Object-Oriented vs. Functional Programming with C# and F#

Urs Enzler
Software Architect
Calitime AG
@ursenzler

Laika – our mascot

Context

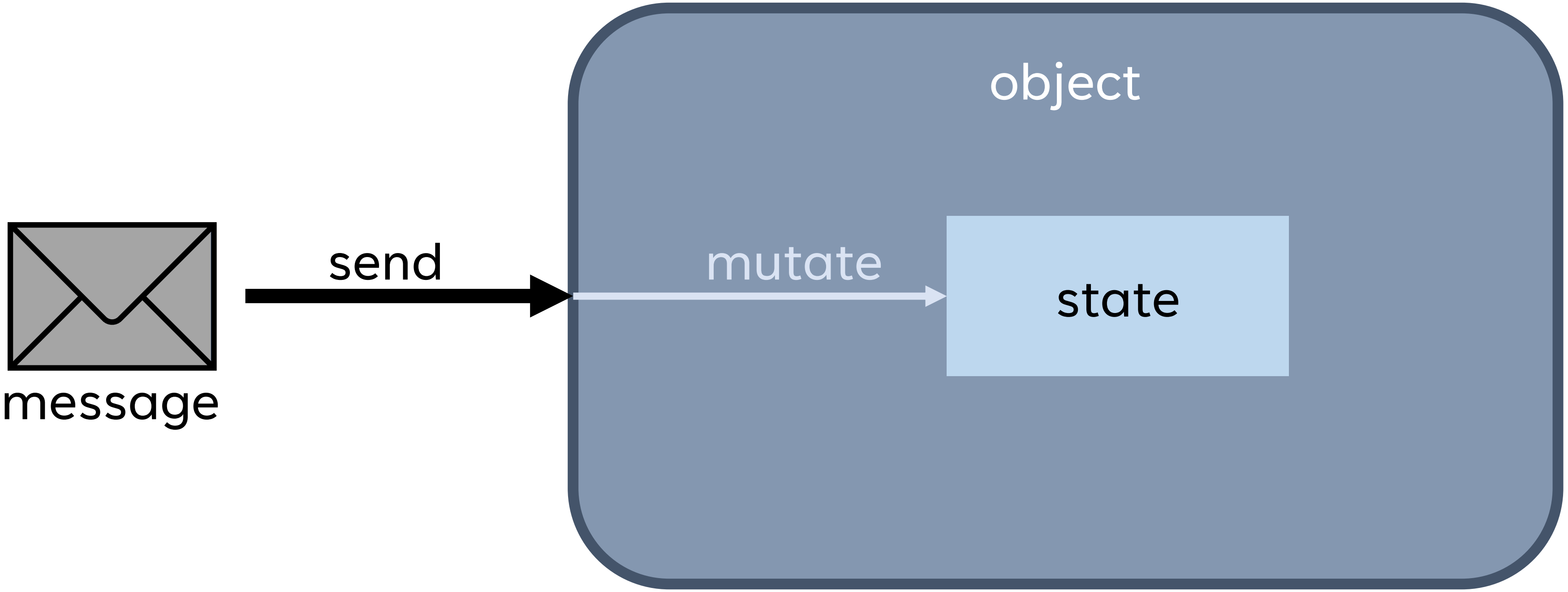
A typical business application.

Modern interpretation of OOP and FP
(not what stands in Wikipedia)

> 20 years of experience in C#
3 years of experience in F#

.NET

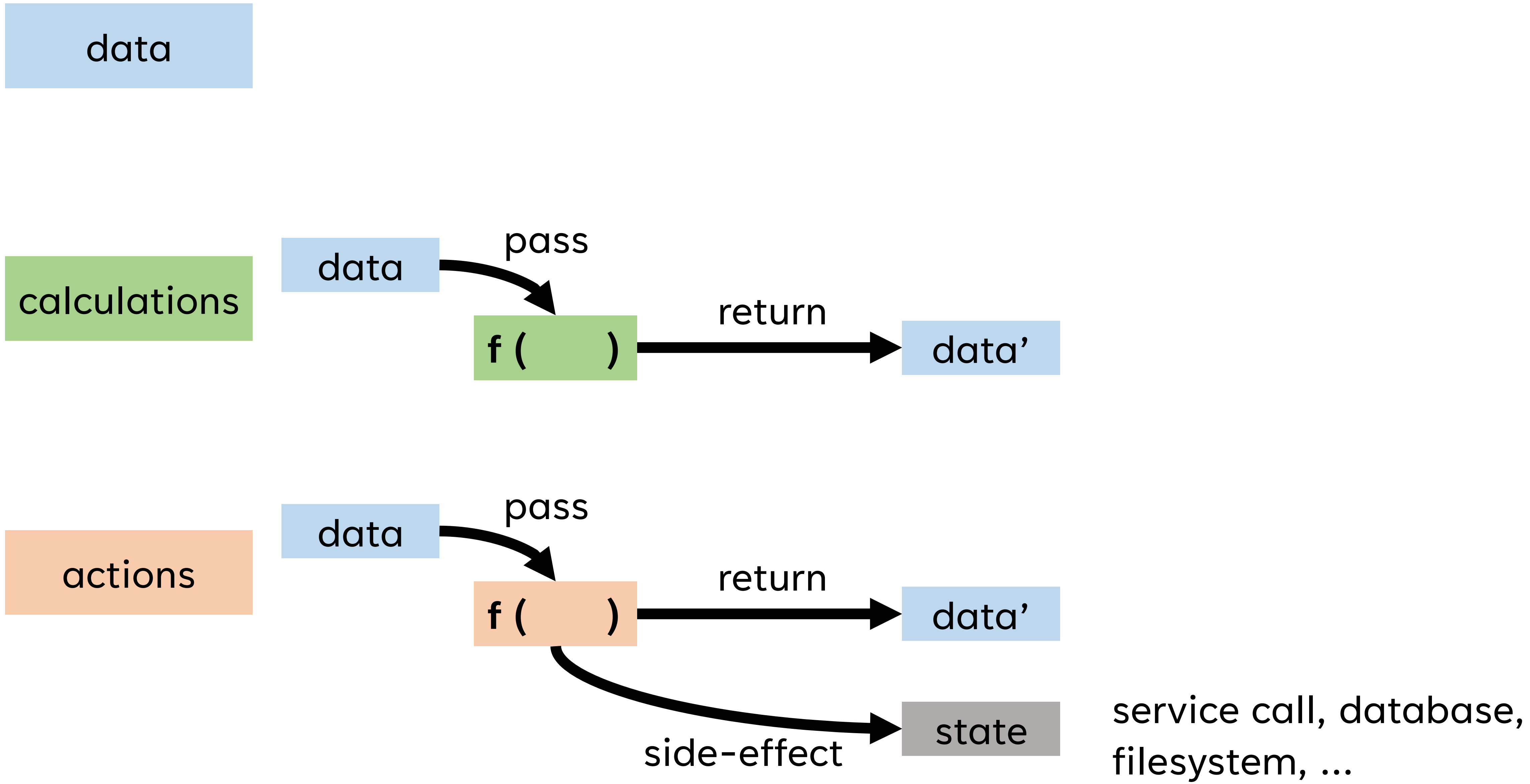
OOP (simplified)



C#

```
public class Object
{
    private int state;

    public void MultiplyBy(int multiplier)
    {
        this.state *= multiplier;
    }

    public void WriteToConsole()
    {
        Console.WriteLine($"{this.state}");
    }
}
```

FP (simplified)



```fsharp
let value = 42

let calculation value = value * 2

let action value =
    Console.WriteLine $"{value}"
```

OOP and FP use different concepts to model the domain and to design the solution.

OOP

Encapsulating and mutating state

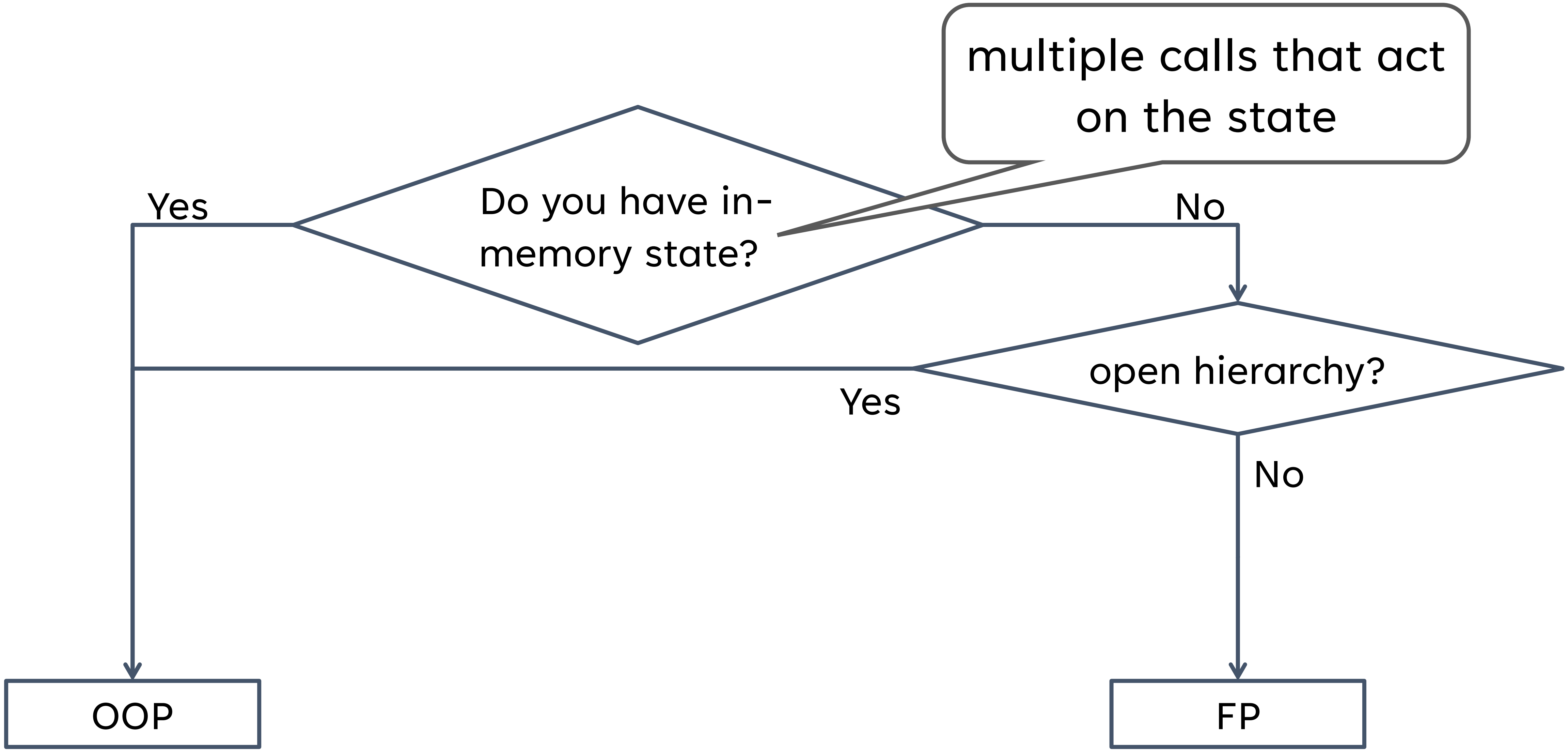
FP

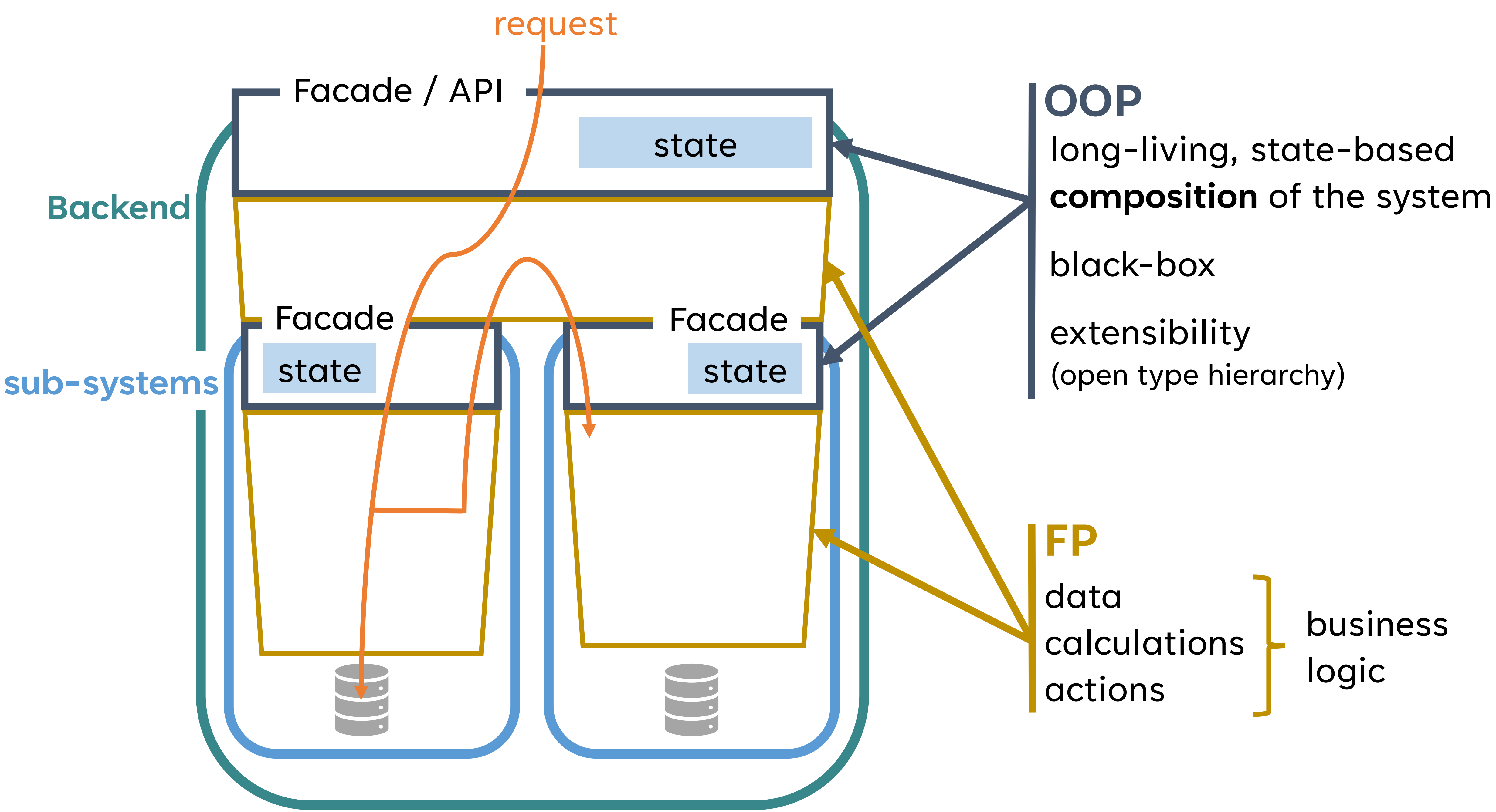
Calculations on immutable data and actions (side-effects)

thinking in

OOP	FP
classes	records
interfaces	discriminated unions
polymorphism	exhaustive pattern matching
mutability by default	immutability by default
methods mutating encapsulated state	functions transforming immutable data
	pipelines

Where is it easier to think in OOP-style and where in FP-style?





request
Facade / API
state
Backend
Facade
state
Facade
state
sub-systems
OOP
long-living, state-based
composition of the system
black-box
extensibility
(open type hierarchy)
FP
data
calculations
actions
business
logic

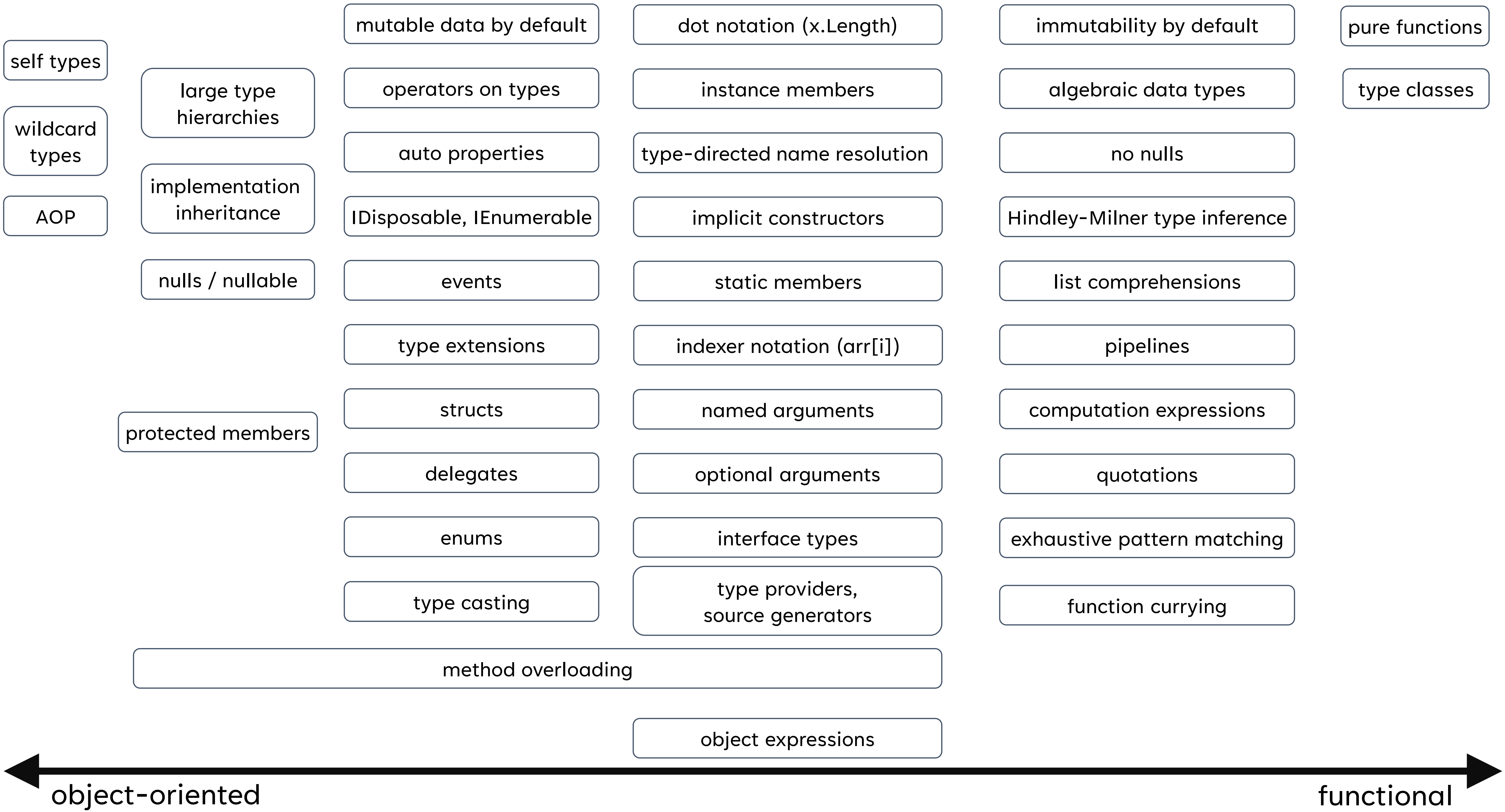
C#

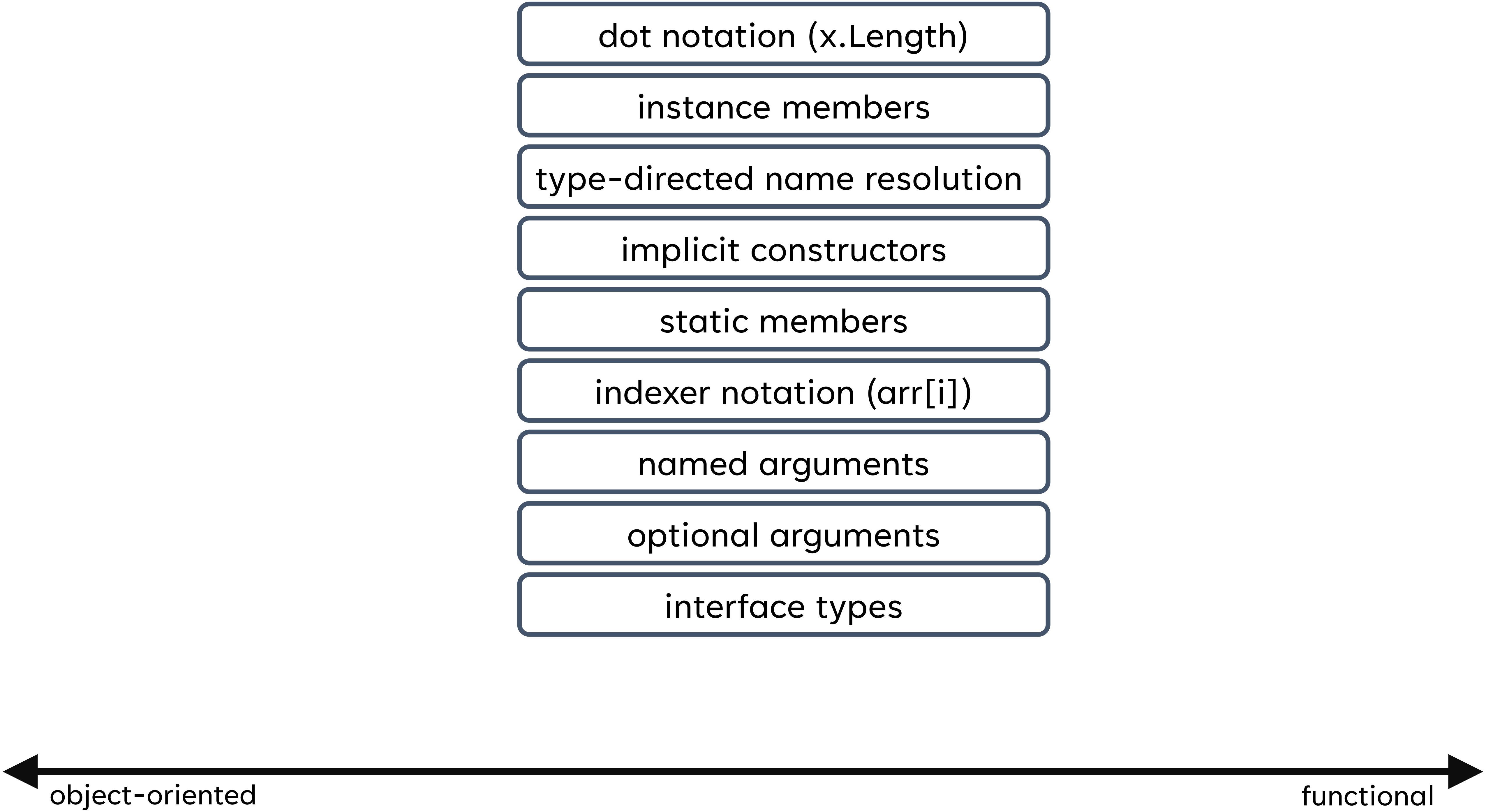
```
public class AttendanceTimeFacade
{
    private readonly Storages storages;
    private readonly OperationRunner operationRunner;
    private readonly IZeitmeldungChangedNotifier zeitmeldungChangedNotifier;

    public AttendanceTimeFacade(
        Storages storages, OperationRunner operationRunner, IZeitmeldungChangedNotifier zeitmeldungChangedNotifie
    {
        this.storages = storages;
        // ...
        this.zeitmeldungChangedNotifier = zeitmeldungChangedNotifier;
    }

    public virtual Task<OperationResult> AddZeitmeldung(
        OperationGuid operationId,
        ZeitmeldungGuid guid,
        AddZeitmeldungOperationData operationData,
        OperationMetadata metadata) =>
        AddZeitmeldungOperation
            .AddZeitmeldung(this.storages, this.abschlussChecker, this.operationRunner, this.zeitmeldungChangedNotifier,
                operationId, guid, operationData, metadata);
```

```fsharp
type TagsFacade
  (
    operationLogger : OperationLogger,
    rollbackOperationMessageSender : IRollbackOperationMessageSender,
    storage : TagsStorages,
    checkAuthorization : Bridge.CheckAuthorization
  ) =

  let startContext =
    { OperationRunner.StartContext.OperationLogger = operationLogger
      OperationRunner.StartContext.RollbackOperationMessageSender = rollbackOperationMessageSender }

  member self.ExecuteOperation(operationData : TagsOperationData) =
    match operationData with
    | CreateTag d -> createTag storage.Tag.QueryEvents storage.Tag.PersistEvent startContext checkAuthorization d
    | DeactivateTag d -> deactivateTag storage.Tag.PersistEvent startContext checkAuthorization d
    | ActivateTag d -> activateTag storage.Tag.PersistEvent startContext checkAuthorization d

  member self.QueryAllTags() =
    QueryTags.queryAllTags storage.Tag.QueryEvents
```

How well do C# and F# support OOP and FP concepts?

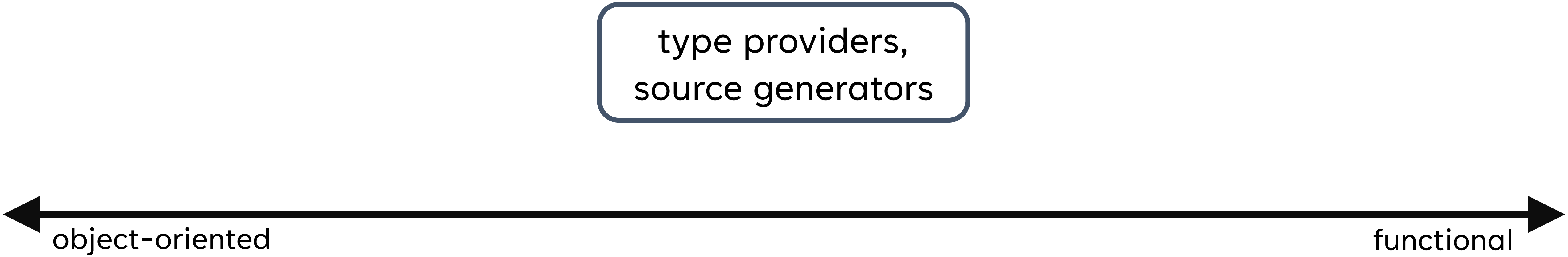
an overview of language features


self types
wildcard types
AOP
large type hierarchies
implementation inheritance
nulls / nullable
protected members
mutable data by default
operators on types
auto properties
IDisposable, IEnumerable
events
type extensions
structs
delegates
enums
type casting
dot notation (x.Length)
instance members
type-directed name resolution
implicit constructors
static members
indexer notation (arr[i])
named arguments
optional arguments
interface types
type providers, source generators
method overloading
object expressions
immutability by default
algebraic data types
no nulls
Hindley-Milner type inference
list comprehensions
pipelines
computation expressions
quotations
exhaustive pattern matching
function currying
pure functions
type classes
object-oriented
functional

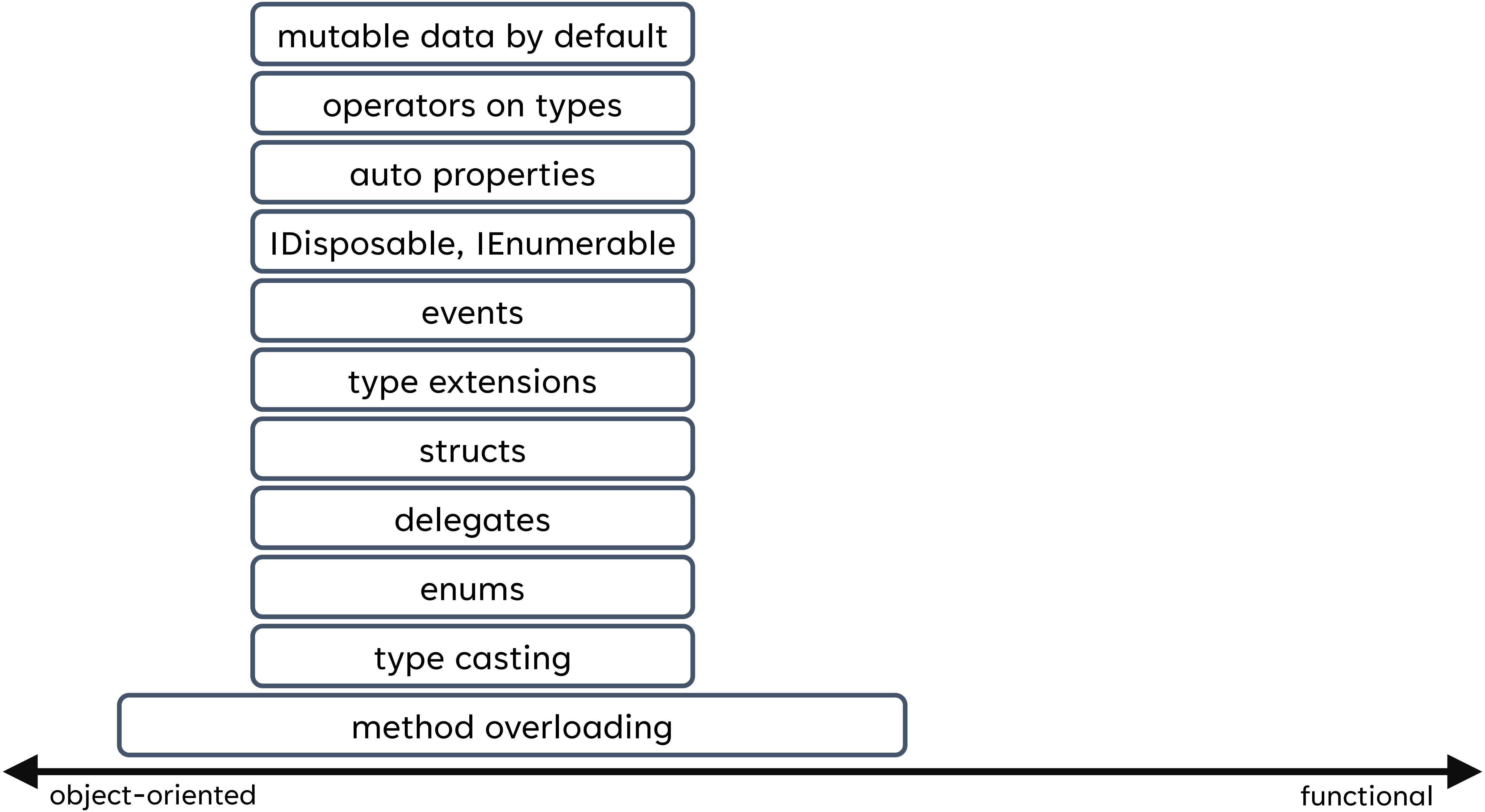

dot notation (x.Length)
instance members
type-directed name resolution
implicit constructors
static members
indexer notation (arr[i])
named arguments
optional arguments
interface types
object-oriented
functional

```fsharp
type PeopleDB = CsvProvider<"people.csv">

let printPeople () =
    let people = PeopleDB.Load("people.csv") // this can be a URL

    for person in people.Rows do
        printfn $"Name: %s{person.Name}, Id: %i{person.Id}"
```

C# “alternative”: Source generators



```fsharp
type ICalculator =
    abstract Sum: x: int -> y: int -> int

let calculation () =
    let calculator = { new ICalculator with member this.Sum x y = x + y }
    calculator.Sum 3 8
```


mutable data by default
operators on types
auto properties
IDisposable, IEnumerable
events
type extensions
structs
delegates
enums
type casting
method overloading
object-oriented
functional


large type hierarchies
implementation inheritance
nulls / nullable
protected members
object-oriented
functional


self types
wildcard types
AOP
object-oriented
functional

immutability by default

C#

```
public record ImmutableRecord(int I);

public class ImmutableClass
{
    public ImmutableClass(int i) { I = i; }
    public int I { get; }
}
```

F#

```
let calculation () =
    let value = 3

    let mutable variable = 7
    variable <- 13

    value + variable
```

object-oriented ⟷ functional

algebraic data types

```fsharp
type Student = { Name: string; Age: int }
```

F#

```fsharp
type Temperature =
    | Celsius of float
    | Fahrenheit of int
```

```fsharp
let asText temperature =
    match temperature with
    | Celsius c -> $"{c}°C"
    | Fahrenheit f -> $"{f}°F"
```

closed type hierarchy (no need for _ => throw)

exhaustive pattern matching

object-oriented ⟷ functional

no nulls

```
let printEvenOrOdd (x: string option) =
    match option with
    | Some value -> printfn $"{value}"
    | None -> printfn “no value“


let x = Some “hello world with options“
let y = None
```

F#

object-oriented ⟷ functional

Hindley-Milner type inference

F#

```fsharp
let someWhenEven x =   int -> int option
    if x % 2 = 0 then Some x
    else None

let printEvenOrOdd x =   int -> unit
    match someWhenEven x with
    | Some even -> printfn $"{even} is even"
    | None -> printfn "value was odd"
```

object-oriented ⟷ functional

```
let list = [ 0; 2; 4; 6 ]
```

```
let list = [ for i in 0 .. 10 do i * 2 ]
```

F#

[0; 2; 4; 6; 8; 10; 12; 14; 16; 18; 20]

list comprehensions

object-oriented ⟷ functional

```fsharp
let add1 x = x + 1
let multiplyBy3 x = x * 3
let square x = x * x

let pipeline x =
    x |> add1 |> multiplyBy3 |> square
```

```fsharp
let pipeline' = add1 >> multiplyBy3 >> square
```

pipelines

object-oriented ⟷ functional

```fsharp
let fetchAndDownload url =
    async {
        let! data = downloadData url

        let processedData = processData data

        return processedData
    }
```

```fsharp
int }
let query students =
    query {
        for student in students do
        groupBy student.Age into group
        where (group |> Seq.length > 1)
        select (group.Key, group |> Seq.length)
    }
```

computation expressions

```fsharp
let continueWhenSome (optional: int option) =
    option {
        let! value = optional
        return value + 17
    }
```

```fsharp
let continueWhenSome' = Option.map (fun value -> value + 17)
```

object-oriented ⟷ functional

F#

```fsharp
let quotation = <@ 1 + 2 + 3 @>
```

quotations

C#

```csharp
Expression<Func<int>> expression = () => 1 + 2 + 3;
```

object-oriented functional

input input output

```
let f a b = a + b    int -> int -> int

let add1 = f 1    int -> int

let x = add1 2    int
```

F#

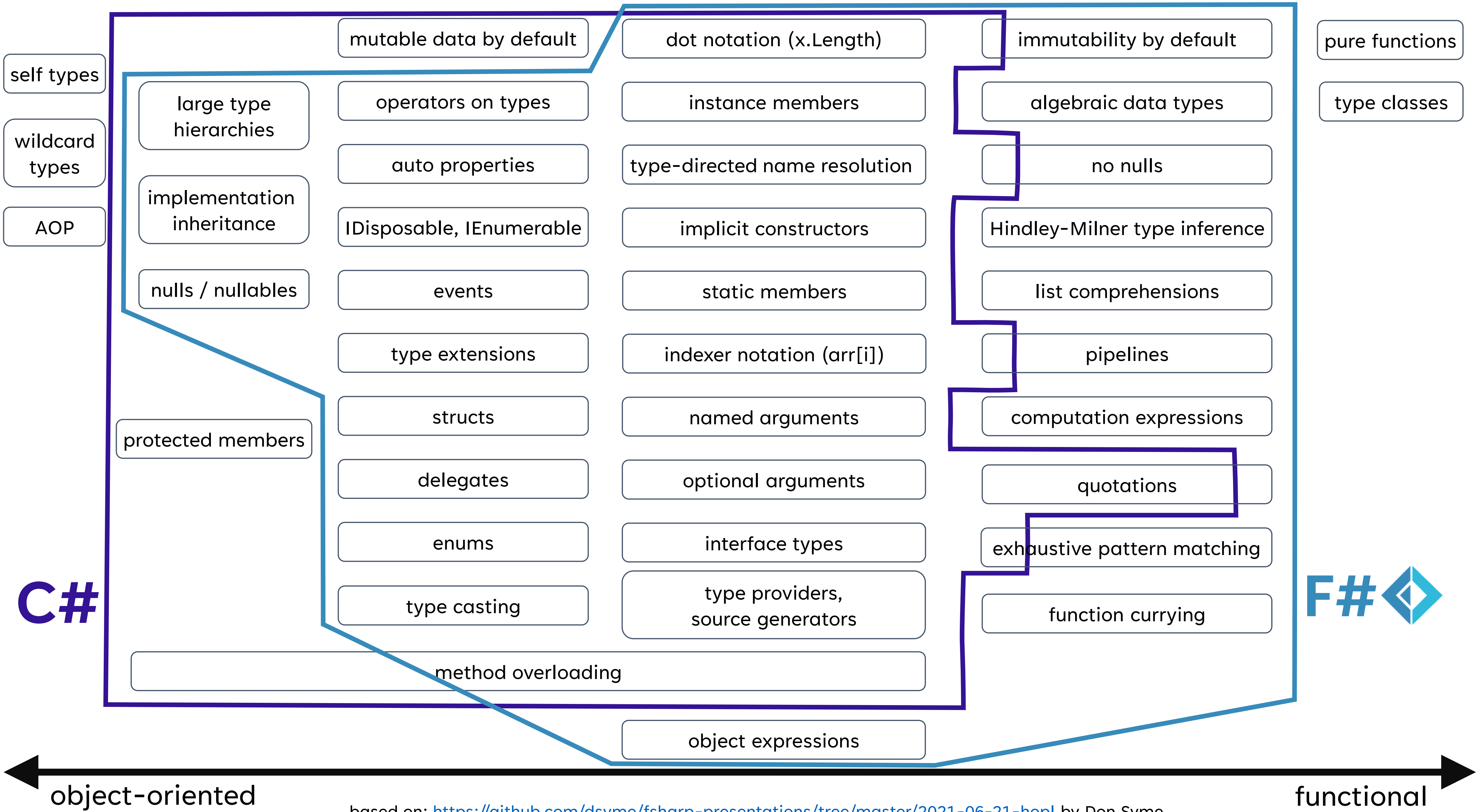
1 +

function currying

object-oriented

functional


pure functions
type classes
object-oriented
functional



based on: https://github.com/dsyme/fsharp-presentations/tree/master/2021-06-21-hopl by Don Syme

Are there other advantages?

C#

it’s the standard on .NET

more docu
(not necessarily better though)

most frameworks and libraries are
built with C# in mind

F#

easier refactoring

easier composition

easier to maintain

Interop between F# and C# is very good!

when to use

OOP

composition root
(when there is variability)

sub-system facades
("caches")

plug-ins
(external extensibility)

FP

business logic

domain modelling

algorithms

Mix concepts from both paradigms!

www.calitime.ch

So einfach geht Zeiterfassung heute!

Urs Enzler – u.enzler@calitime.ch
Software Architect
Calitime AG

@ursenzler
https://ursenzler.github.io