

```
class Incrementor {
  private x = 1
  function Incrementor() {
    this.x = getUserInput()
  }
  function execute() {
    ++this.x
  }
}
```

( logic )

????

$\forall x \exists y (x = a \supset y = a + 1)$

( functional )

```
function increment(x) {
  return x + 1
}
```

```
x = increment(
  getUserInput()
)
```

//  $f(x) = 3x^2 + 2x + 5$

**++X**

**this.x**

**( procedural )**

```
var x = getUserInput()
```

```
x = x+2
```

```
x = '$' + (string) x + '.00'
```

```
print x
```

**( functional )**

```
print(
```

```
    concat(
```

```
        '$', add(2, getUserInput()), '.00'
```

```
    )
```

```
)
```

**( procedural )**

```
for (var i = 0; i < max; ++i) {  
    doSomething(x)  
}
```

**( functional )**

```
doSomethingForEveryElementInList(  
    {1..max},  
    doSomething  
)
```



( why?  
testing - no mocks!  
easier to reason about - stay tuned )

```
doSomethingForEveryElementInList(  
  {1..max},  
  doSomething  
)
```

```
doSomethingForEveryElementInList(  
  {1..max},  
  doSomething  
)
```

```
[1, 2, 3].forEach(doSomething)
```

```
print(concat('$', add(2, getUserInput()), '.00'))
```

```
print(concat('$', add(2, getUserInput()), '.00'))
```

```
transformEveryElementInList(  
  transformEveryElementInList(  
    {1..max},  
    doSomething  
  ),  
  doSomethingElse  
)
```



```
print(concat('$', add(2, getUserInput()), '.00'))
```

```
transformEveryElementInList(  
  transformEveryElementInList(  
    {1..max},  
    doSomething  
  ),  
  doSomethingElse  
)
```

```
[1, 2, 3].map(doSomething).map(doSomethingElse)
```

# Three functions to transform an array into something else

• `Array.prototype.map()` creates a new array with the results of calling a provided function on each element of the array.

• `Array.prototype.filter()` creates a new array with all elements that pass the test implemented by the provided function.

• `Array.prototype.reduce()` executes a reducer function on each element of the array, resulting in a single output value.

• `Array.prototype.sort()` sorts the elements of an array in place and returns the sorted array.

• `Array.prototype.concat()` joins two or more arrays and returns the new array.

• `Array.prototype.slice()` returns a new array consisting of the elements from the specified start to the end.

• `Array.prototype.splice()` changes the contents of an array by removing or replacing existing elements and/or adding new elements to the array.

# Three functions to transform an array into something else

**// transform into subset of itself**

```
[1,2,3,4,5].filter(function(item) {  
  return item > 3  
})
```

**// [4,5]**

# Three functions to transform an array into something else

**// transform into a new list**

```
[1,2,3,4,5].map(function(item) {  
  return item + 1  
})
```

**// [2,3,4,5,6]**

# Three functions to transform an array into something else

**// collapse**

```
[1,2,3,4,5].reduce(function(last, item) {  
  return item + lastResult  
}, 0)
```

**// 15**

```
[ [1, 2], [3, 4], [5, 6]].reduce(function(last, item) {  
  return last.concat(item)  
}, [])
```

**// [1,2,3,4,5]**



<http://github.com/menssen/jsmn-february-2015>