Functional Programming Patterns

Prof. Dr-Ing. Klaus Ostermann

Slides are in part adapted from a talk by Scott Wlaschin

OO design vs FP design ☺

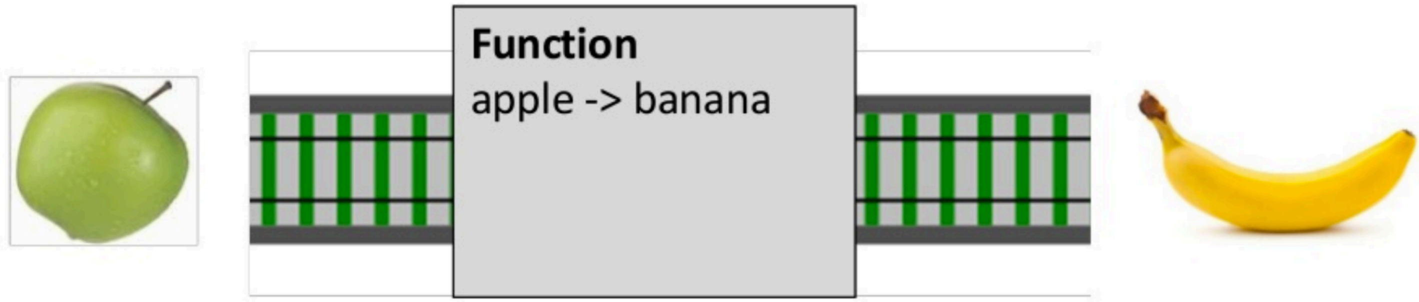
OO pattern/principle	FP equivalent
• Single Responsibility Principle	• Functions
• Open/Closed principle	• Functions
• Dependency Inversion Principle	• Functions, also
• Interface Segregation Principle	• Functions
• Factory pattern	• You will be assimilated!
• Strategy pattern	• Functions again
• Decorator pattern	• Functions
• Visitor pattern	• Resistance is futile!

Functional Design

- Core Principles
  - Functions
  - Types
  - Composition
- Functions as Values
- Monads
- Maps
- Monoids and folds

Functions are Values!

Functions are Values!



Functions are "stand alone" and not associated to a class

They are "first class": They are values, just like numbers or strings, and can hence be passed to or returned from other functions

Functions are values!

- A small Haskell session…

```
> let x = 1
> let add1 y = y+1
> let add1 = \y -> y+1
> let twice f x = f (f x)
> twice add1 x
3
```

Same keyword to bind names for all values, including functions

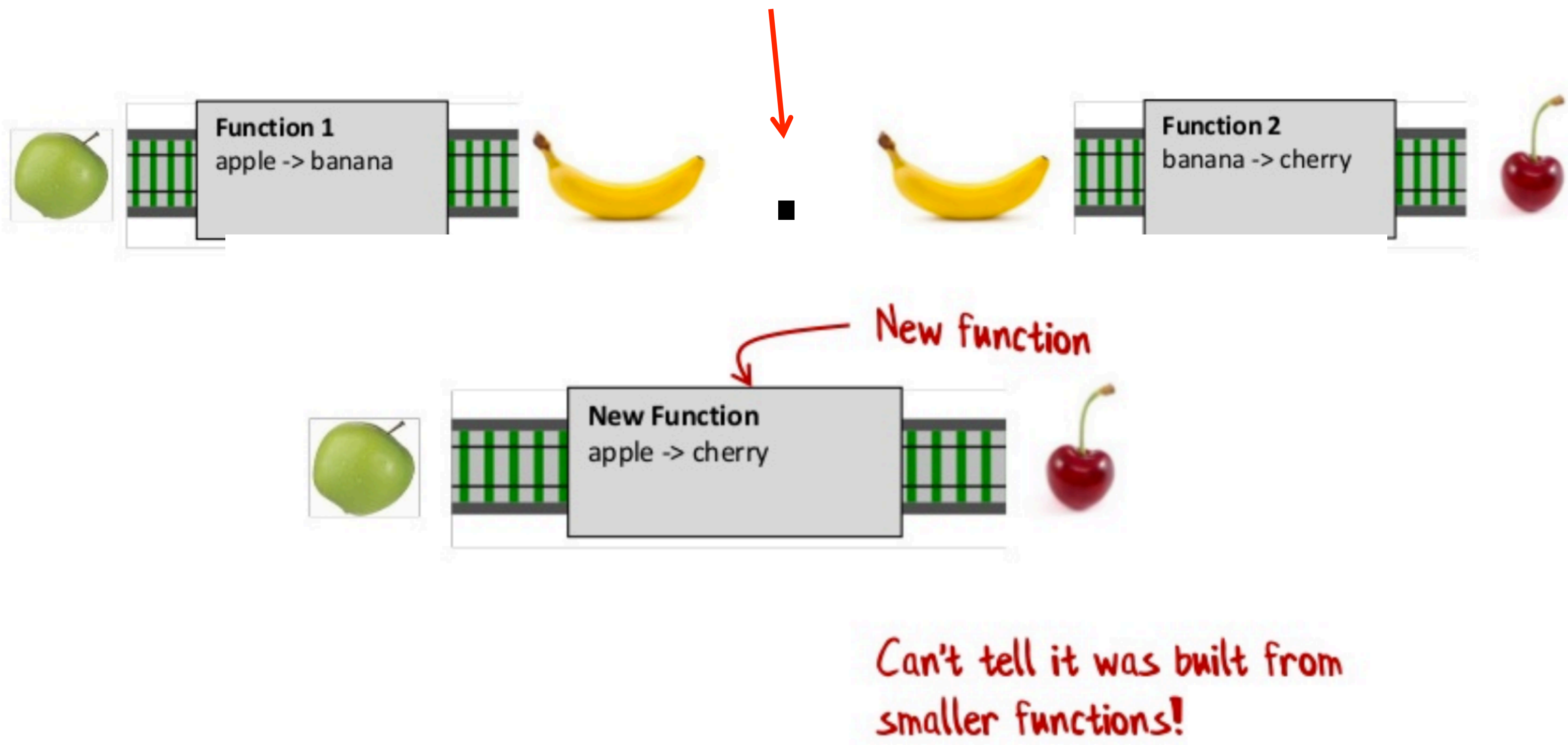
Lambda notation

Higher-Order functions

Composition everywhere!

Composition everywhere!

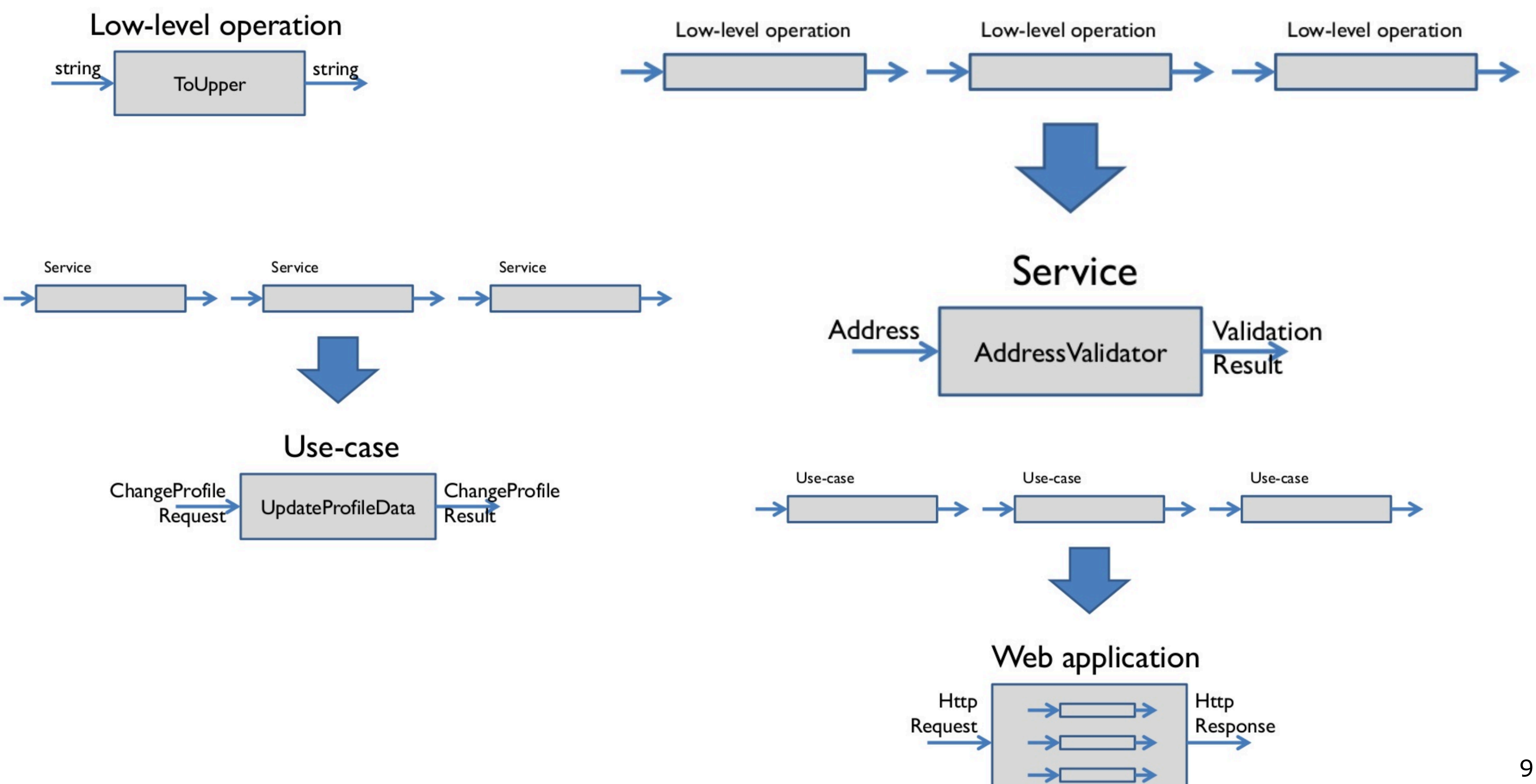
Function composition operator
(is itself a function)



Functions scale!

"Functions in the small, objects in the large"?

No. Due to composability, functions
can be used on every abstraction level!



Types are not classes

What are types?

- Types are "sets" of values
  - (we use quotes around "set" because it may not strictly be a set as defined in set theory)
- If an expression e has type T, then this is a prediction that evaluation of e yields a value that is a member of the "set" T.
- Function types denote mathematical functions (recall that a mathematical function is also a set, namely a relation that is deterministic, …)

- In FP, a type describes the structure of the set it denotes

- In most OO languages, a type is a class name

Nominal vs Structural typing

- In FP, a type describes the structure of the set it denotes
  - Two types are equivalent if they have the same structure
  - This is called “structural typing”
- In most OO languages, a type is a class name
  - Two types are equivalent if they are identical
  - This is called “nominal typing”
- Example: The types A and B are different in Java (a nominal system)

```
class A { int x; int y }
class B { int x; int y }
```

- Example: The types A and B are equivalent in Haskell (a structural system)

```
type A = (Int,Int)
type B = (Int,Int)
```

Composition of Types

- Types can be composed, too
- Standard type composition operators: Products and Sums
- Sum types correspond to disjoint unions
- Product types correspond to Cartesian products
- Examples for sum types:
  - **Either** type in Scala or Haskell
  - E.g., type Bool = Either Unit Unit
    type Maybe a = Either a Unit
- Sum types destructed via pattern matching
- Some languages also feature non-disjoint unions
  - Union types in C
  - Not type-safe

Composition of Types

- Examples for Product types
  - Tuples : (1,”hi”) has type (Int,String)
  - Records: (x = 5, y = 7) has type (x: Int, y: Int)

Products and Sums together: Algebraic data types

- E.g. in Haskell:

```
data Color = Red | Green | Blue

data Point = Point Float Float

data UniversityPerson = Professor String | Student Int String
```

Data of unbounded size via recursion

- With products and sums, we can only construct data types of fixed size
- To have things like lists, we need some form of recursion

- One way: Fixed point operator on the type level

```
data IntListF x = EmptyList | Cons Int x

type IntList = Fix IntListF
```

- More common way: Nominal types with recursion
- Algebraic data types allow recursion!

```
data IntList = EmptyList | Cons Int IntList
```

Algebraic data types vs. OO classes

- In OO languages, product types are formed by fields of classes
- Sum types are expressed via subtyping

Example:
data IntList = EmptyList | Cons Int IntList

can be expressed as

abstract class IntList

case class EmptyList() extends IntList
case class Cons(x: Int, rest: IntList) extends IntList

Important difference: Sum types in FP are usually closed, i.e., non-extensible, whereas sums expressed via subtyping are open

This is related to the “expression problem” we discussed earlier

Datatype-Generic Programming

- Observation: Many standard functions can in principle be derived from the shape of an algebraic data type
  - Equality of two types, various traversals, “folds”
- But we need to repeat those definitions for every datatype

- Way out: Datatype-generic programming
- We can’t address the topic in detail, see *www.cs.ox.ac.uk/**jeremy**.**gibbons**/publications/dgp.pdf* for a good tutorial
- Common idea: Express datatypes via “polynomial functors”
  - Functor: Function on the type level that comes with a “map” function
  - Polynomial functor: Type constructors that can show up in the functor restricted to product and sum operators
- E.g., instead of writing
  data ListF x = EmptyList | Cons Int x
  we can write
  type ListF X = 1 + (Int * X)
  (1 is something like Unit, 0 is something like Nothing)

Polynomial functors

- Using polynomial functors, standard data type isomorphisms coincide with standard identities known from basic algebra:

1 * 1 = 1
1 + 0 = 0 + 1 = 1
1 * 0 = 0
A * B = B * A
A * (B + C) = A * B + A * C
1 + 1 = 2

- Even standard rules for derivation of polynoms can be interpreted in the polynomial functor world

F(X) = X * X * X = X^3
F’(X) = 3 * X * X = 3 * X^2 = (X^2 + X^2 + X^2)
F(X) describes containers with three elements.
F’(X) describes the types of containers with three elements with a “hole”:
Either the left element is missing, or the middle one, or the right one
For more on this topic, consider
http://chris-taylor.github.io/blog/2013/02/10/the-algebra-of-algebraic-data-types/

FP pattern: Make effects explicit

- “Effect”: Things a function does in addition to (or instead of) computing a value
  - Example: Non-termination, I/O, Mutation of variables
- Idea: Type signatures should not “lie”
  - If a function signature promises to map every string to an integer, it should not sometimes return abnormally with an exception

- Example: In FP, an integer parsing function string2int would have a type like:
  string2int : String -> Maybe Int

  instead of

  string2int: String -> Int

  and sometimes throwing an exception

What's good about explicit effects?

- Making effects explicit reduces or eliminates the dependence of the program result on the order of evaluation

Example:

```
val myfunc = try {
  fun (x: String) => if … then throw SomeException …
} catch (SomeException e) …
```

The exception handler won't work if the exception is thrown in a part of the code whose evaluation is deferred, e.g., by being inside a function body

Explicit effects

- Explicit effects are only a “design pattern” in some FP languages; in other languages they are (partially) enforced by the type system
  - Haskell, Clean, Idris, …
  - For instance, Haskell enforces explicit mutation and I/O, but does not enforce termination
  - For instance, a Haskell function of type Int -> Int will, given input x
    - Either diverge on x
    - Or return another integer y but not perform any mutation, not print something on the screen, not write something to your harddrive, not communicate on the network
- There are several advanced FP “patterns” for dealing with explicit effects in an elegant way
  - Monads, effect types, algebraic effects, …
  - Not in the scope of this lecture

Back to basic FP...

- Meta-Pattern in FP: “We can parameterize/abstract over anything”

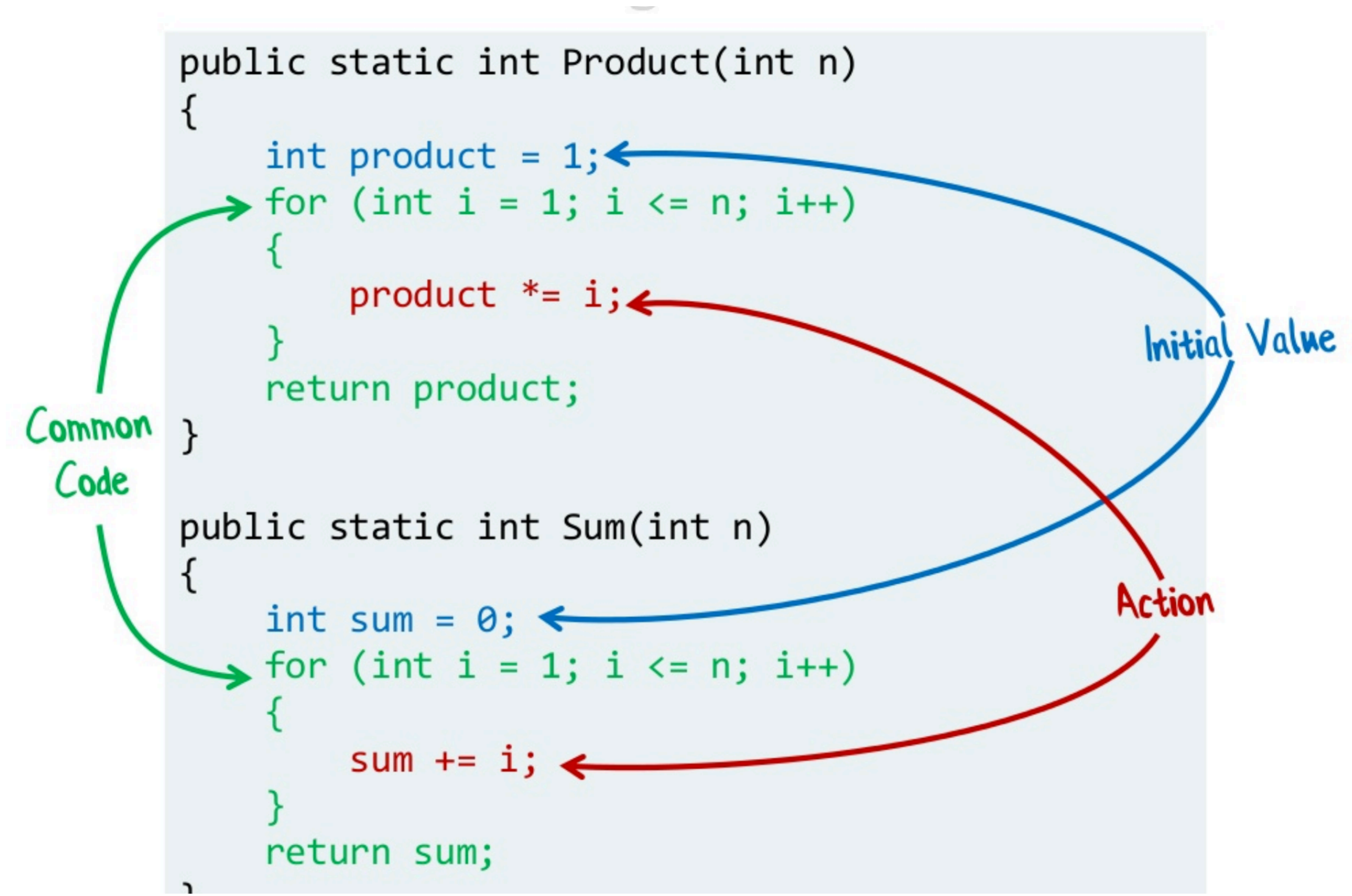
- Concrete instance of the meta-pattern: Parameterize over functions
- Example: FP programmers hate this kind of redundancy

```
public static int Product(int n)
{
    int product = 1;
    for (int i = 1; i <= n; i++)
    {
        product *= i;
    }
    return product;
}

public static int Sum(int n)
{
    int sum = 0;
    for (int i = 1; i <= n; i++)
    {
        sum += i;
    }
    return sum;
}
```

Don't Repeat Yourself

Parameterize all things...



Parameterize all the things

```
let product n =
    let initialValue = 1
    let action productSoFar x = productSoFar * x
    [1..n] |> List.fold action initialValue


let sum n =
    let initialValue = 0
    let action sumSoFar x = sumSoFar+x
    [1..n] |> List.fold action initialValue
```

Initial Value

Parameterized action

Common code extracted

Lots of collection functions like this: "fold", "map", "reduce", "collect", etc.