Real World Functional Programming

Mikael Lundin

Email
mikael.lundin@valtech.se

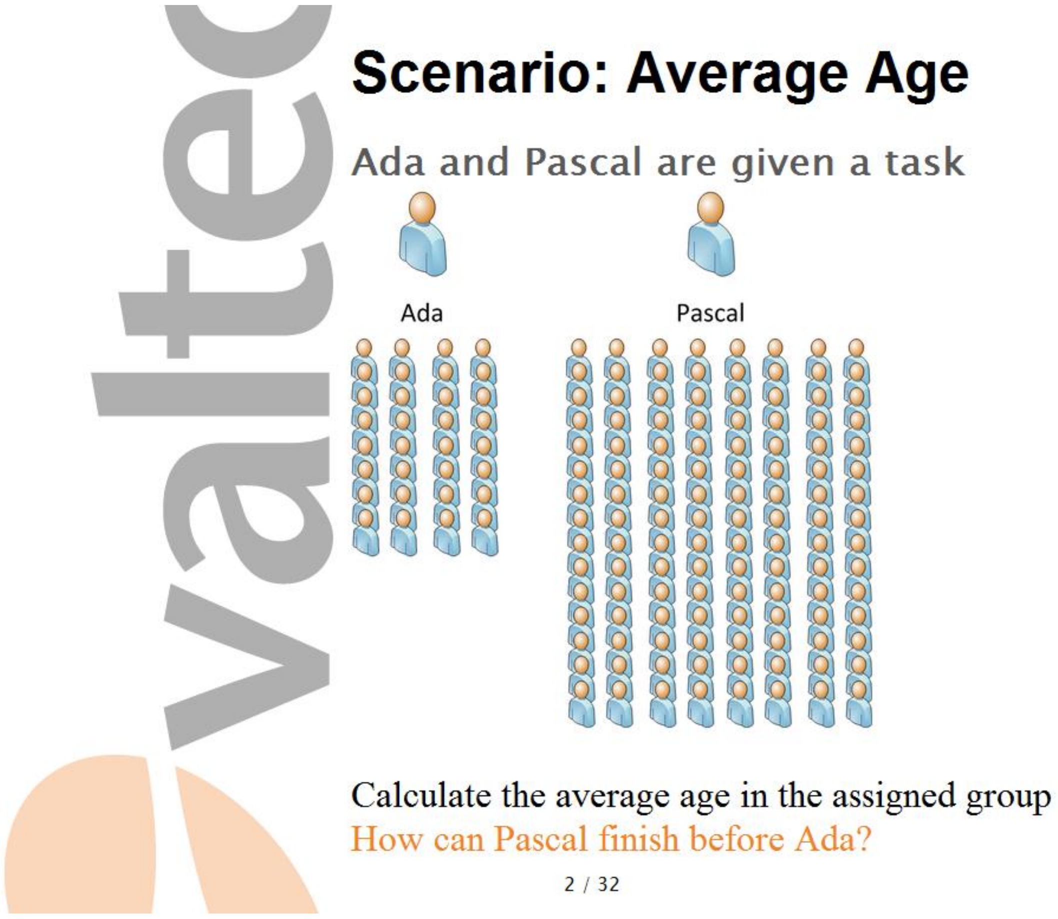
Blog
litemedia.info

Twitter
@mikaellundin

Most of my examples will be in C# but it should be comprehensible by any Java programmer also.

Why Functional Programming?

- Code with less side effects
- Code that is more expressive
- Code that is easy to parallelize



Scenario: Average Age

Ada and Pascal are given a task

Calculate the average age in the assigned group

How can Pascal finish before Ada?

2 / 32

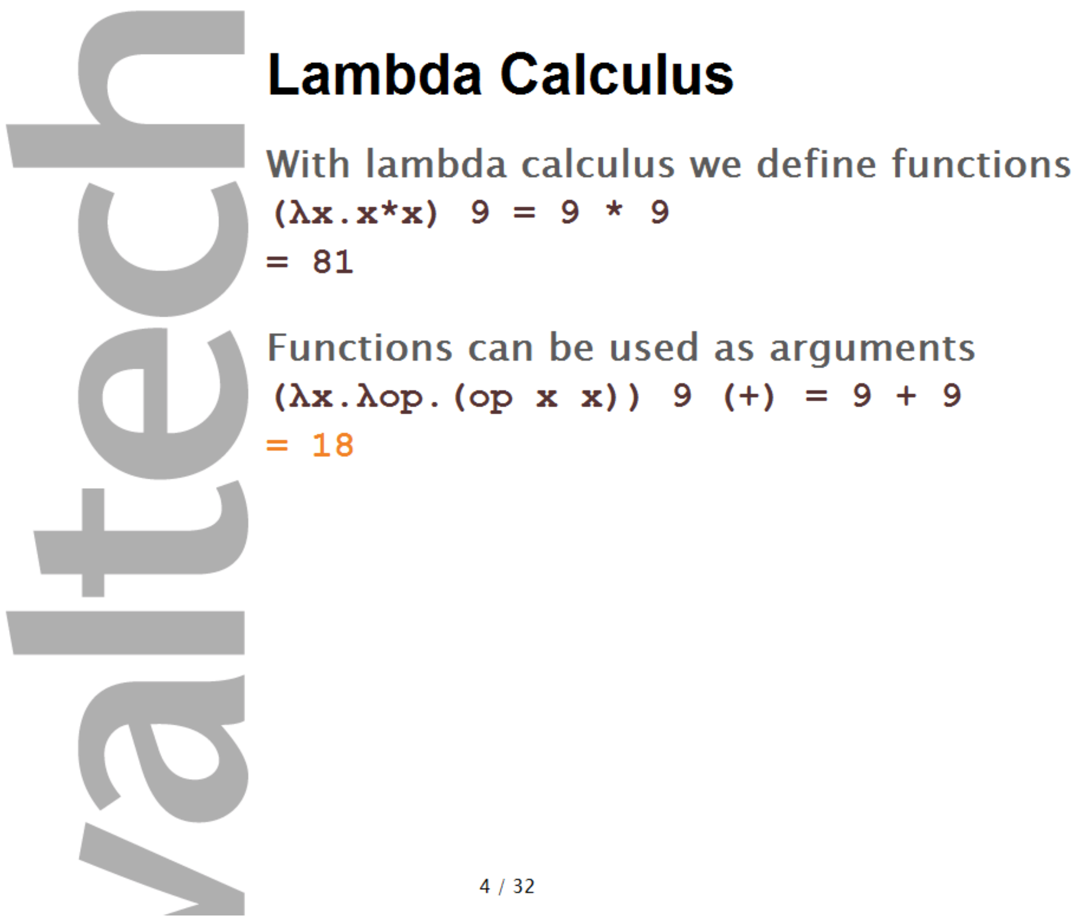
Calculate the average age in the assigned group.While you think about that I will talk about some of the history of functional programming.

History

Alonzo Church

λ

It all began in the 1930's with Alonzo Church when he created the lambda calculus. He wanted to discribe the world in functions.



Lambda Calculus

With lambda calculus we define functions

```
(λx.x*x) 9 = 9 * 9
= 81
```

Functions can be used as arguments

```
(λx.λop.(op x x)) 9 (+) = 9 + 9
= 18
```

4 / 32

This is a function that squares a number.

We can call this with the number nine

If you type out it, it will become 9 * 9

The result is of course 81

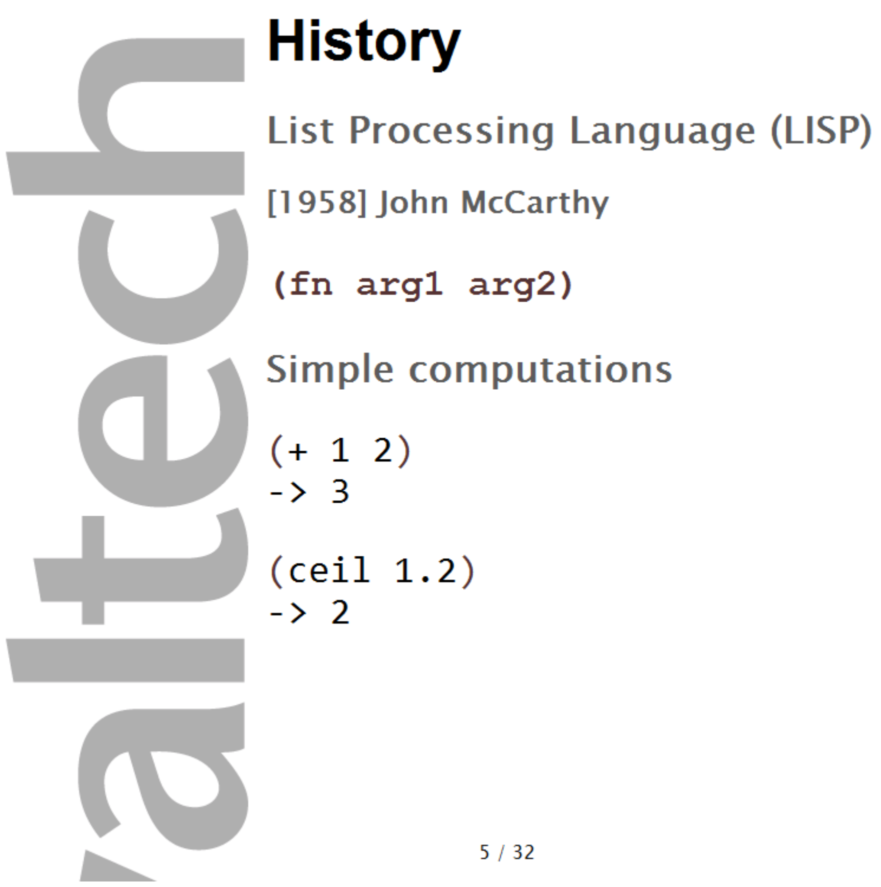
The point here is the function and that we can reuse it with other arguments.

Here's a function that takes two arguments, x and op. op is a function that operates on x and takes two arguments.

We can call this with the number 9 and operation (+)

This of course prints out as 9 + 9

And the result is 18



History

List Processing Language (LISP)

[1958] John McCarthy

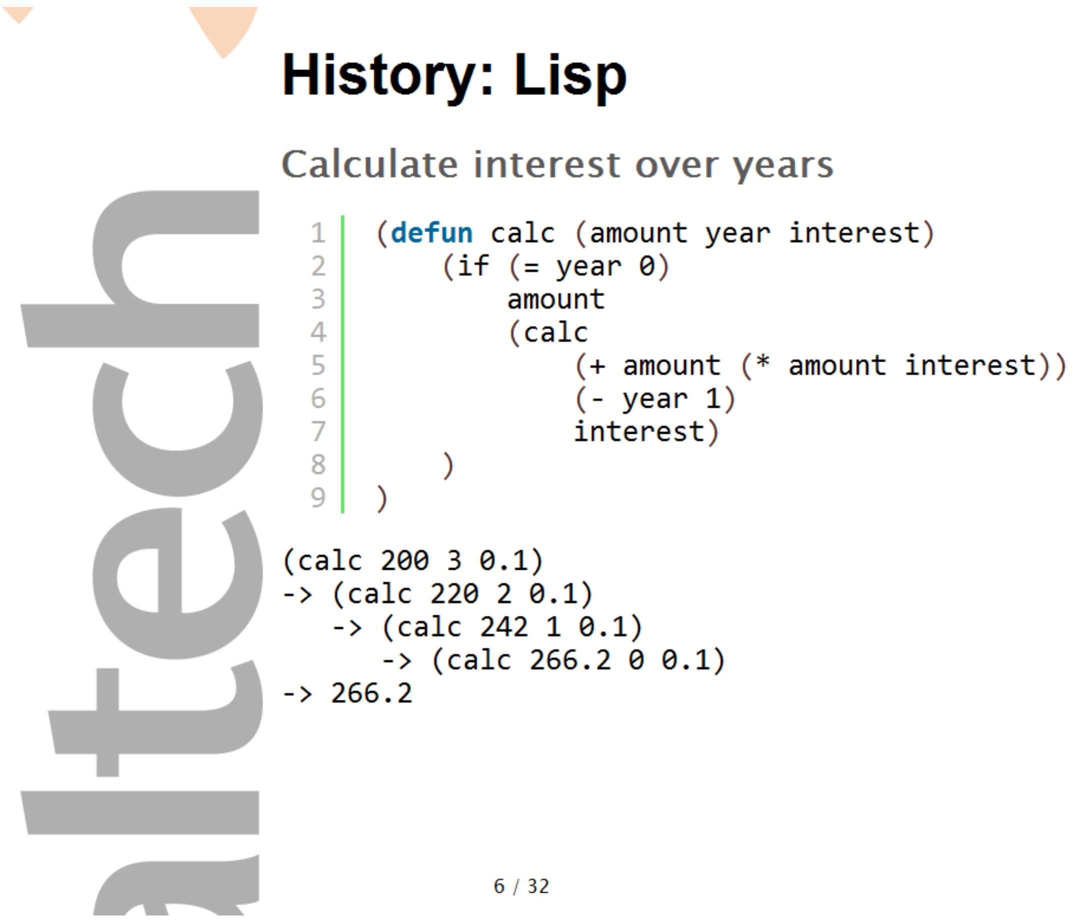
```
(fn arg1 arg2)
```

Simple computations

```
(+ 1 2)
-> 3

(ceil 1.2)
-> 2
```

LISP was created 1958 as the first functional language, at the same time as COBOL and Fortran two imperative programming languages.

LISP pioneered computer science paradigms that we still use today, tree data structures, dynamic typing and self hosting compiler.



History: Lisp

Calculate interest over years

```lisp
(defun calc (amount year interest)
    (if (= year 0)
        amount
        (calc
            (+ amount (* amount interest))
            (- year 1)
            interest)
    )
)
```

```
(calc 200 3 0.1)
-> (calc 220 2 0.1)
   -> (calc 242 1 0.1)
      -> (calc 266.2 0 0.1)
-> 266.2
```

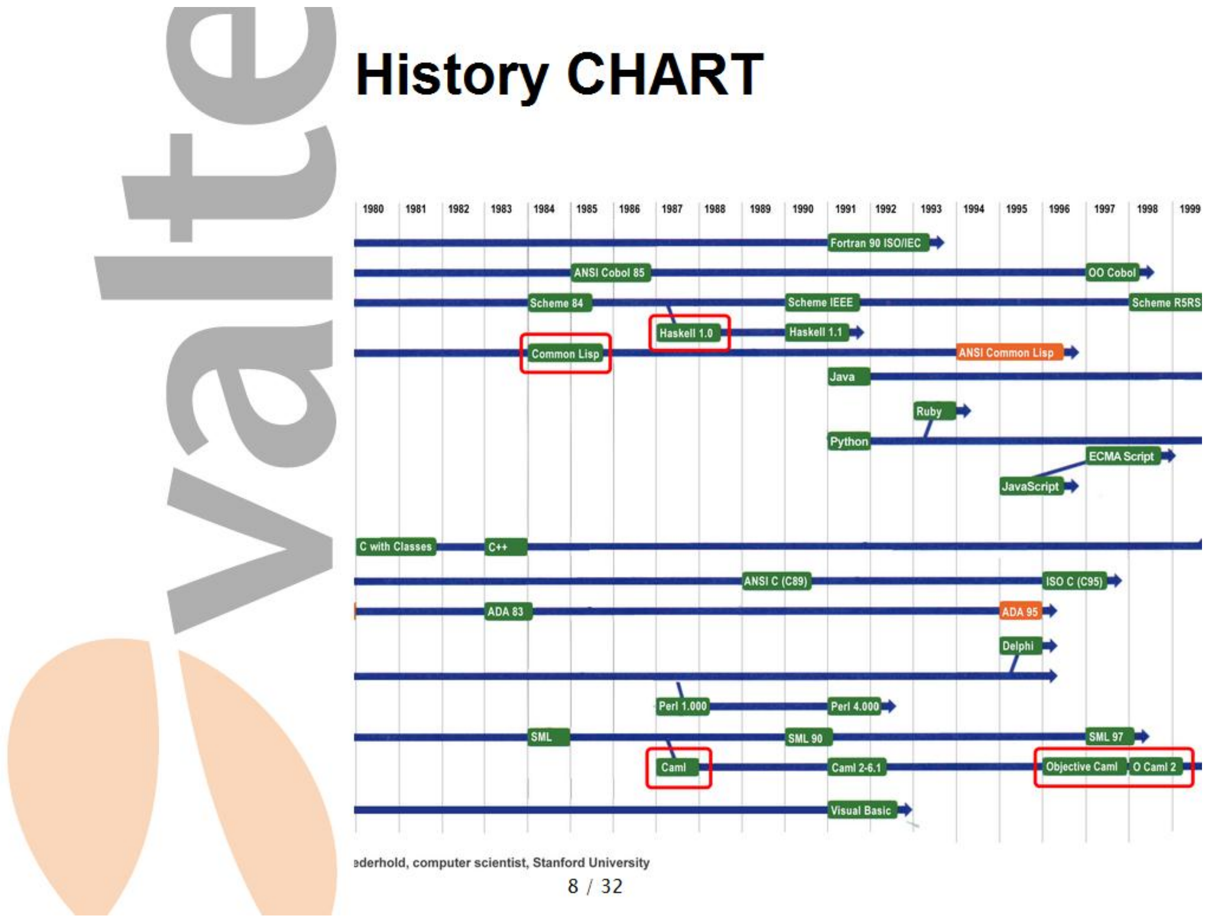
6 / 32

Here's some code that will calculate interest for an amount of money over a couple of years in lisp. It reads like this

```
if year = 0 then
           return amount
else
           calc
                      amount + (amount * interest / 100)
                      year--
                      interest
```

Recursion

Recursion: ***see recursion.***

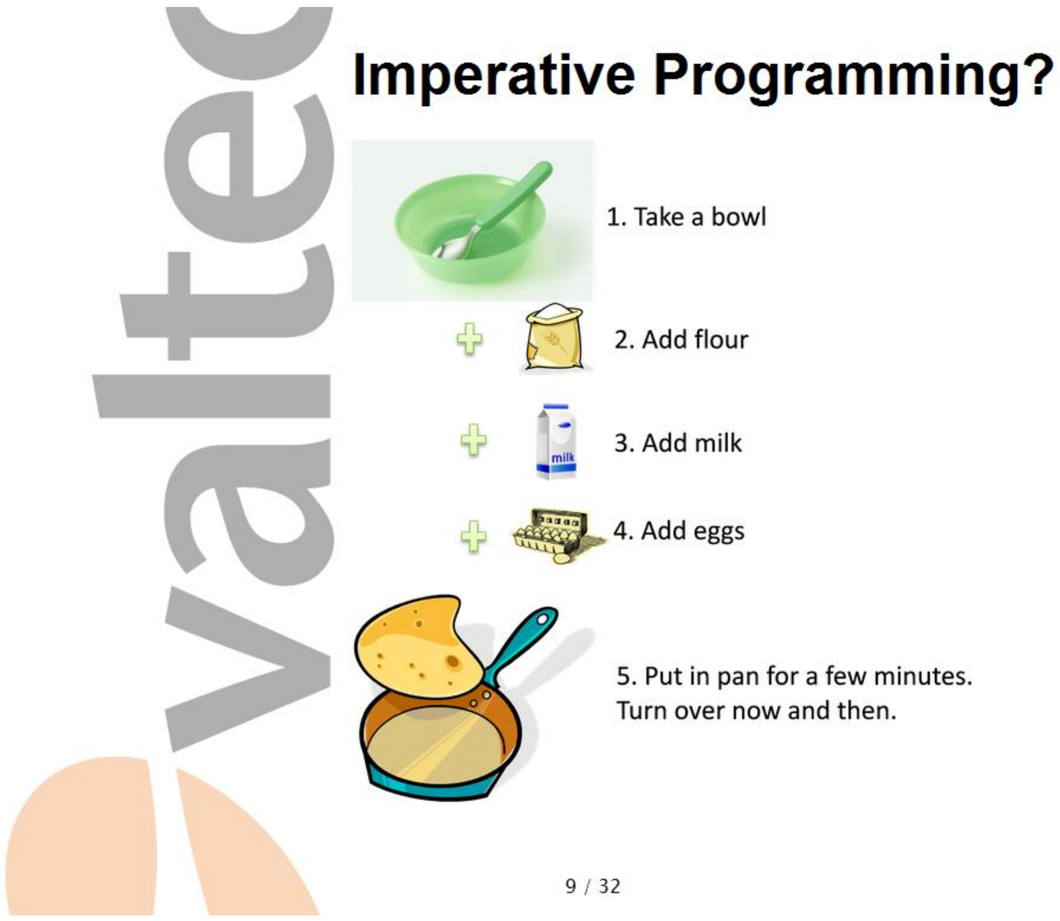


Lisp came 1958 right between the large imperative languages Fortran and Cobol. Most of the languages we use today derive from those two languages.

Scheme is a more formalized version of Lisp and it came about 1975 and became quite popular, especially in the academic world.

It's worth to notice Common Lisp, the standardization of Lisp in 1984 and how Haskell came out of Scheme in 1987. Haskell is a pure functional programming language, and that means that it does not allow side effects.

Caml that came out of ML and OCaml that came out of Caml is worth to notice. Objective Caml is what inspired F# that is the main functional programming language on the .NET platform today.



Imperative Programming?

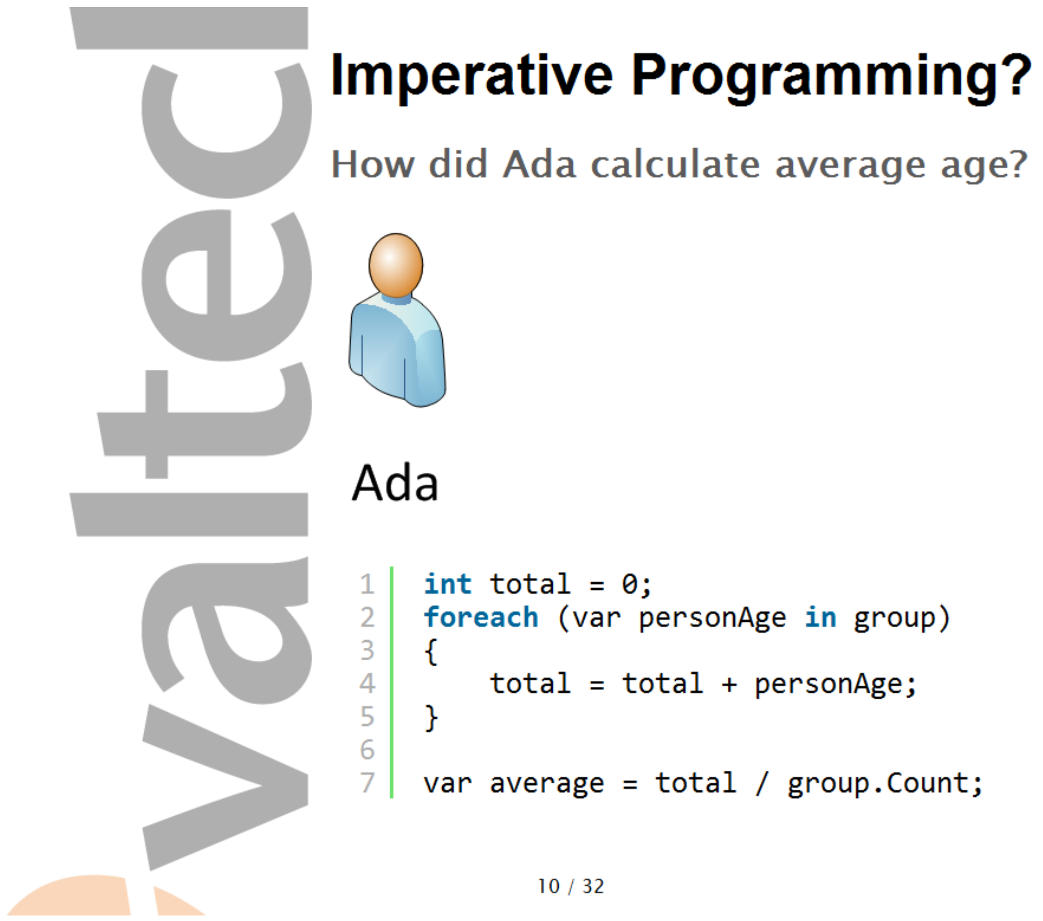
1. Take a bowl
2. Add flour
3. Add milk
4. Add eggs
5. Put in pan for a few minutes.
Turn over now and then.

9 / 32

Imperative programming includes state. You have a state and you modify it until you're satified with the result.

It's like making pancakes. You start with an empty bowl. You had flour, eggs, and milk. You take the contents and put it into a frying pan and after 2 minutes you have a pancake.

Imperative programming is having a state and mutating it into completeness.



Imperative Programming?

How did Ada calculate average age?

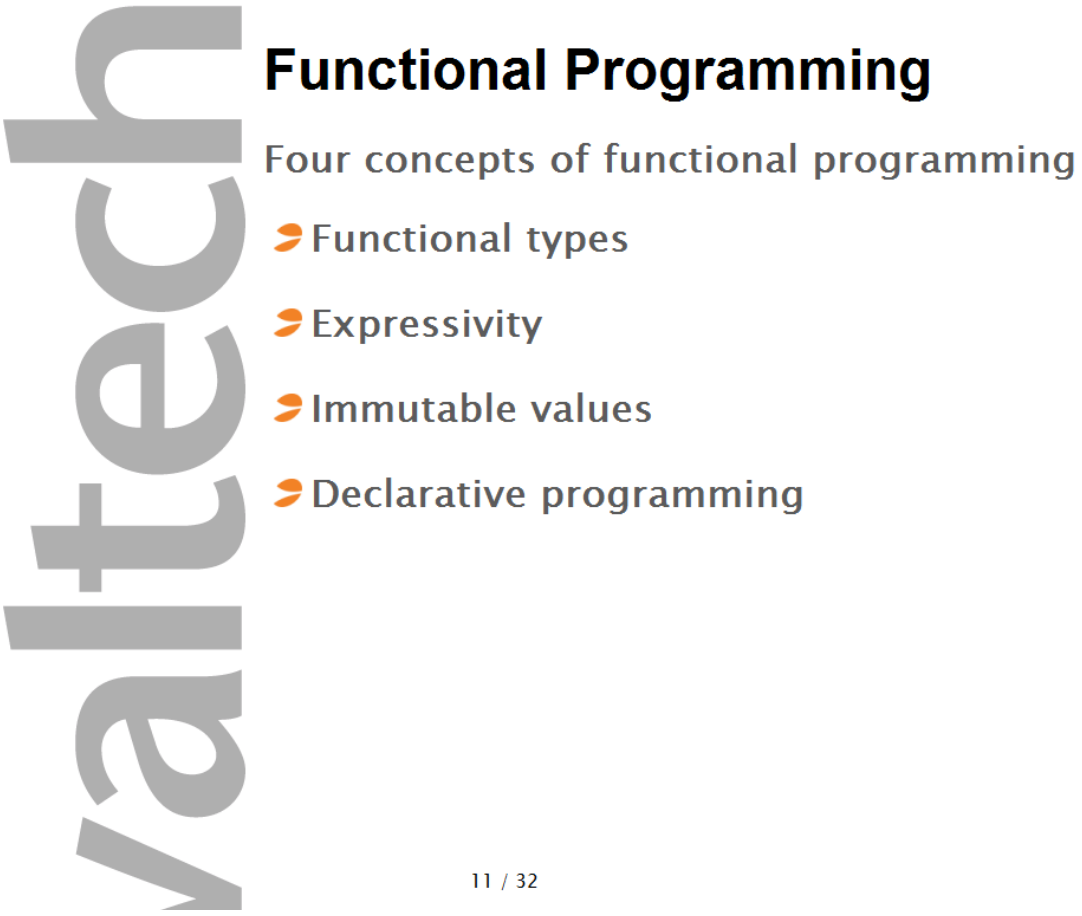
Ada

```
int total = 0;
foreach (var personAge in group)
{
    total = total + personAge;
}

var average = total / group.Count;
```

10 / 32

Remember Ada and Pascal? They're going to calculate the average age of everyone in their group. Ada goes for a purely imperative approach. She sums up everyones age in the group and then divide with the number of people in the group.

It is very easy for our brains to understand imperative thinking because we have been schooled that way.



Functional Programming

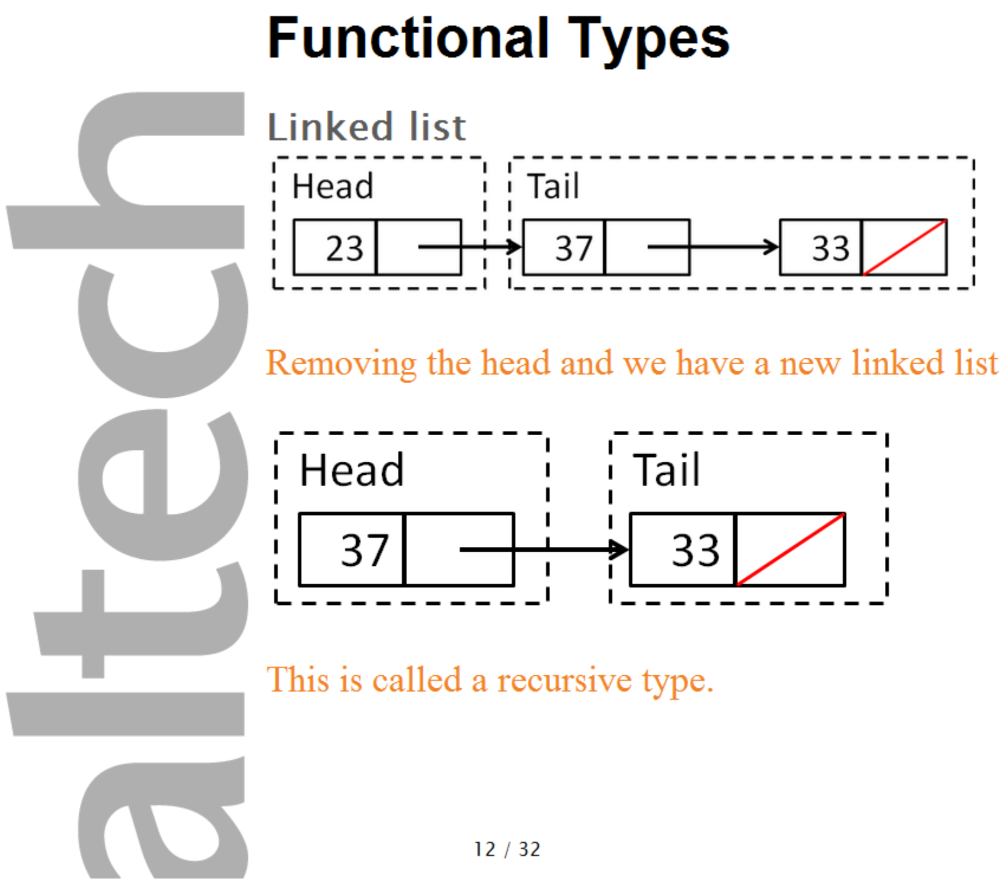
Four concepts of functional programming

- Functional types
- Expressivity
- Immutable values
- Declarative programming

11 / 32

Pascal used a more functional way to solve the problem and managed to do so before Ada even though he had a four times larger group of people.

The four concepts of functional programming are Functional types, expressivity, immutable values and declarative programming. I will look into what these concepts are, and how Pascal can use them to complete the task.



Functional Types

Linked list

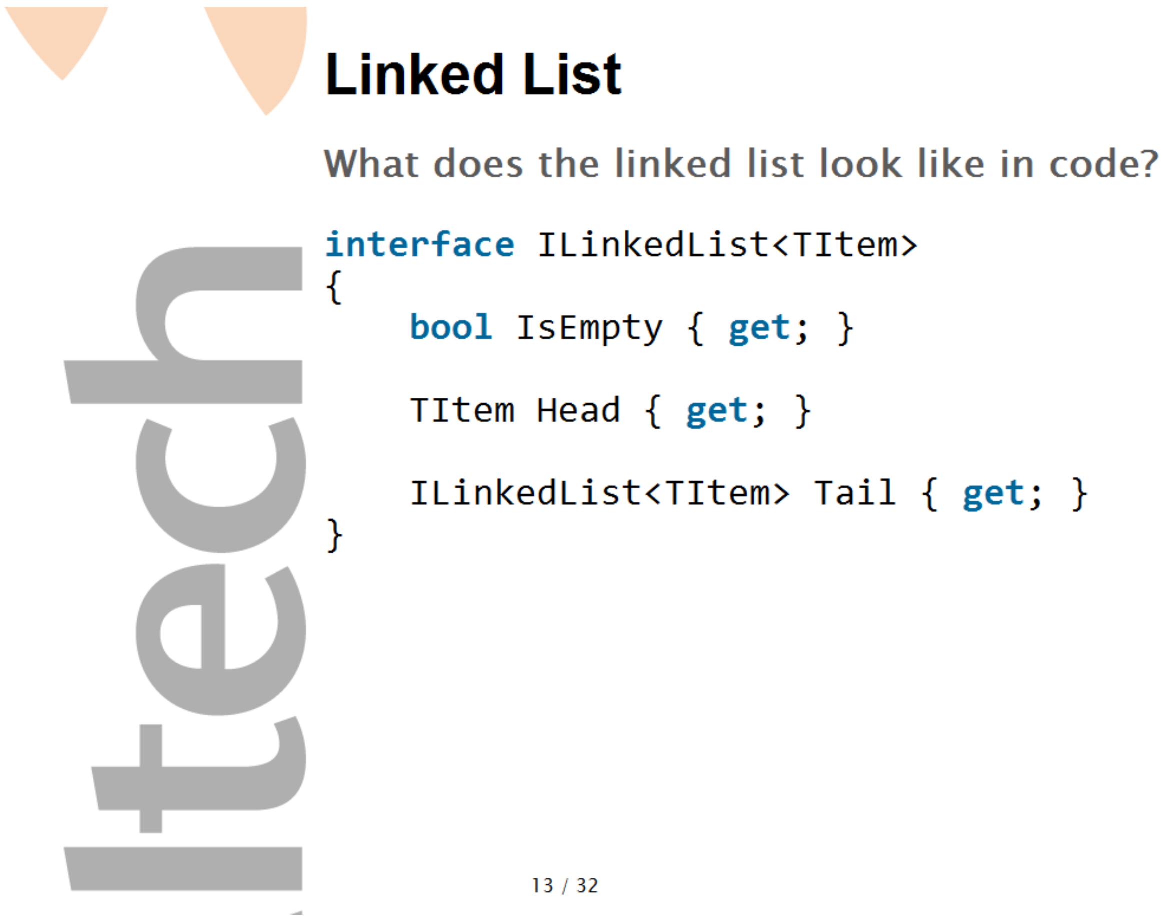
Removing the head and we have a new linked list

This is called a recursive type.

12 / 32

A functional type is a type that works well with functional programming paradigms. The most common is the linked list.

The linked list has two parts, a head and a tail. The tail is in itself a linked list. This makes the type recursive.



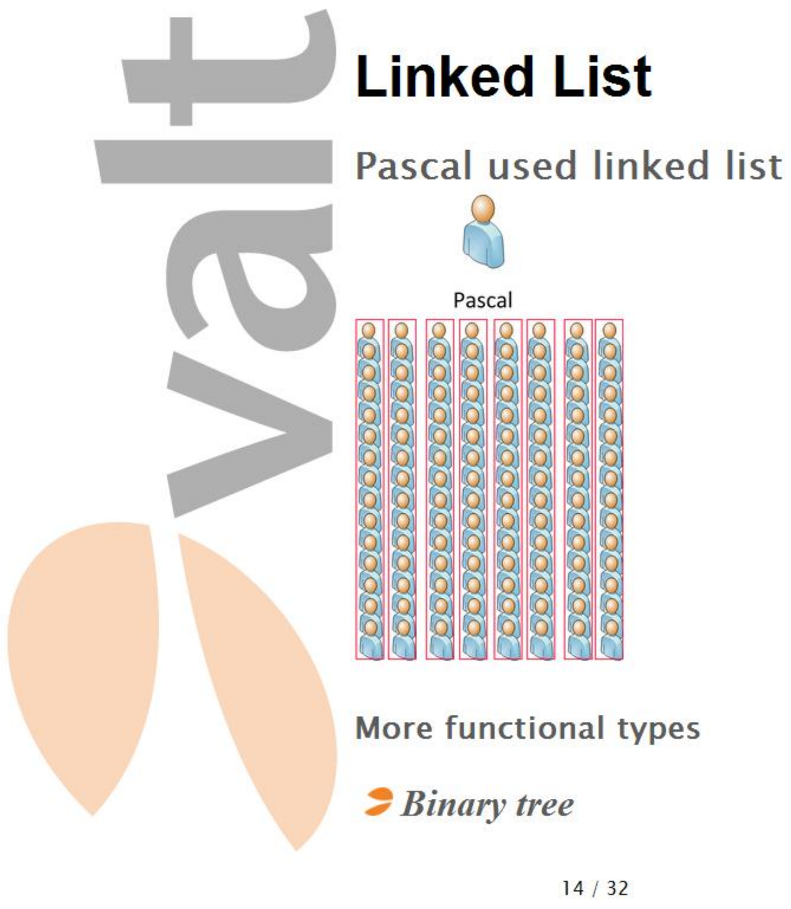
Linked List

What does the linked list look like in code?

```
interface ILinkedList<TItem>
{
    bool IsEmpty { get; }

    TItem Head { get; }

    ILinkedList<TItem> Tail { get; }
}
```

13 / 32

How would a linked list definition look in code? Something like this where the type has a head, and a tail. You have a property to tell if the list is empty or not. It's all you need for this recursive type.



Linked List

Pascal used linked list

More functional types

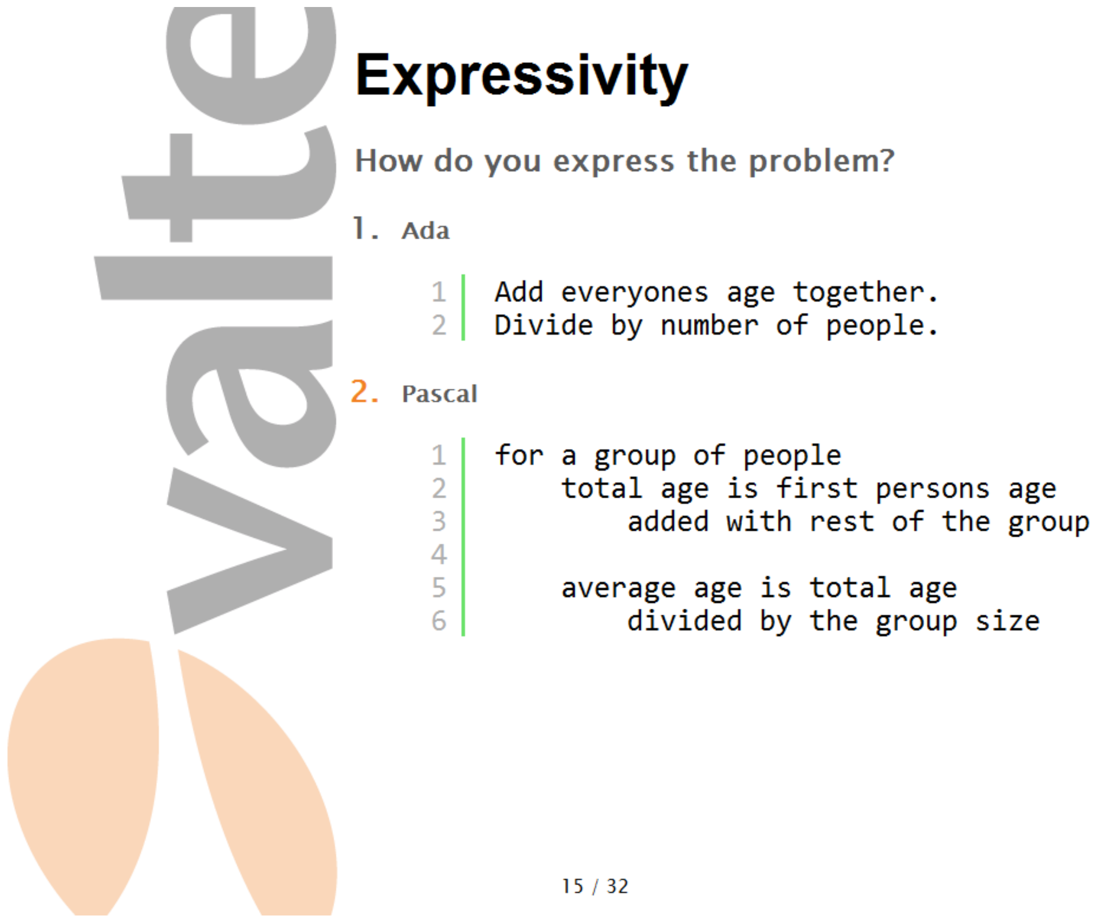
- *Binary tree*

14 / 32

Pascal used the linked list functional type to structure his group.

Another common functional type is the binary tree. You use that for search and sorting algorithms.

In a functional language each function has a type. This makes it easier to use functions as values.



Expressivity

How do you express the problem?

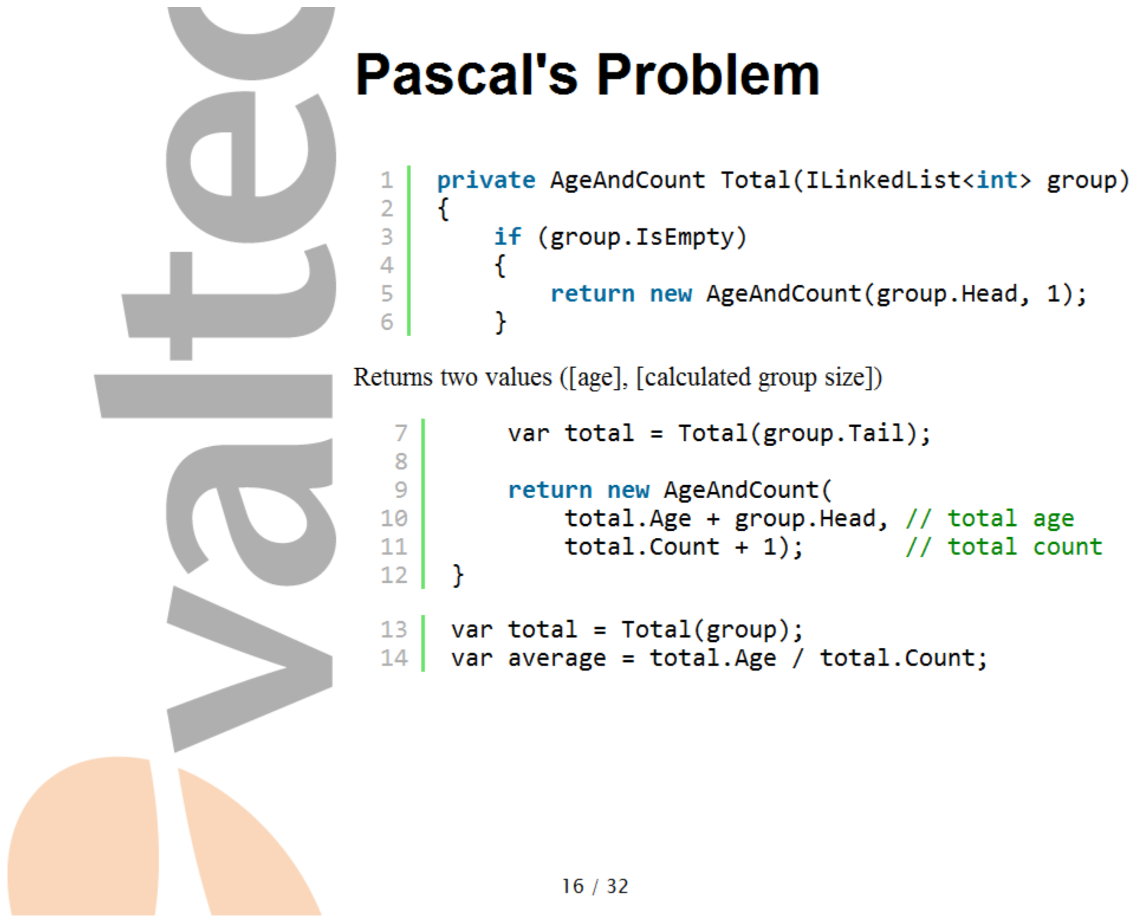
1. Ada

```
Add everyones age together.
Divide by number of people.
```

2. Pascal

```
for a group of people
    total age is first persons age
        added with rest of the group

    average age is total age
        divided by the group size
```

The key to expressivity is how you express the problem. Ada tackles the problem as the entire group where as Pascal tries to solve the problem for the indiviual person.



Pascal's Problem

```
private AgeAndCount Total(ILinkedList<int> group)
{
    if (group.IsEmpty)
    {
        return new AgeAndCount(group.Head, 1);
    }
```

Returns two values ([age], [calculated group size])

```
    var total = Total(group.Tail);

    return new AgeAndCount(
        total.Age + group.Head, // total age
        total.Count + 1);       // total count
}
```

```
var total = Total(group);
var average = total.Age / total.Count;
```

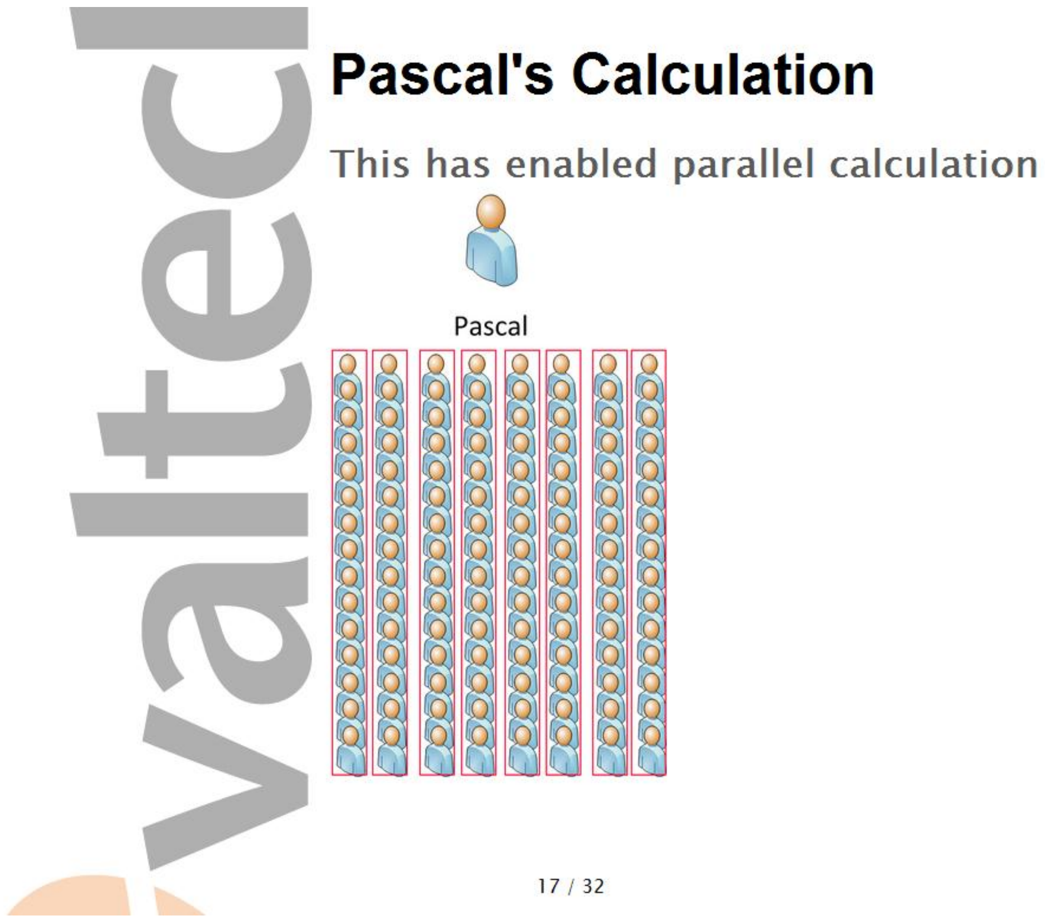
16 / 32

We define the following

What happens when there's only one person in the group? Return that persons age with group size 1

For a group we express that the total age is first persons age + rest of the groups calculated age

We calculate the number of persons in the group at the same time as we calculate the age. Since we need to return two values form the calculation we uses another functional construct called Tuple. It is simply two values as one.



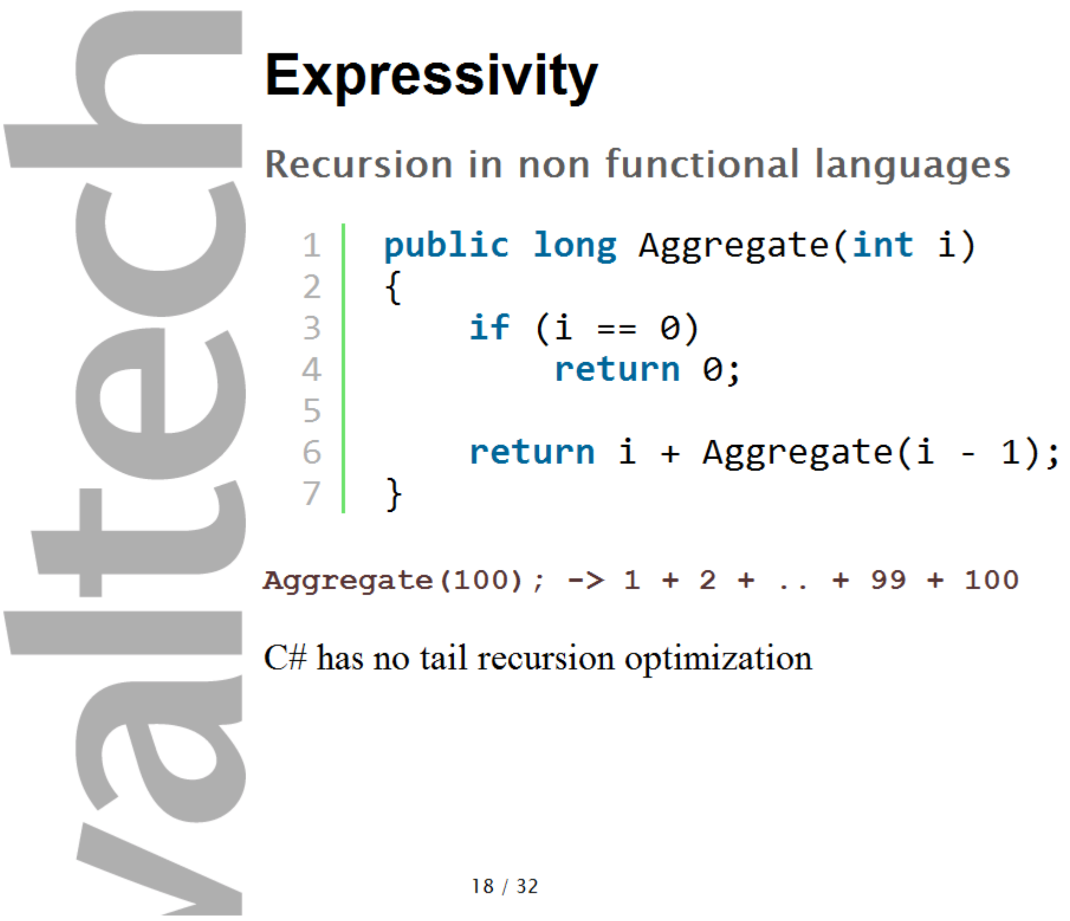
Pascal's Calculation

This has enabled parallel calculation

17 / 32

Instead of counting from start to finish Pascal divided his problem into 8 subproblems

He told the first person in each column to calculate the ages and return the result to him.



Expressivity

Recursion in non functional languages

```
public long Aggregate(int i)
{
    if (i == 0)
        return 0;

    return i + Aggregate(i - 1);
}
```

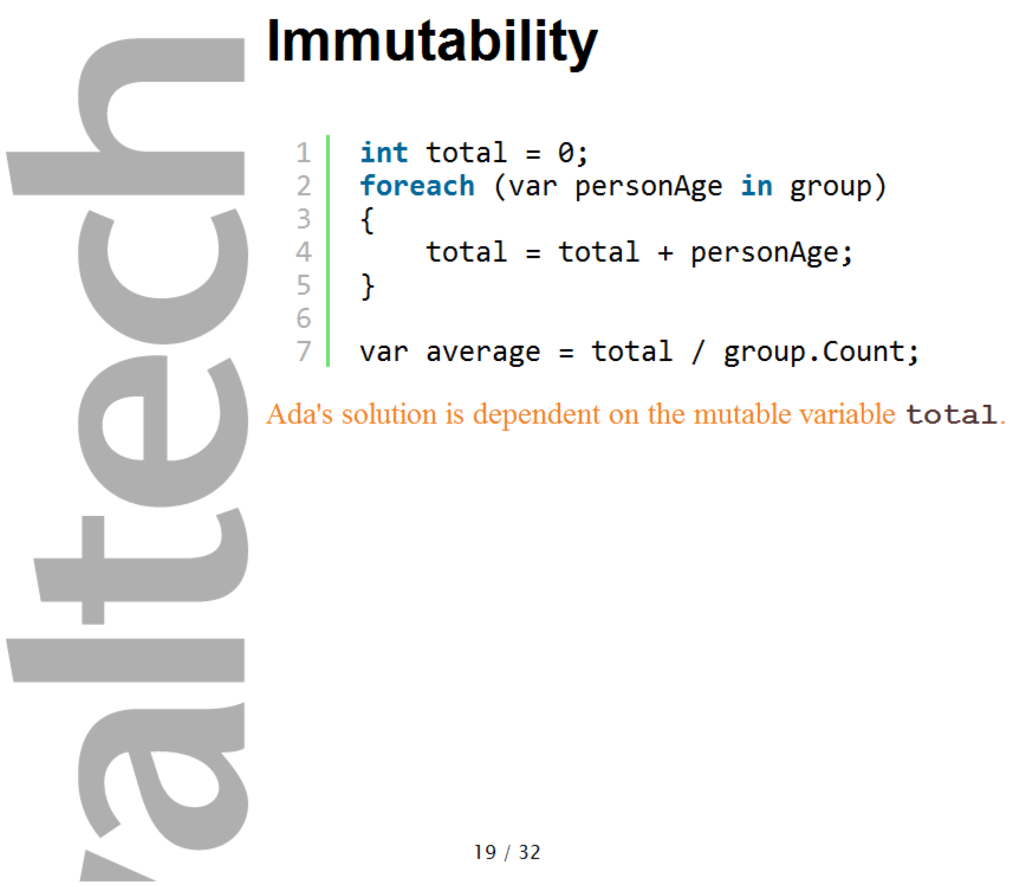
```
Aggregate(100); -> 1 + 2 + .. + 99 + 100
```

C# has no tail recursion optimization

C# has no tail recursion optimization

You should beware of recursing in non functional languages, because they do not optimize for recursion. When you write a recursion like this it will compile into function calls where as a functional language might optimize it into a while loop. This is called tail recursion optimization because extensive functional calls is expensive.

Make sure that you know that the depth has a limit before you start recursing in C#.



Immutability

```
int total = 0;
foreach (var personAge in group)
{
    total = total + personAge;
}

var average = total / group.Count;
```

Ada's solution is dependent on the mutable variable `total`.

19 / 32

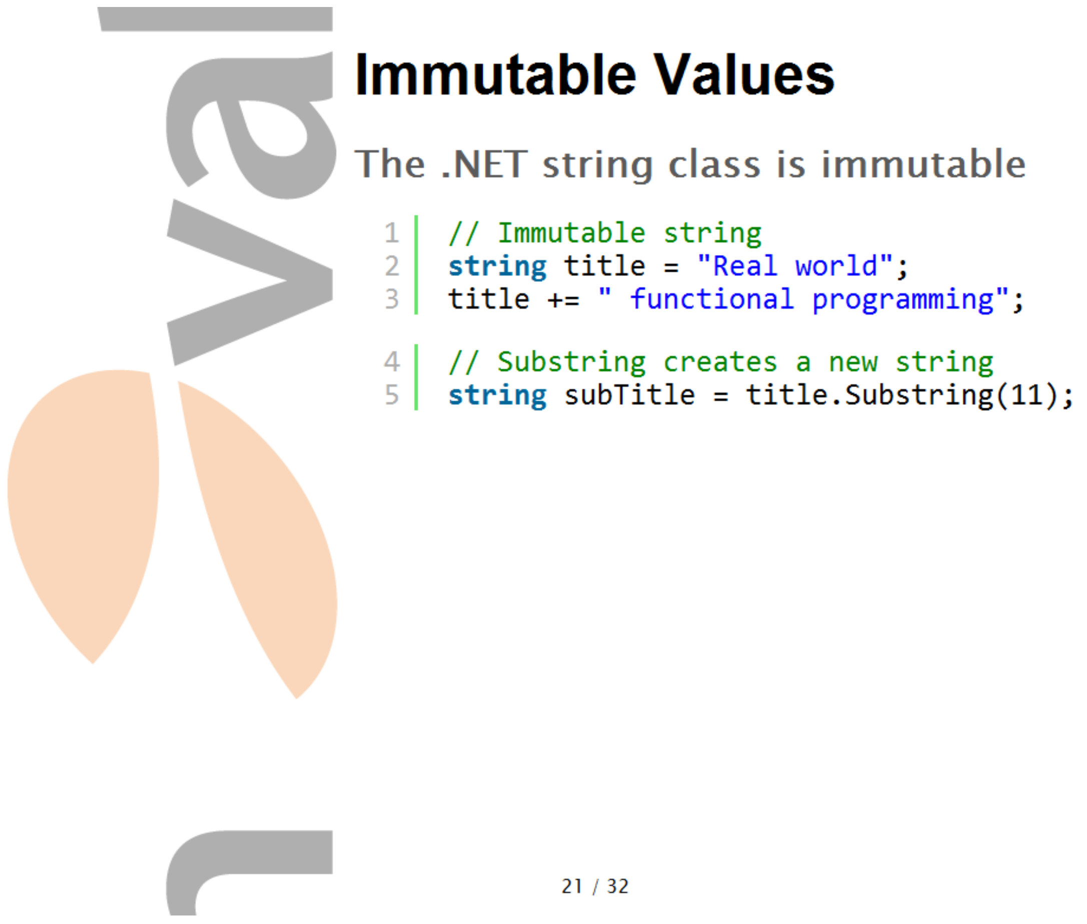
It's mutable values that makes parallelization so hard. If we we're to parallelize Ada's solution it would be necessary to share the mutable state "total" between different threads. This is the source of deadlocks and race conditions.

Immutable Values

An immutable value cannot change

"In FP you don't set variables, but bind values."

- Less side effects
- Enforces expressive problem solving



Immutable Values

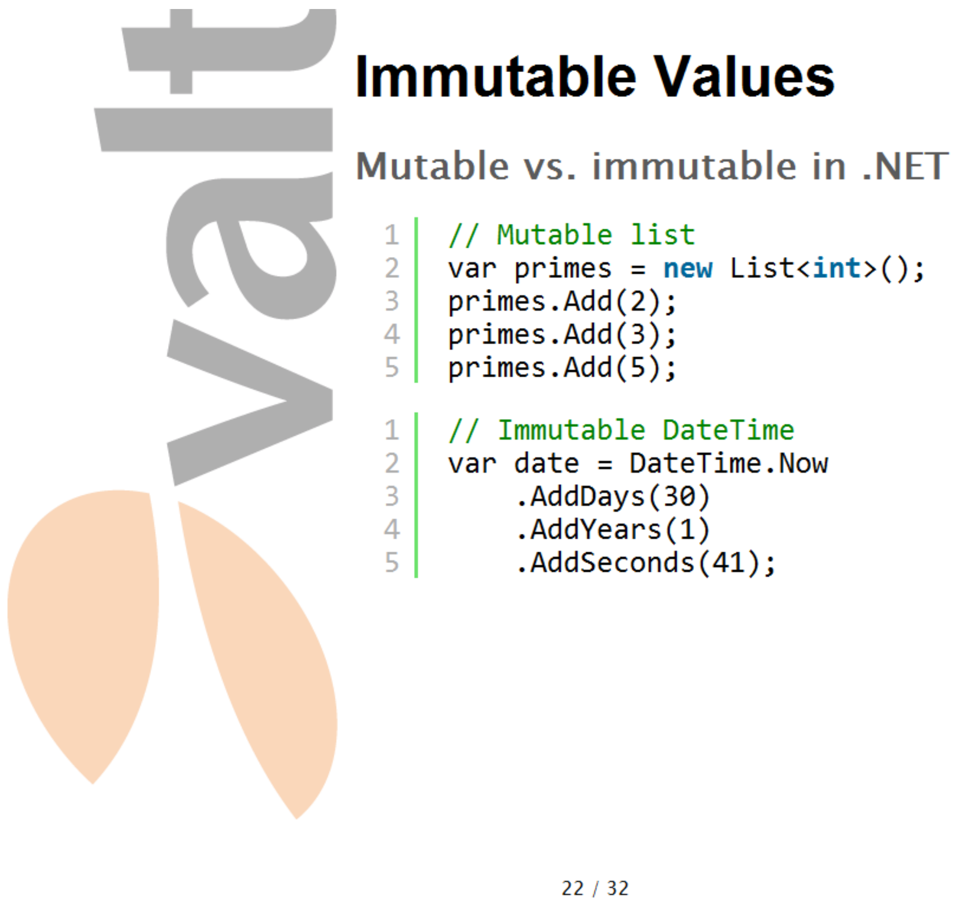
The .NET string class is immutable

```
// Immutable string
string title = "Real world";
title += " functional programming";

// Substring creates a new string
string subTitle = title.Substring(11);
```

21 / 32

We have immutable types in the .NET framework. String is immutable. If we add another string to a string, it will create a new string - not modify the existing.

String instance methods that manipulates the string always returns a new string instance.



Immutable Values

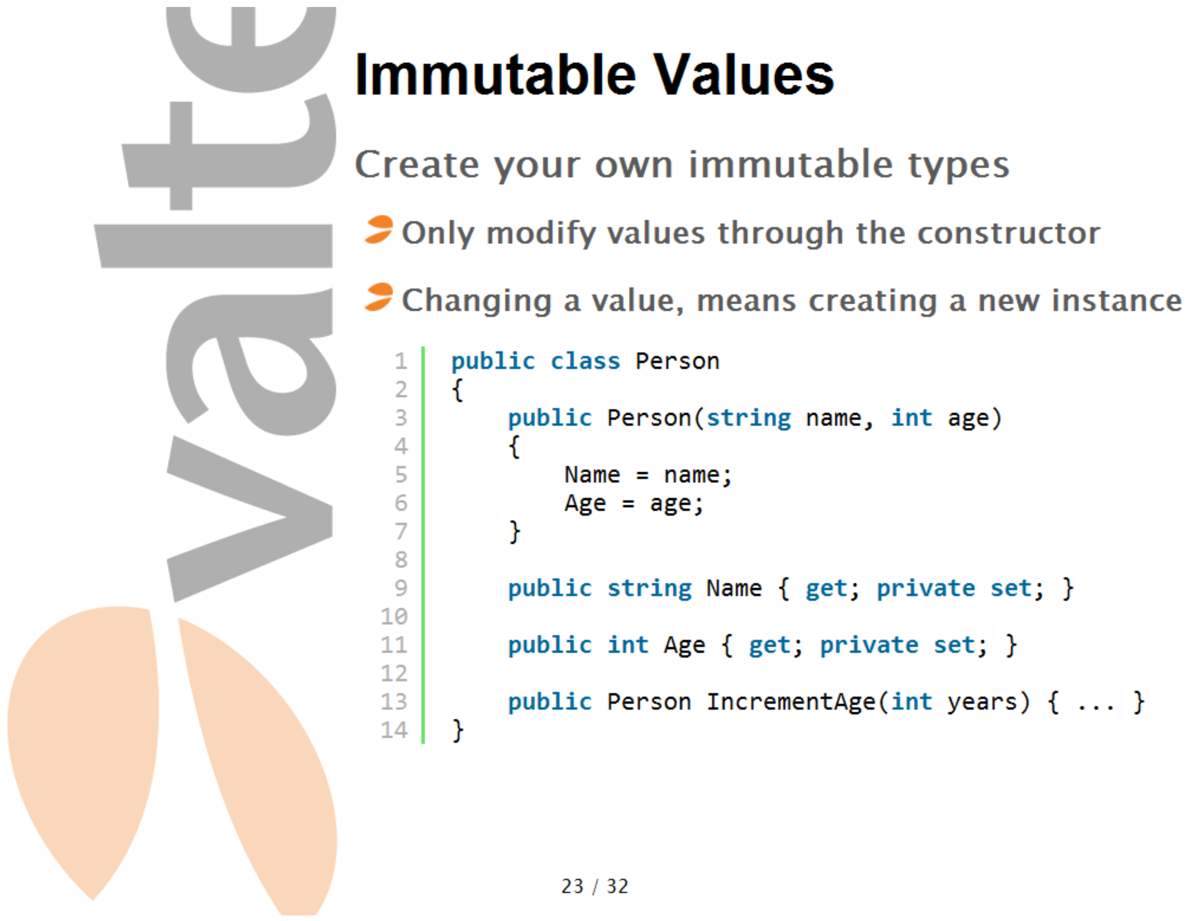
Mutable vs. immutable in .NET

```
// Mutable list
var primes = new List<int>();
primes.Add(2);
primes.Add(3);
primes.Add(5);
```

```
// Immutable DateTime
var date = DateTime.Now
    .AddDays(30)
    .AddYears(1)
    .AddSeconds(41);
```

22 / 32

Immutable leads to function chaining. No operation on datetime can ever return null



Immutable Values

Create your own immutable types

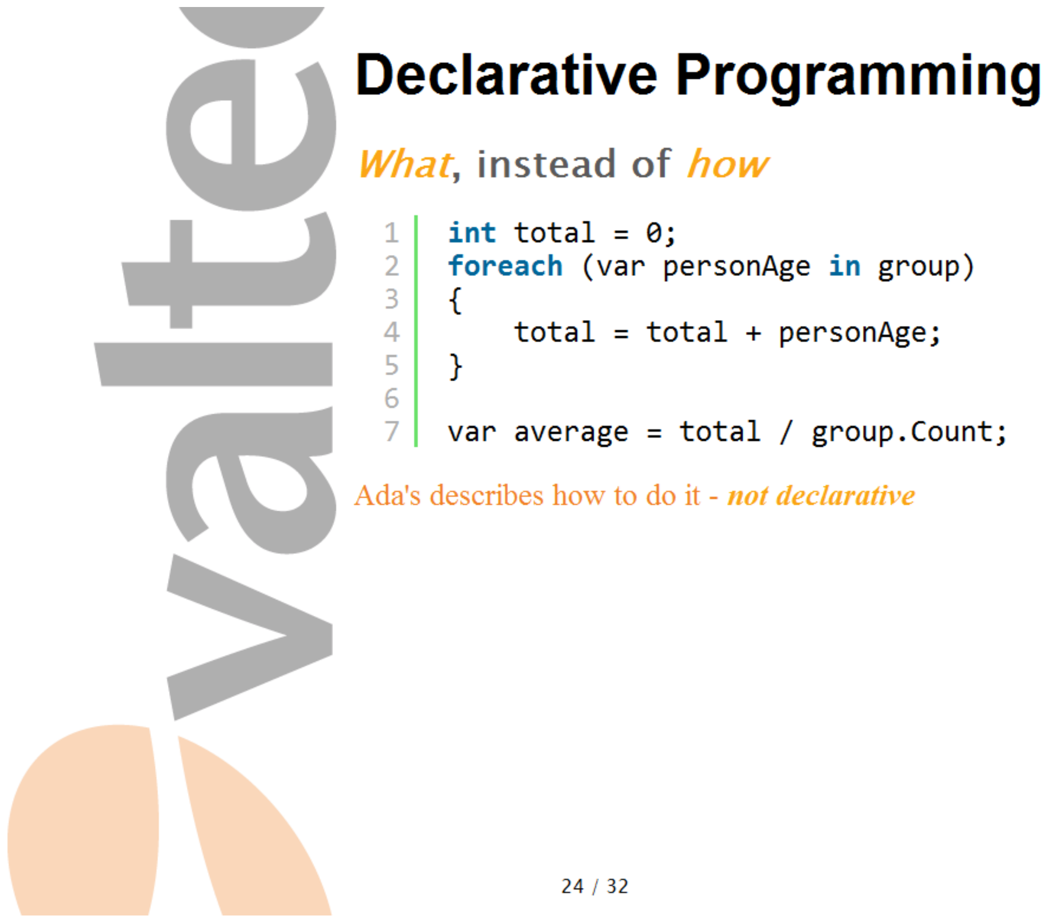
- Only modify values through the constructor
- Changing a value, means creating a new instance

```
public class Person
{
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }

    public string Name { get; private set; }

    public int Age { get; private set; }

    public Person IncrementAge(int years) { ... }
}
```

23 / 32

Creating immutable types will help you code more functionally so you should make your types immutable where you can. In an immutable type you can't modify values without getting a new instance of the type.

In above example you will get a new Person instance when you want to increase the age of the Person.



Declarative Programming

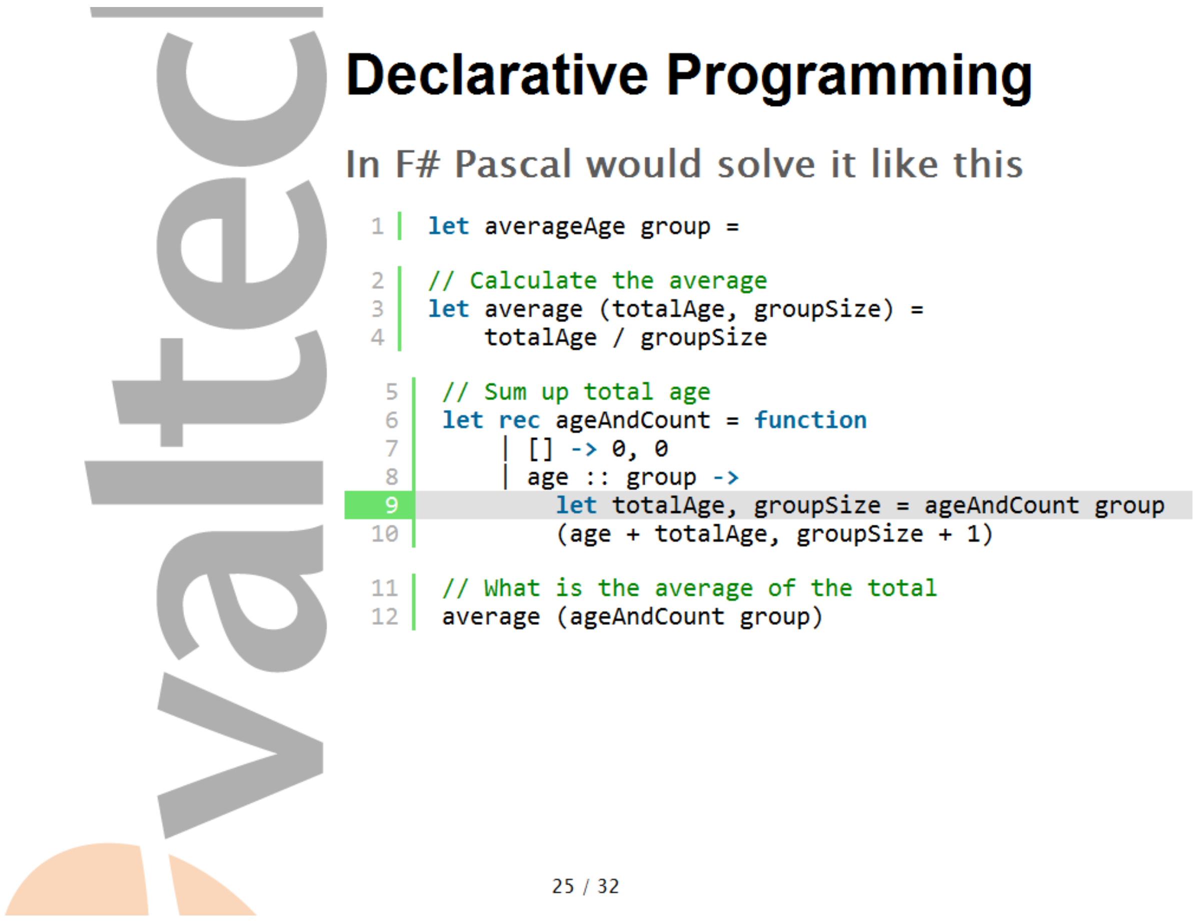
What, instead of *how*

```
int total = 0;
foreach (var personAge in group)
{
    total = total + personAge;
}

var average = total / group.Count;
```

Ada's describes how to do it - ***not declarative***

24 / 32

Ada's problem solution focus on how to calculate the result. A declarative problem solution will focus on what we want instead of how to get it.



Declarative Programming

In F# Pascal would solve it like this

```fsharp
let averageAge group =

// Calculate the average
let average (totalAge, groupSize) =
    totalAge / groupSize

 // Sum up total age
 let rec ageAndCount = function
     | [] -> 0, 0
     | age :: group ->
         let totalAge, groupSize = ageAndCount group
         (age + totalAge, groupSize + 1)

 // What is the average of the total
 average (ageAndCount group)
```

25 / 32

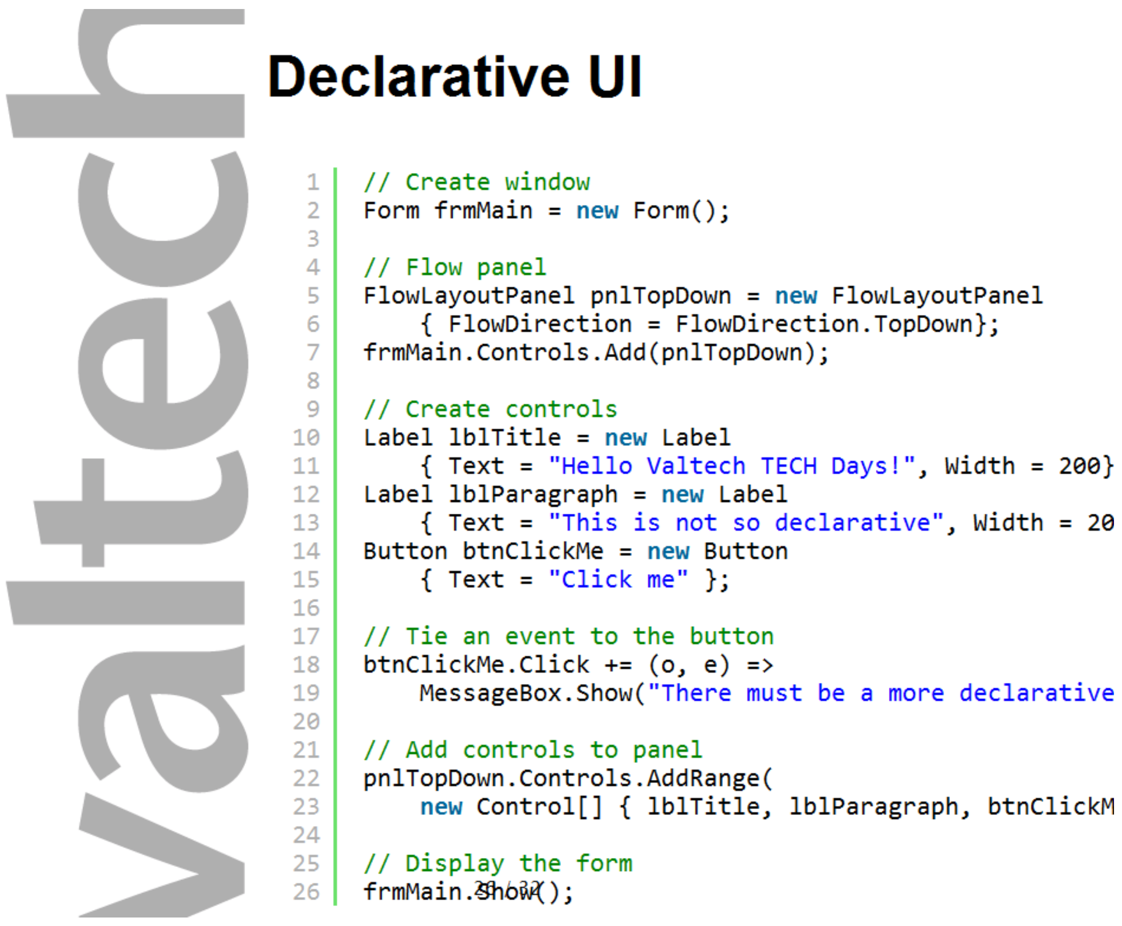
In declarative programming we declare what, instead of how.

averageAge of group is -> call average with result from totalAgeAndCount of group

average is -> totalAge divided by groupSize

totalAgeAndCount is -> 0, 0 when group is empty first persons age + rest of the groups age and rest of the group size + 1

Like this we have declared what we want instead of how to get it. This method is declarative and very expressive.



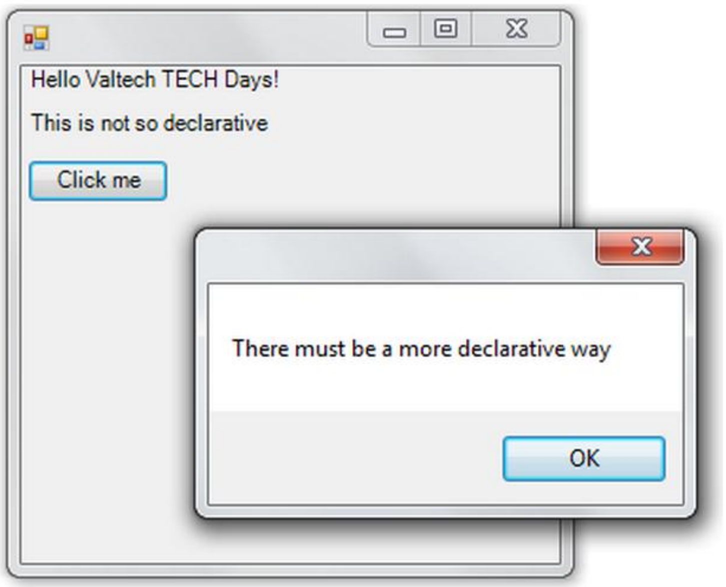
Declarative UI

```
// Create window
Form frmMain = new Form();

// Flow panel
FlowLayoutPanel pnlTopDown = new FlowLayoutPanel
    { FlowDirection = FlowDirection.TopDown};
frmMain.Controls.Add(pnlTopDown);

// Create controls
Label lblTitle = new Label
    { Text = "Hello Valtech TECH Days!", Width = 200}
Label lblParagraph = new Label
    { Text = "This is not so declarative", Width = 20
Button btnClickMe = new Button
    { Text = "Click me" };

// Tie an event to the button
btnClickMe.Click += (o, e) =>
    MessageBox.Show("There must be a more declarative

// Add controls to panel
pnlTopDown.Controls.AddRange(
    new Control[] { lblTitle, lblParagraph, btnClickM

// Display the form
frmMain.Show();
```

Windows forms programming is an exellent example of UI programming that is not declarative. All the way you tell the computer how it should produce the desired UI.

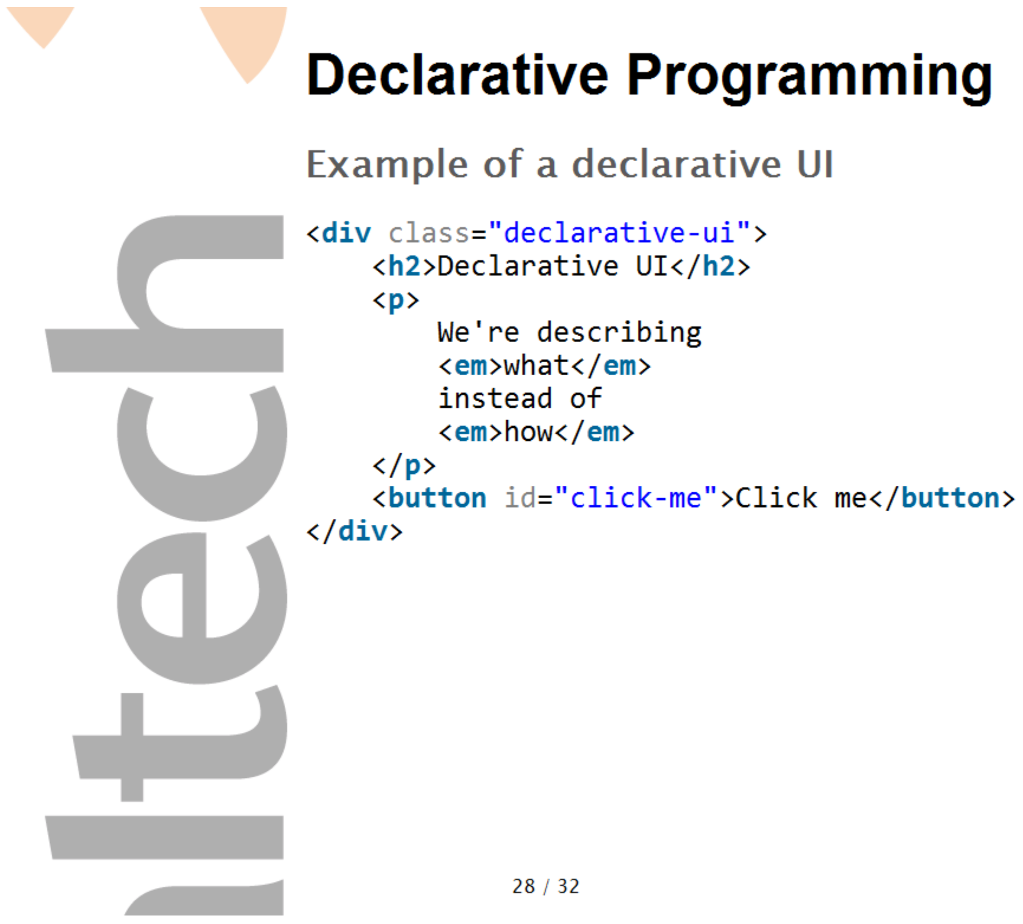
Declarative UI

WinForms is not declarative


Hello Valtech TECH Days!

This is not so declarative

There must be a more declarative way



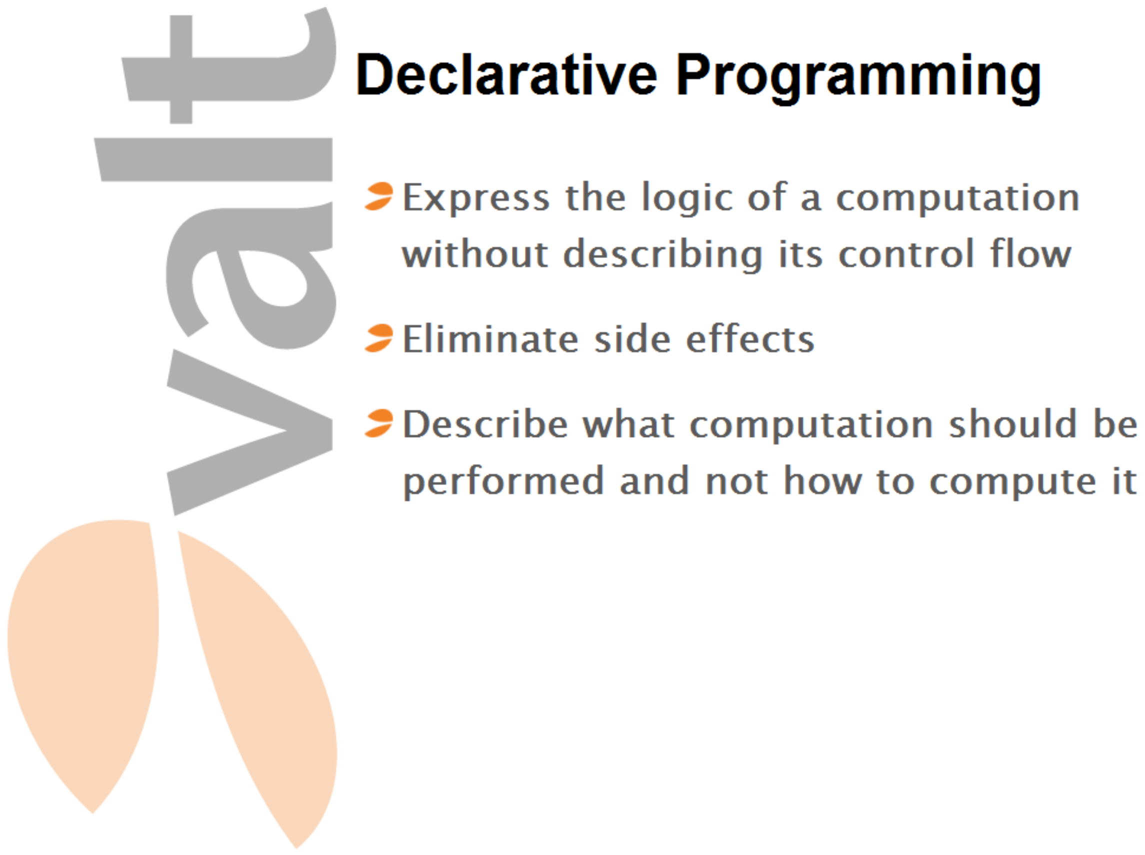
Declarative Programming

Example of a declarative UI

```
<div class="declarative-ui">
    <h2>Declarative UI</h2>
    <p>
        We're describing
        <em>what</em>
        instead of
        <em>how</em>
    </p>
    <button id="click-me">Click me</button>
</div>
```

28 / 32

In HTML you tell the browser what it should display, and not how to do it. With the previous example of WinForms programming, it is interesting that XAML is declarative. Microsoft took windows programming from being imperative to declarative with the transformation from WinForms to WPF.



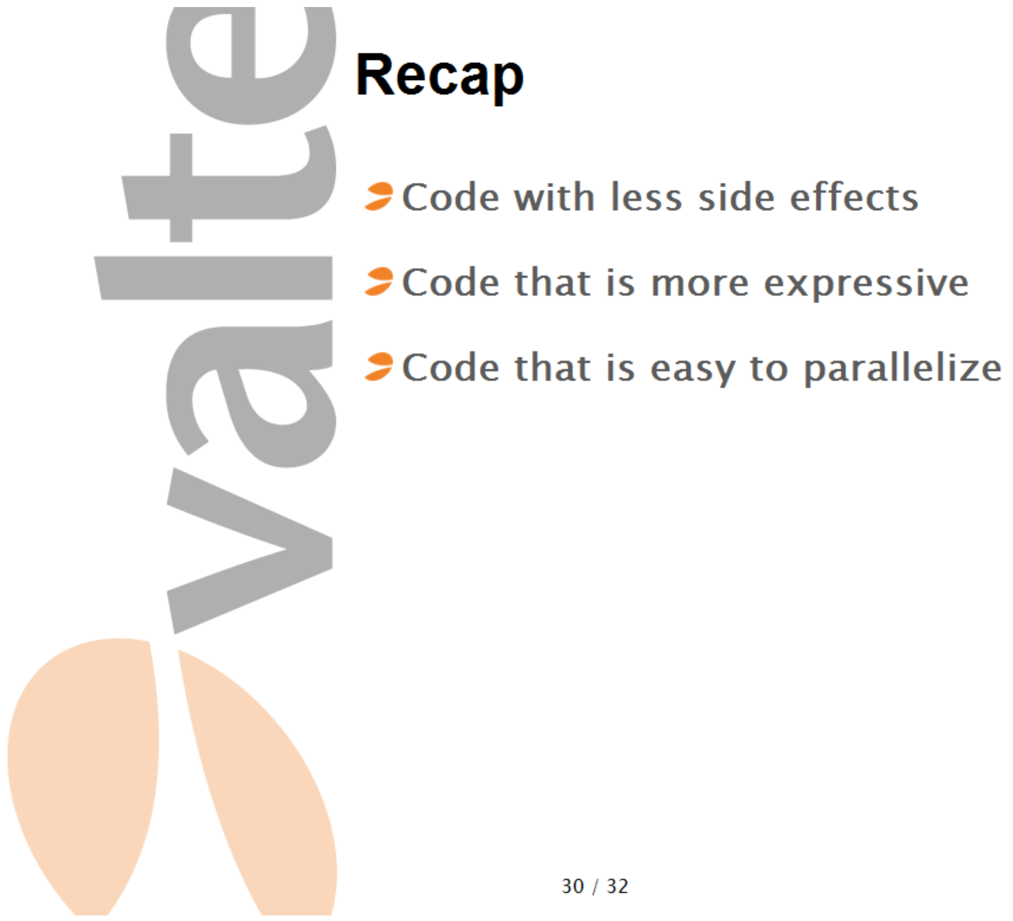
Declarative Programming

- Express the logic of a computation without describing its control flow
- Eliminate side effects
- Describe what computation should be performed and not how to compute it

Declarative programming should be independent on what order you declare things. A good example of this is XSLT style sheets where you define templates for rendering and it does not matter in what order you do things.

It eliminates side effects since you don't have any mutable state.

The trick is to define what instead of how.



Recap

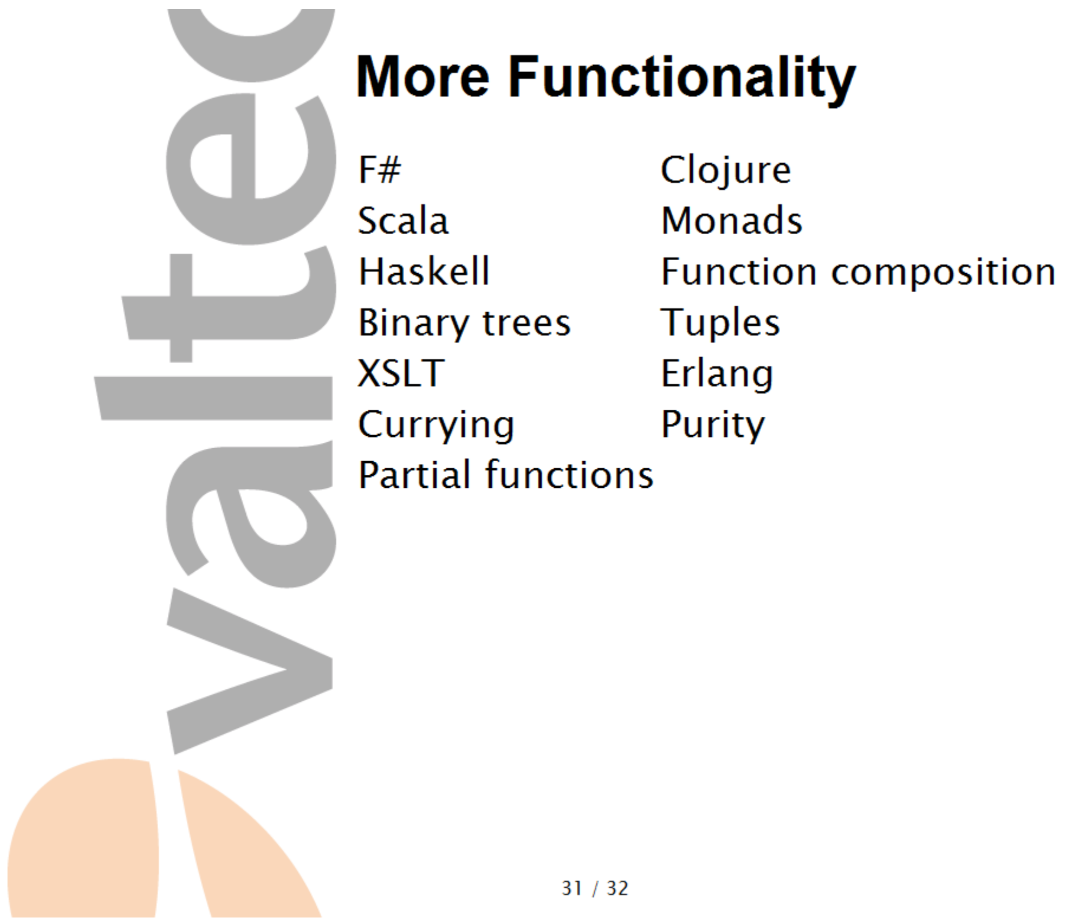
- Code with less side effects
- Code that is more expressive
- Code that is easy to parallelize

30 / 32

Using immutable values instead of mutating states causes less side effects that is a known source for bugs

Expressive code is easier to read as it states what is being done instead of how

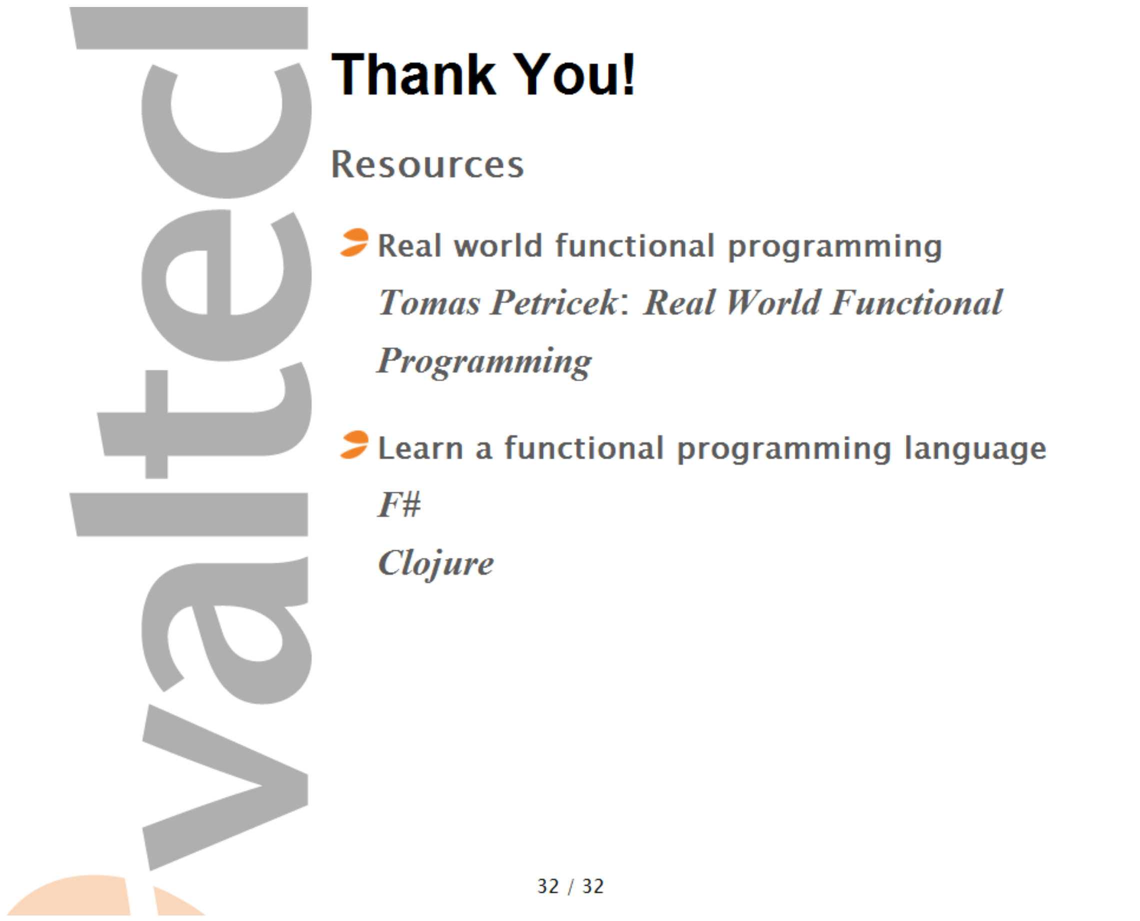
If we want to make parallelization embarrassing, we should use functional programming



More Functionality

F#
Scala
Haskell
Binary trees
XSLT
Currying
Partial functions
Clojure
Monads
Function composition
Tuples
Erlang
Purity

Things that did not make it into this talk.



Thank You!

Resources

- Real world functional programming

  Tomas Petricek: Real World Functional Programming

- Learn a functional programming language

  F#

  Clojure

32 / 32

Thank you for listening. Go to my blog litemedia.info for more info. Read Tomas Petricek's book about functional programming. Learn a functional programming language.