Concurrency paradigm and Functional Programming

BY RICCARDO TERRELL- @TRIKACE

“OOP makes code understandable by encapsulating moving parts. FP makes code understandable by eliminating moving parts”

— *Michael Feathers*
(author of Working with Legacy Code)

“It’s really clear that the imperative style of programming has run its course. … We’re sort of done with that. … However, in the declarative realm we can speculate a 10x improvement in productivity in certain domains”

— Anders Hejlsberg
(C# Architect)

“We cannot solve our problems with the same thinking we used when we created them”

- Albert Einstein

Agenda

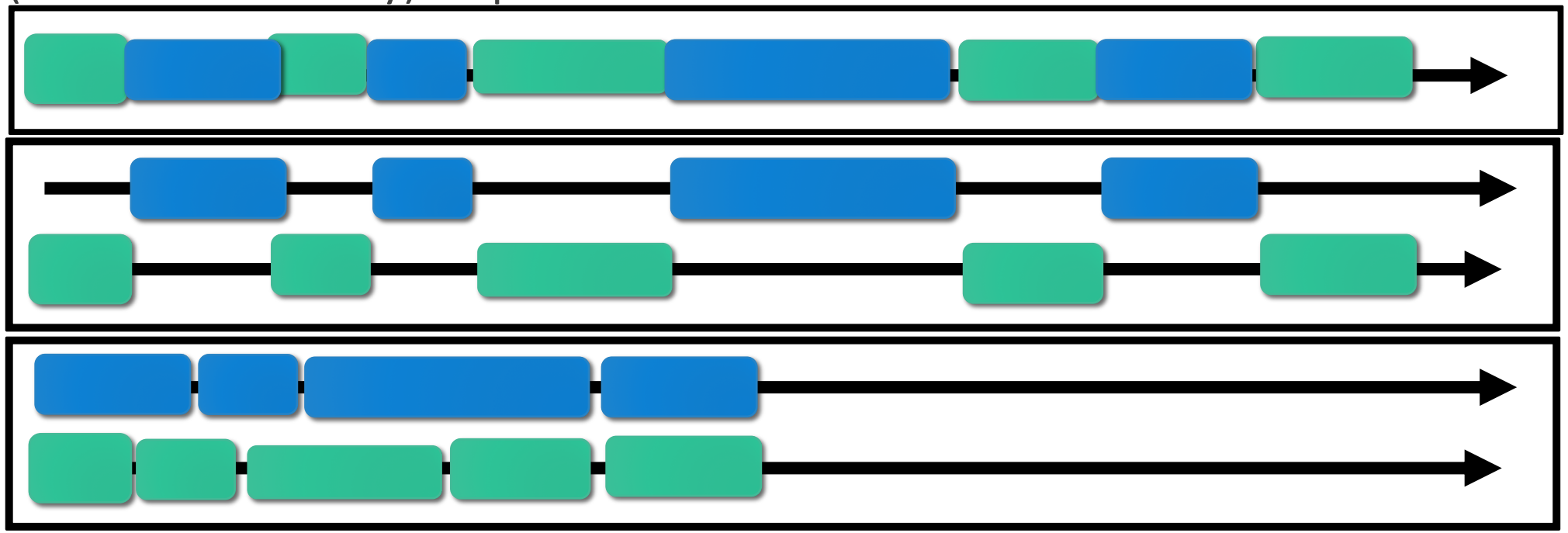
- Why & What concurrent programming
- The tenants of concurrent programming
- Influence of Functional Programming
- What is Functional Programming and motivation
- Functional Programming foundations

Objectives

- **Immutability** and **Isolation** are your best friends to write concurrent applications
- Use functions as building block and **composition** to solve complex problems with simple code
- Use independent computations as building block and composition to solve in parallel complex problems
- **Functional and imperative** styles should NOT stand in opposition … they **should *COEXIST***
  - Poly-paradigm programming is more powerful and effective than polyglot programming

Concurrency programming

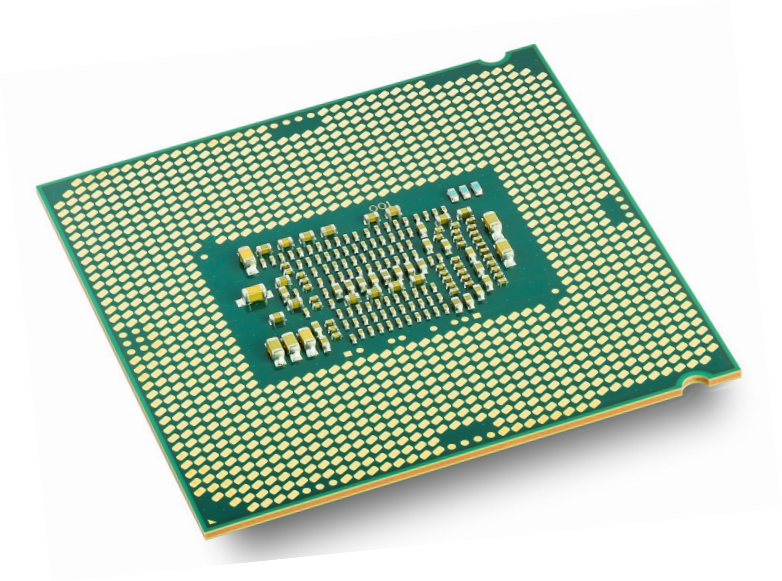
Concurrency provides a way to **structure a solution** to solve a problem that may (but not necessarily) be parallelizable.

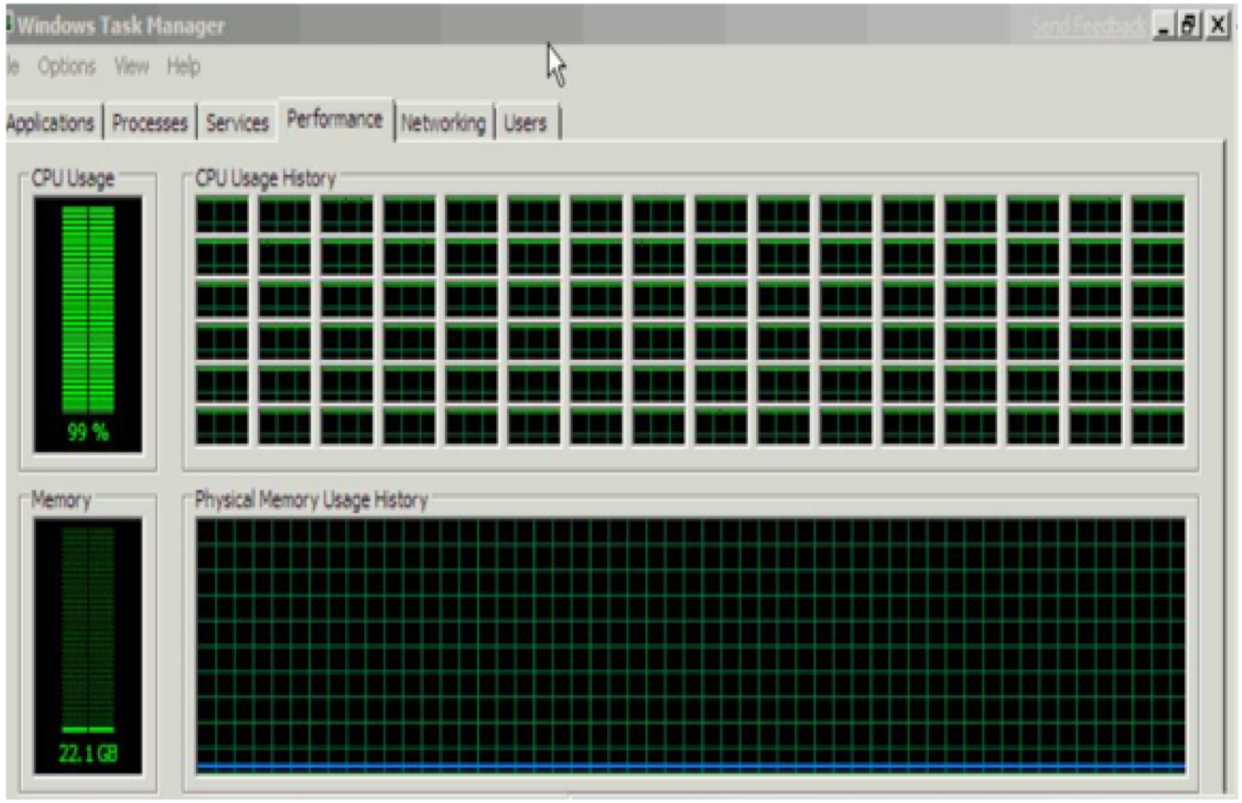


Concurrency is the **composition** of independently executing computations.

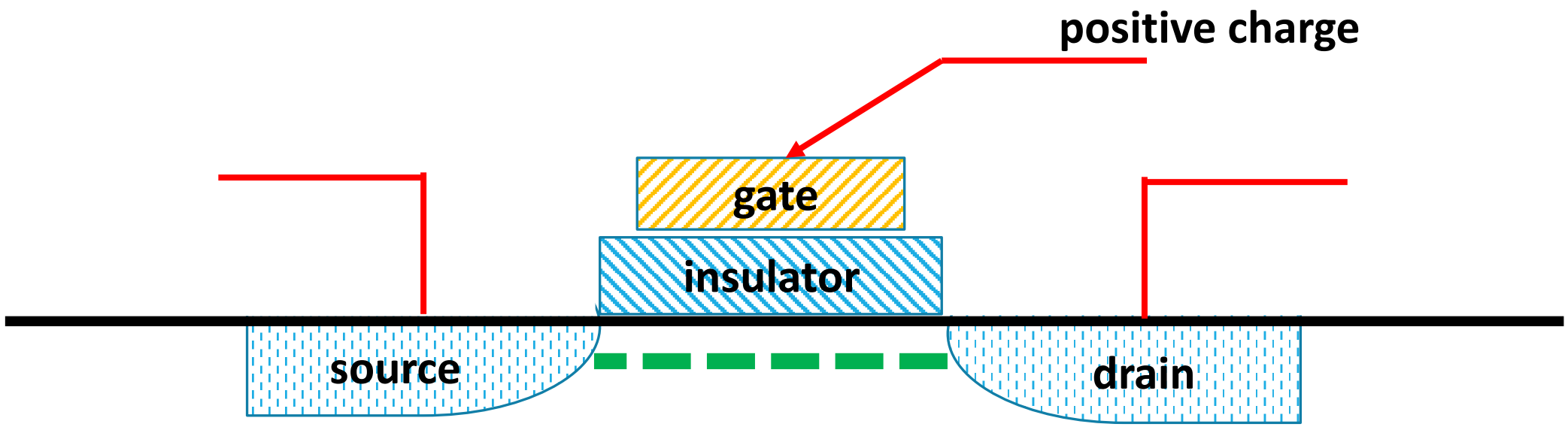
Moore’s law -The classical computer is reaching quantum state

"The number of transistors incorporated in a chip will approximately double every 24 months."
-- Gordon Moore, Intel Co- Founder

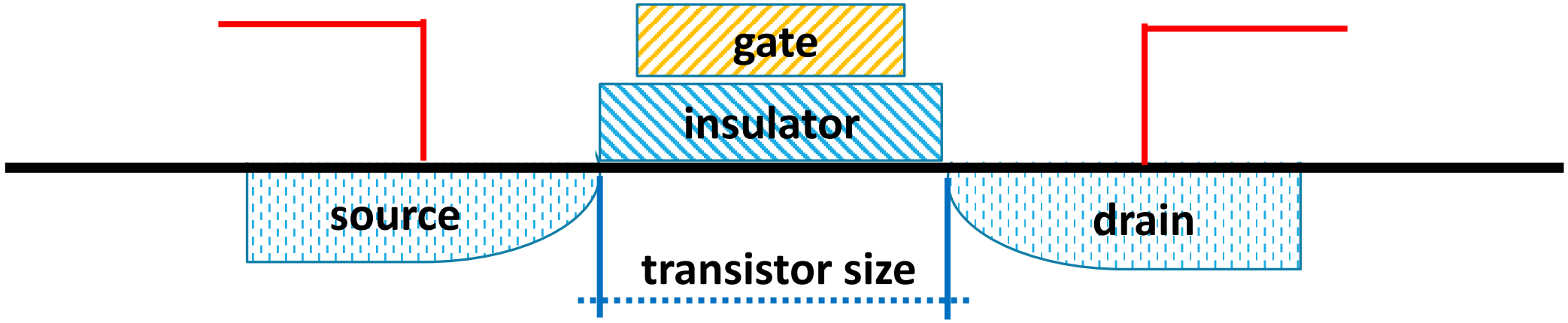




Transistors and the “End of Moore’s Law”

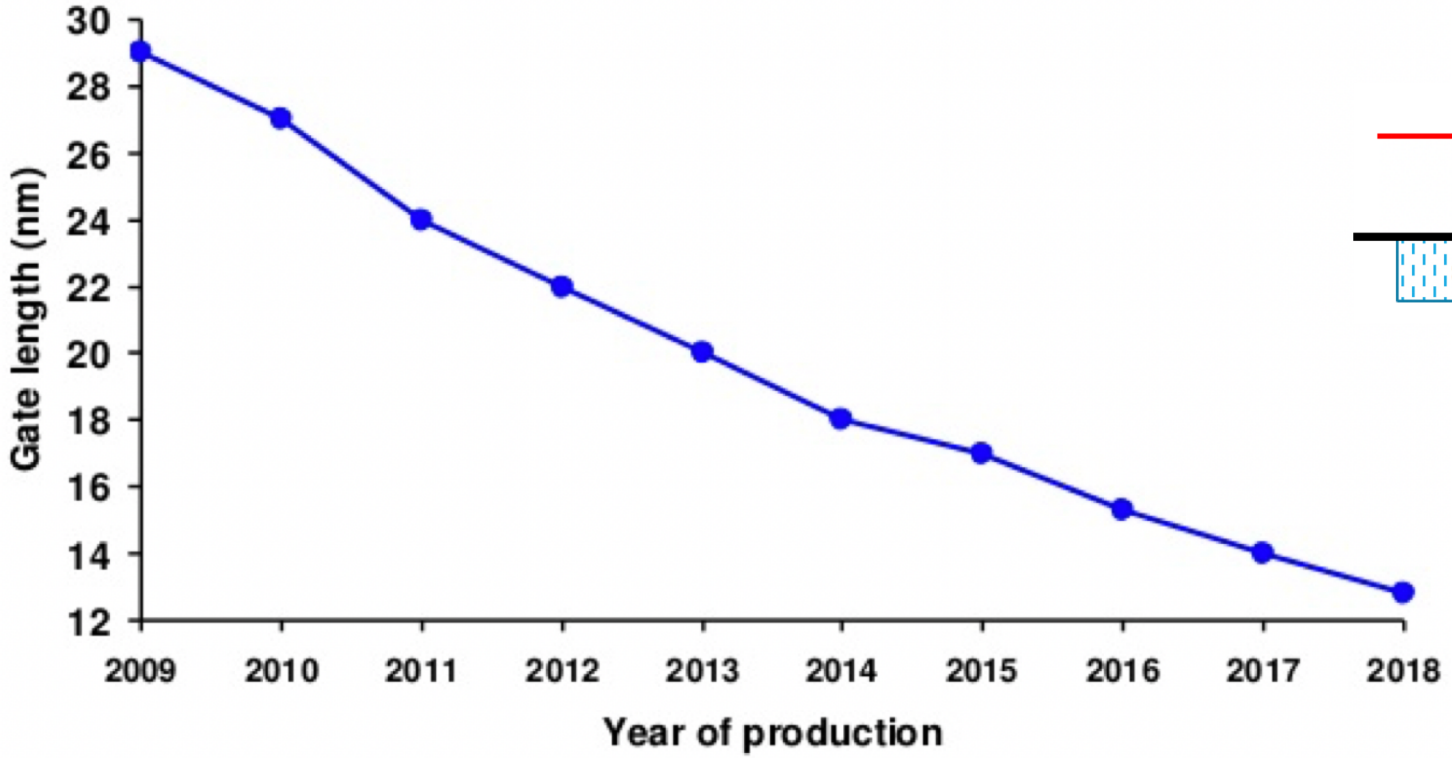


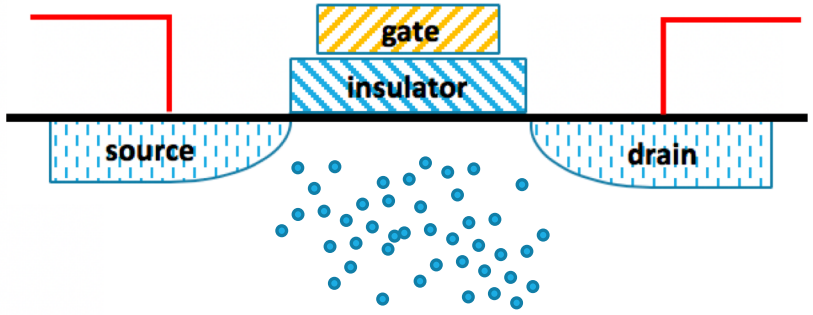
Transistors and the “End of Moore’s Law”



Transistors and the “End of Moore’s Law”

32 nm
22 nm
14 nm
10 nm
7 nm
5 nm
END





In a 22nm transistors there are about 50 silicon atoms between source and drain

Quantum tunneling

Parallel and Concurrent programming for the multicore future

MULTITHREADING
THEORY

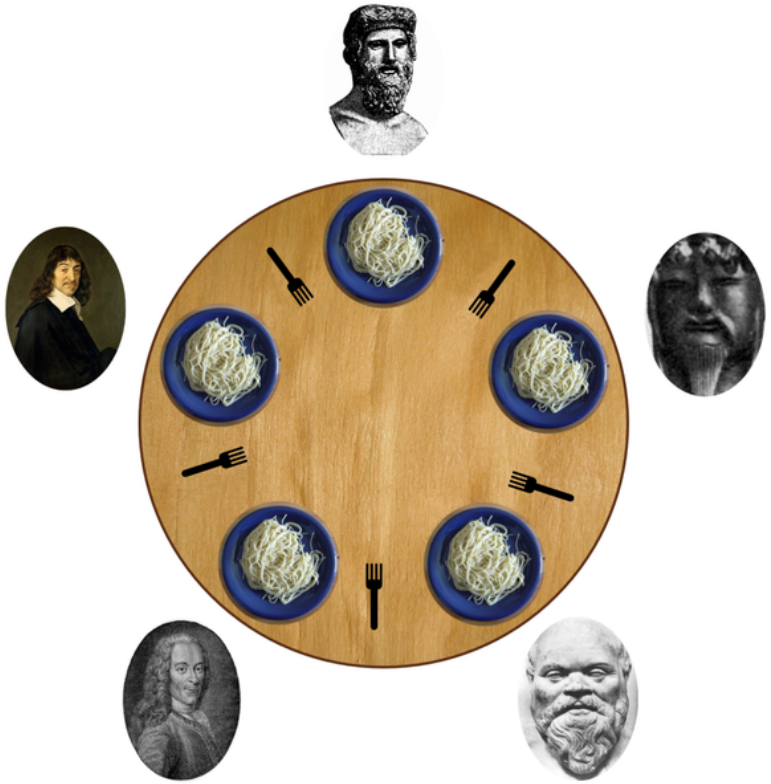
PRACTICE

Concurrent Programming is HARD

The issue is Shared Memory

- Shared Memory Concurrency
- Data Race / Race Condition
- Works in sequential single threaded environment
- Very difficult to parallelize
- Difficult to maintain and test
- Locking, blocking, call-back hell

Fully Synchronized Objects



Issues

Concurrent common issues

- Deadlock and starvation
- Livelock
- Race conditions

Data Race

Two concurrent accesses to a memory location where at least one of them is a write.

Example:

```
int x = 1;
Parallel.Invoke(
                 () => { x = 2; },
                 () => { System.Console.WriteLine(x);}
                );
```

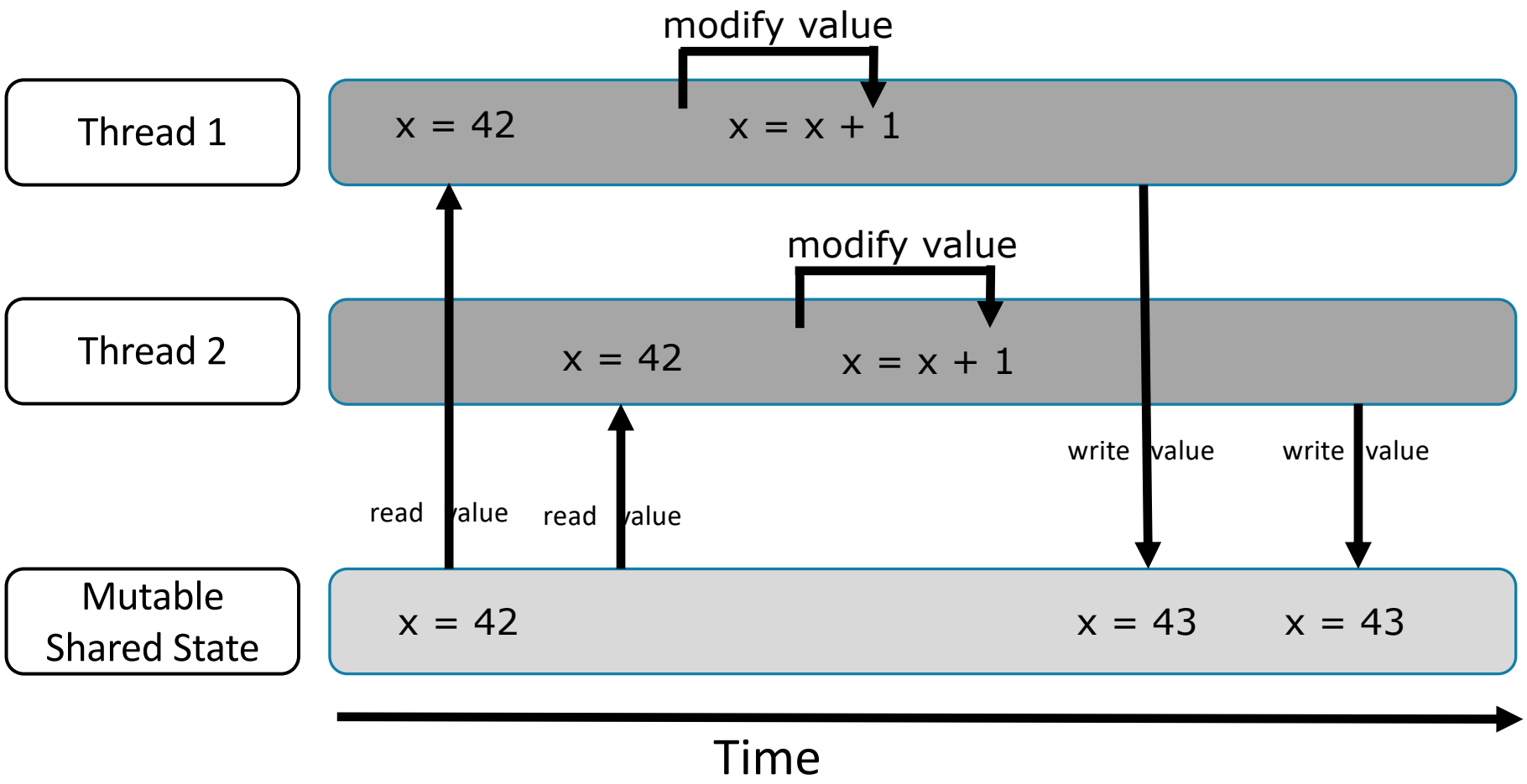
writes x

reads x

Possible outcomes?

- 1
- 2

Data Race


modify value
Thread 1
x = 42
x = x + 1
modify value
Thread 2
x = 42
x = x + 1
write value
write value
read value
read value
Mutable Shared State
x = 42
x = 43
x = 43
Time

Concurrency Primitives

```
public class BankAccount {
    Object _lock = new Object();
    int balance;

    public void adjustBalance(int amount) {
        lock(_lock) {
            balance = balance + amount;
        }
    }
```

Deadlock and Starvation

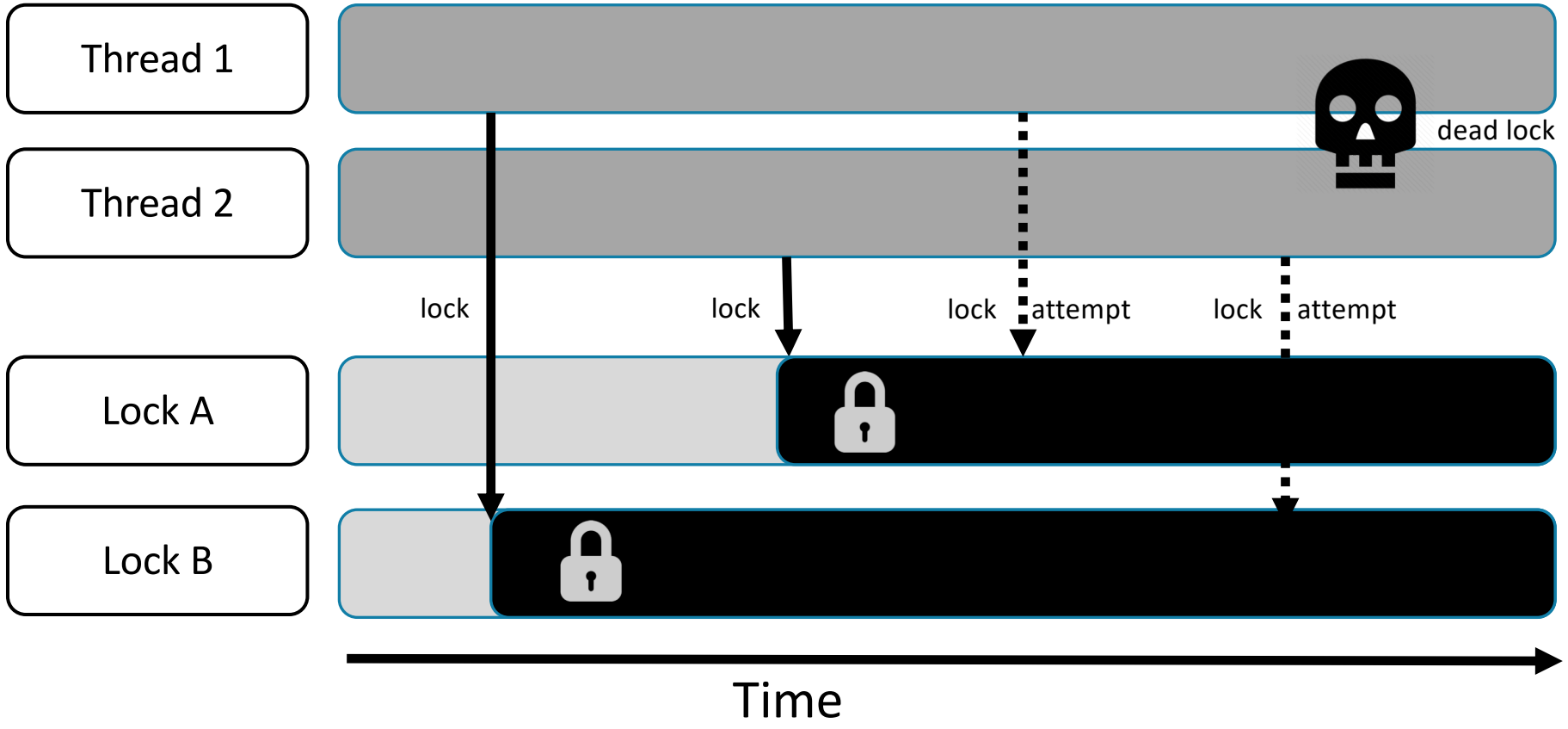
Deadlock – A situation where no further progress can be made by the program (when all processes are blocked).

Let *S* and *Q* be two locks

P_0	P_1
S.Lock();	*Q.Lock();*
Q.*Lock*();	*S.Lock();*
⋮	⋮
*S.Un*lock();	*Q.Unlock();*
Q.Unlock();	*S.Unlock();*

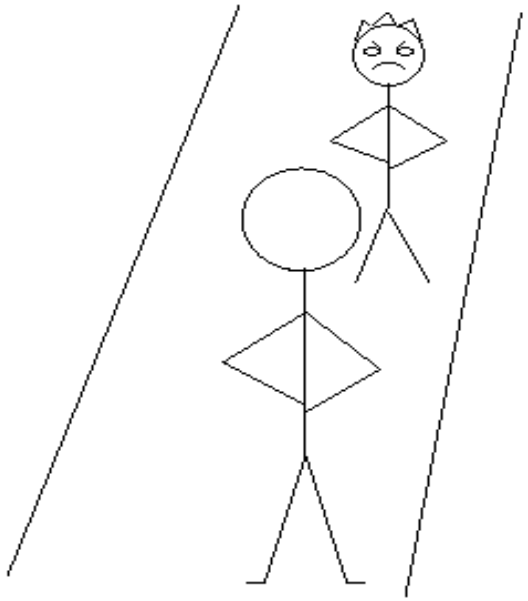
Starvation - A process is ready to run or to use a resource, but is never given a chance by the scheduler.

Dead-Lock


Thread 1
Thread 2
Lock A
Lock B
dead lock
lock
lock
lock
attempt
lock
attempt
Time

Live-Lock

Threads are doing something… yet they are not doing anything useful!!



Livelock when two polite people are in the hallway

Where is the Live-Lock?

```
// thread 1
getLocks_1_2(lock1, lock2)
{
  lock1.lock();
  while (lock2.locked())
  {
     lock1.unlock();
     wait();
     lock1.lock();
   }
   lock2.lock();
}
```

```
// thread 2
getLocks_2_1(lock1, lock2)
{
  lock2.lock();
  while (lock1.locked())
  {
     lock2.unlock();
     wait();
     lock2.lock();
   }
   lock1.lock();
}
```

Lock Convoys

Immutability

- Functional languages do not use mutable data or mutable states, and thus are inherently thread safe – the output value of a function depends entirely on its arguments.

- This frees up the programmer from having to manage complex thread synchronization / locks

- Like string in C# **everything** is immutable
- Help in concurrent/parallel code
- Immutability is **default** in F#
- You can choose to go mutable (**use carefully!**)

Copy-on-Write

Copy by value

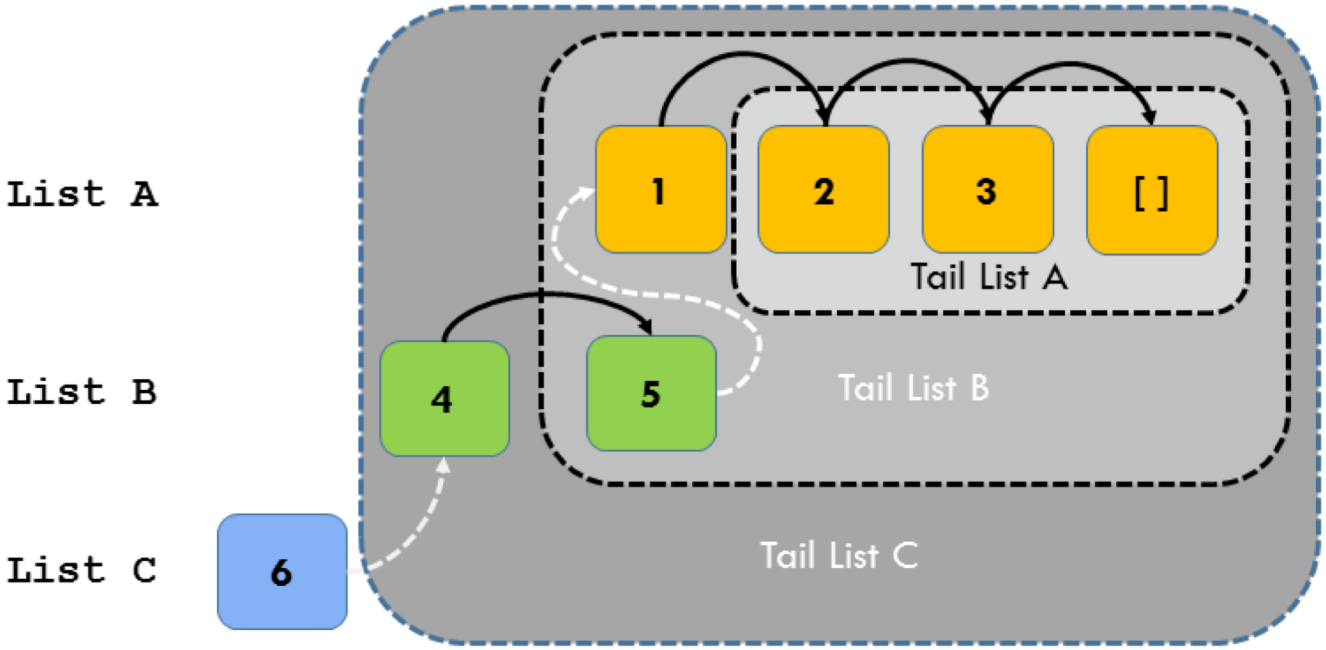
```
// The parameters x and y are passed by value.
static int Multiply(int x, int y)
{
    int z = x * y;
    System.Console.WriteLine("The value inside the method: {0}", z);
   return z;
}
```

If you are accessing the structure directly in a manner that doesn't involve copy-by-value semantics across multiple threads, then you have to take into account the thread-safety of the instance.

Benefits of immutable state

- Immutable objects classes lead into **composition abstraction**
- Immutability ensures that **objects are always in a correct state**.
- Immutability will make you **write less code**, the complexity associated with checking the current state of an object is not required, because it will never change
- Immutability makes it **easier to read and to reason about the code**
- Immutability will **minimize the number of bugs in your code**. Likewise it will **reduce tests**, which exist to validate mutation and to check the success of state changes.
  - The more your code is mutating state, the more testing is required to assert the correctness
- Immutable code **makes it easy to test**, because transformation occurs only in the object construction, immutable classes **make it trivial to write unit tests**, and easy to mock.
- **Immutable object are thread-safe in nature allowing to run correctly in parallel your program with minimal changes in the code base**

Structural Sharing



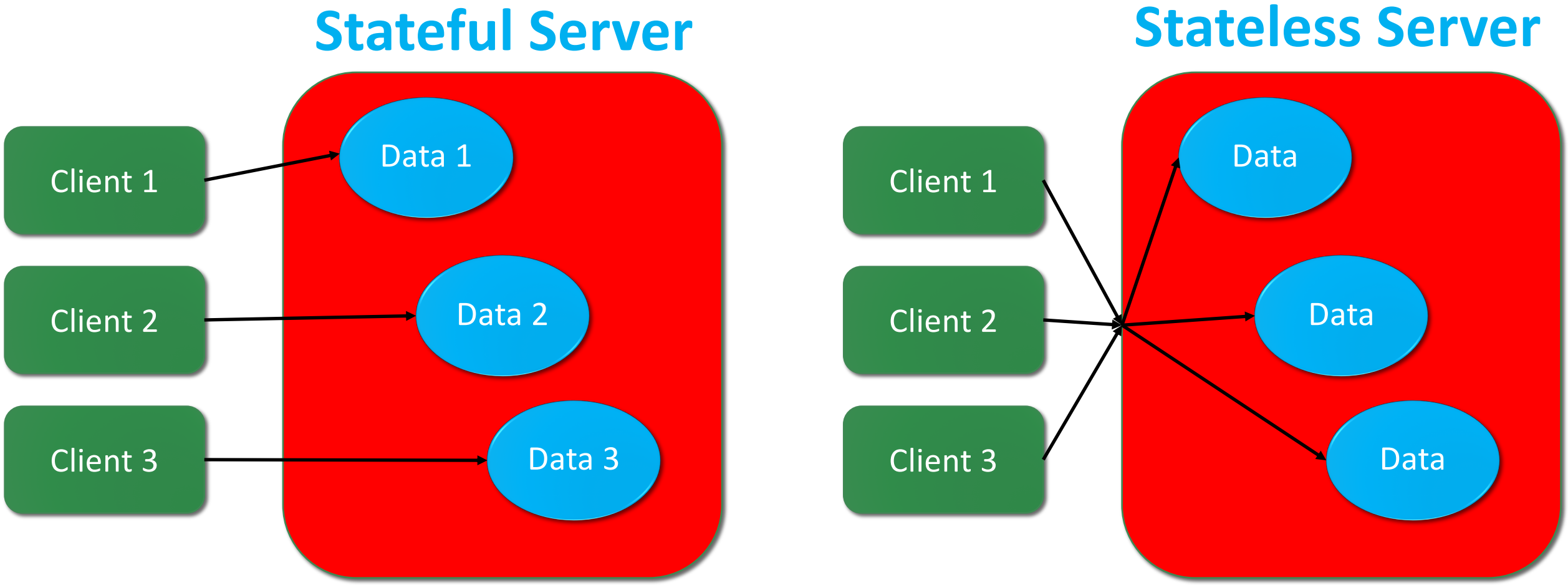
Isolation (Natural)

```
Bitmap image = new Bitmap(imagePath);

Task taskRed = Task.Factory.StartNew<Bitmap>(() =>
            {
                Bitmap bmap = (Bitmap)image.Clone();
                return SetColorFilter(bmap, ColorFilterTypes.Red);
            });
Task taskBlue = Task.Factory.StartNew<Bitmap>(() =>
            {
                Bitmap bmap = (Bitmap)image.Clone();
                return SetColorFilter(bmap, ColorFilterTypes.Blue);
            });
Task taskGreen = Task.Factory.StartNew<Bitmap>(() =>
            {
                Bitmap bmap = (Bitmap)image.Clone();
                return SetColorFilter(bmap, ColorFilterTypes.Green);
            });

Task.WaitAll(taskRed, taskBlue, taskGreen);
```

Isolation (Natural)

```
ThreadLocal<Bitmap> image = new ThreadLocal<Bitmap>(() => new Bitmap(imagePath));

Task taskRed = Task.Factory.StartNew<Bitmap>(() =>
                        SetColorFilter(image.Value, ColorFilterTypes.Red));
Task taskBlue = Task.Factory.StartNew<Bitmap>(() =>
                        SetColorFilter(image.Value, ColorFilterTypes.Blue));
Task taskGreen = Task.Factory.StartNew<Bitmap>(() =>
                        SetColorFilter(image.Value, ColorFilterTypes.Green));

Task.WaitAll(taskRed, taskBlue, taskGreen);
```

Stateless is an architecture pattern for scalable systems



Combining Immutability and isolation: the shared nothing approach

Immutability

```
public class ImmutablePerson
{
    public ImmutablePerson(string firstName, string lastName, int age) {
        FirstName = firstName;
        LastName = lastName;
        Age = age;
    }
    public string LastName { get; }
    public string FirstName { get; }
    public int Age { get; }

    public ImmutablePerson UpdateAge(int age)
    {
            return new ImmutablePerson(this.FirstName, this.LastName, age);
    }
}
```

Immutability

```
public class ImmutablePerson
{
    public ImmutablePerson(string firstName, string lastName, int age) {
        FirstName = firstName;
        LastName = lastName;
        Age = age;
    }
    public string LastName { get; }
    public string FirstName { get; }
    public int Age { get; }

    public ImmutablePerson UpdateAge(int age)
    {
            return new ImmutablePerson(this.FirstName, this.LastName, age);
    }
}
```

```
type Person = {FirstName:string; LastName:string; Age:int}
```

Passing immutable objects as messages between threads.

Concurrent modification ceases to be a problem!

Compare And Swap - CAS

```
void DoWithCAS<T>(ref T location, Func<T, T> generator)
                                                    where T : class
      {
         T temp, replace;
         do
         {
             temp = location;
             replace = generator(temp);
         } while (Interlocked.CompareExchange(ref location,
                                              replace, temp) != temp);
      }
```

CAS + Spin-Wait

```
public void Push(T item)
{
    var spin = new SpinWait();
    Node node = new Node { Value = item }, head;
    while (true)
    {
        head = m_head;
        node.Next = head;
        if (Interlocked.CompareExchange(ref m_head, node, head) == head) break;
        spin.SpinOnce();
    }
}
```

Compare And Swap - CAS

```
public class LockFreeStack<T> {
    private class Node {
        public T Data;
        public Node Next;
    }
    private Node head;

    public void Push(T element) {
        Node node = new Node {Data = element};
        DoWithCAS(ref head, h => {
            node.Next = h;
            return node;
        });
    }
```

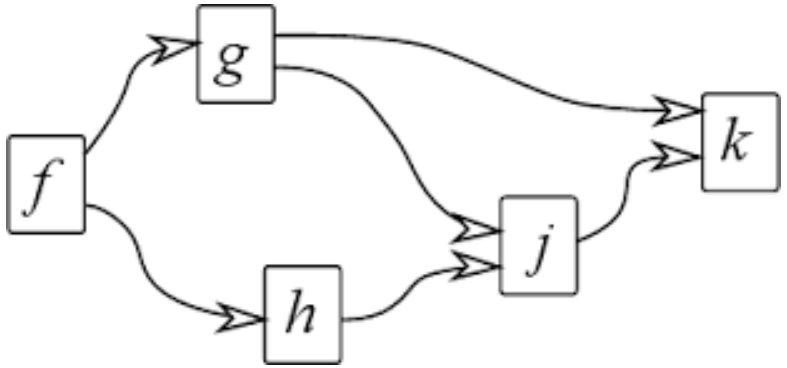
Parallel Loop & Compare And Swap - CAS

```
var results = new AtomImmutable<TResult>(ImmutableArray<TResult>.Empty);
Parallel.ForEach(partitioner,
    pos,
    () => new List<TResult>(),
    (item, loopState, local) =>
    {
        local.Add(selector(item));
        return local;
    },
    // final => results.Swap(o => o.AddRange(final))
    final => DoWithCas(ref final, o => o.AddRange(final))
    );
```

Referential transparency

An expression is said to be referentially transparent if it can be **replaced** with its corresponding value **without changing the program's behavior**.

As a result, evaluating a referentially transparent function gives the same value for same arguments.



In general it means that the function **don’t have side effect** which may influence its result.

Referential transparency

Which of the following methods are an example of Referential Transparency?

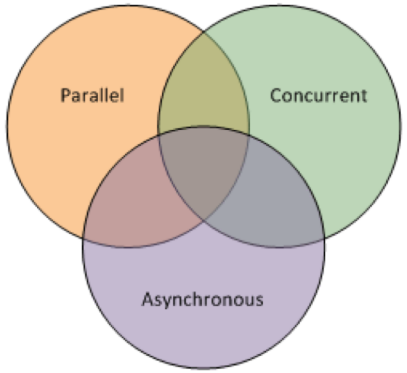
C#

```
DateTime Add(DateTime date, TimeSpen duration) =>
                        date.Add(duration);

DateTime AddDay(DateTime date) =>
                        Add(date, TimeSpan.FromDay(1));

DateTime Tomorrow() => AddDay (DateTime.Now)

DateTime AddTrigger(DateTime date, Func<int> f) =>
                        Add(date, TimeSpan.FromDay(f()));
```

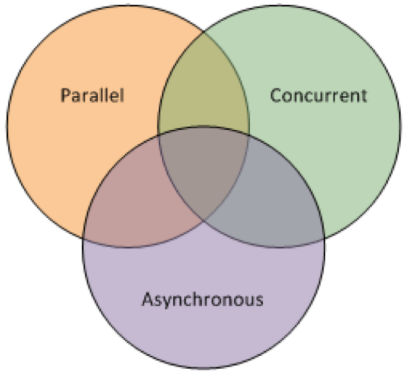
Parallel and Concurrent vocabulary

Definition

- Concurrent
- Multithreaded
- Parallel
- Asynchronous



Definition



Concurrent

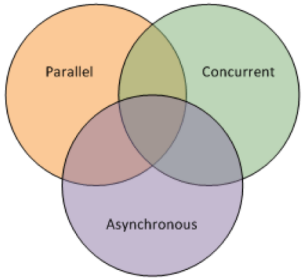
Multithreaded

Parallel

Asynchronous

Several things happening at once
TPL – Async – Agent - RX

Definition

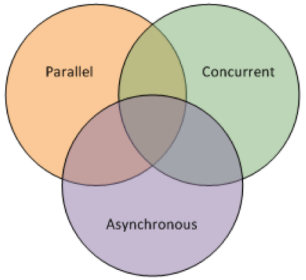


- Concurrent
- Multithreaded
- Parallel
- Asynchronous

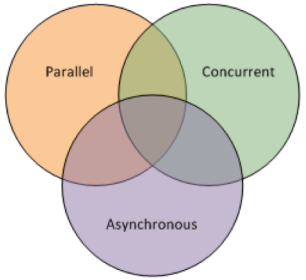
Several things happening at once
TPL – Async – Agent - RX

Multiple execution context

Definition



Concurrent	Several things happening at once TPL – Async – Agent - RX
Multithreaded	Multiple execution context
Parallel	Multiple simultaneous computations TPL – Async – Agent – RX - GPU
Asynchronous	



Definition

Concurrent	Several things happening at once TPL – Async – Agent - RX
Multithreaded	Multiple execution context
Parallel	Multiple simultaneous computations TPL – Async – Agent – RX - GPU
Asynchronous	Computation will complete later in the future Async Workflow / TAP

Concepts and Terminology
Flynn's Classical Taxonomy

Distinguishes multi-processor architecture by instruction and data

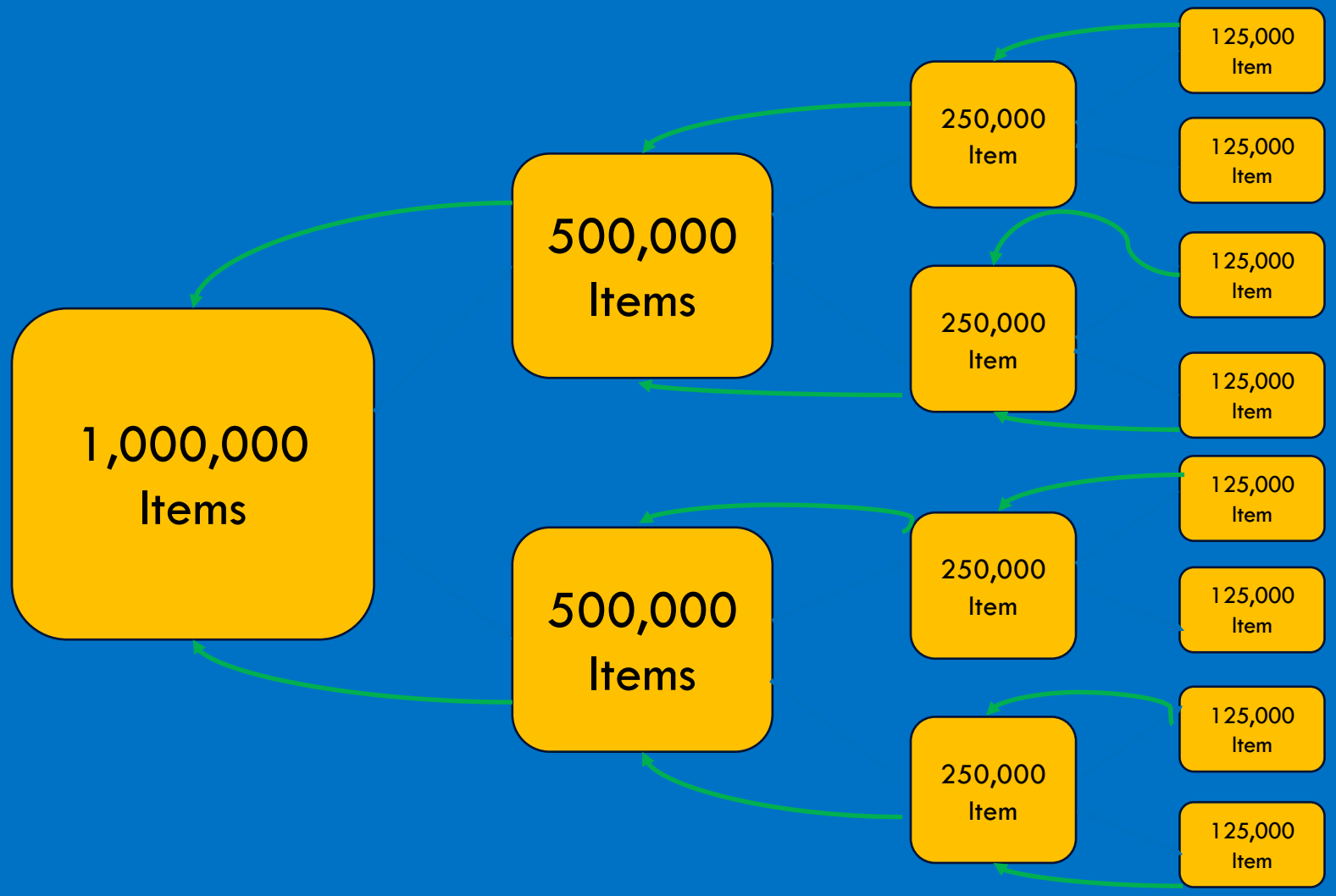
SISD – Single Instruction, Single Data

SIMD – Single Instruction, Multiple Data

MISD – Multiple Instruction, Single Data

MIMD – Multiple Instruction, Multiple Data

Strategies to parallelize the code



How do you develop applications for the multicore processors?

- Identify parallel parts
  - Micro-optimizing code that does not significantly contribute to overall inefficiency is largely a waste of time
- Apply correct design patterns
- Sophisticated synchronization features to avoid deadlocks/race etc…
- Thread safe data structures & containers
- Language/API support for common parallel programming patterns to achieve task/data parallelism
- Optimize the most inefficient portions of a program first to get the largest benefit (20%-80% rule)
  - Never have time to optimize everything, so start intelligently.

Strategies to parallelize the code

Make sure the target machine allows parallelism and ensure dynamic resources adaptation.

There are four main components to consider for designing and implementing a parallel application:

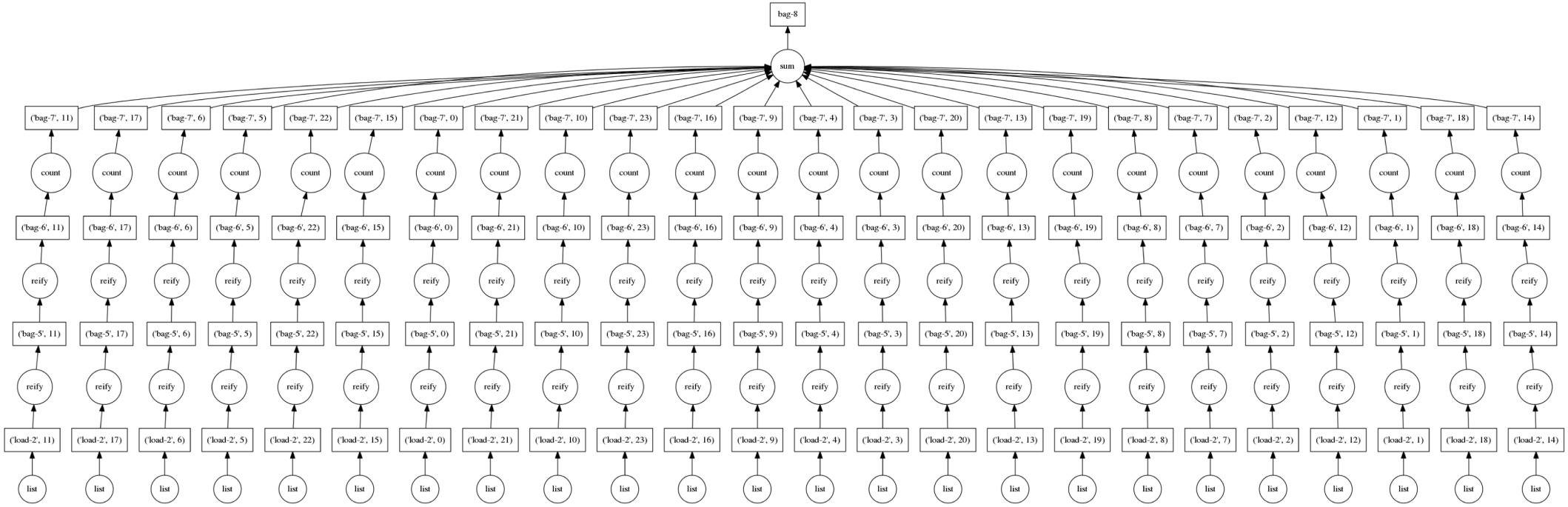
1. Measure the performance and set your goal
2. Partitioning the work into smaller tasks (units of computation)
3. Coordinating the tasks for running in parallel
4. Implement strategies for sharing the state required to execute the tasks.
    1. **Ideally, parallel computation should have multiple autonomous tasks that do not share any data**

Before start to parallelize the code, you must understand the complexities, and you should measure carefully to make sure that it is actually giving you the gains that you expect

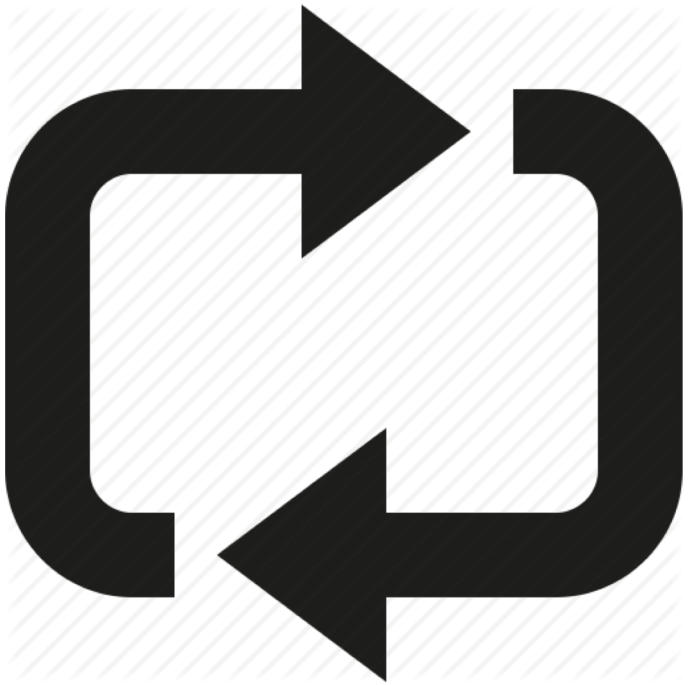
Strategies to parallelize the code

- Divide the program into multiple tasks that can run in parallel
- In case of CPU-bound operation, the number of parallel tasks should match the number of cores available
- Avoid share of mutable state
  - Avoid the share data among threads
  - force immutability in your coding style or isolate and encapsulate mutable objects at the lowest level possible
- Decomposing the work into discrete and independent units of computation, which do not share any data.
- Coordinate these units of computation to be parallelized
- Sharing the data to perform the computation
- **It is better to have more work items than processors**
- When thinking how to parallelize the application, examine the overall architecture of the application
- Analyze and select the correct pattern.
- Consider the usage of primitive locks as last possible solution
- **It's all about profiling your algorithm, and finding the right balance**
- Measure, Measure, Measure

Embarrassingly parallel algorithms



Granular vs Higher level



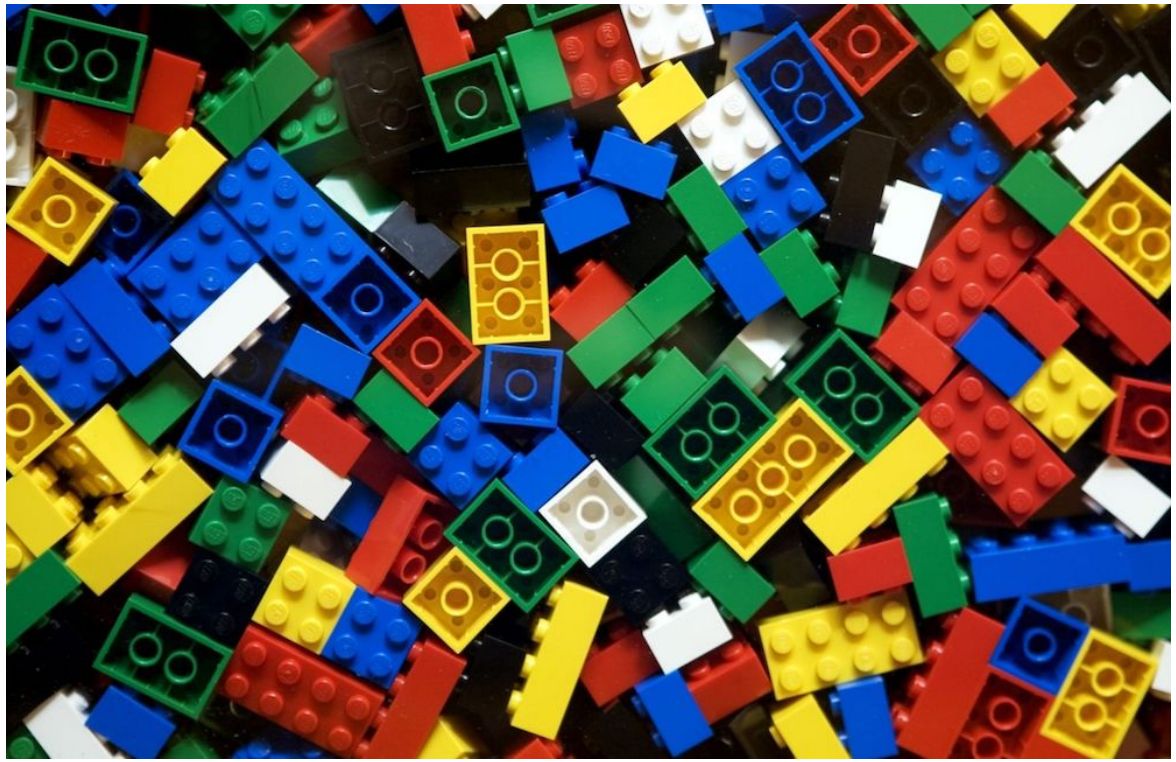
One common strategy to refactor existing code base for improving the execution time, is by determinate one sequential code area that run slow and introduce parallelism in the code. Then repeat the process to the next bottleneck. This approach can provide initial performance benefits and improvements, but it is not the recommended action.

A better solution is starting the optimization at higher level, by understanding the real application problem and by applying parallelism strategies across the entire code atomically as a unit. Then, move to a more granular level of optimization.

Strategies to parallelize the code

The first step in designing any parallelized system is **Decomposition**.

Decomposition is nothing more than taking a problem space and breaking it into discrete parts. When we want to work in parallel, we need to have at least two separate things that we are trying to run. We do this by taking our problem and decomposing it into parts.



Parallel and Concurrent Patterns

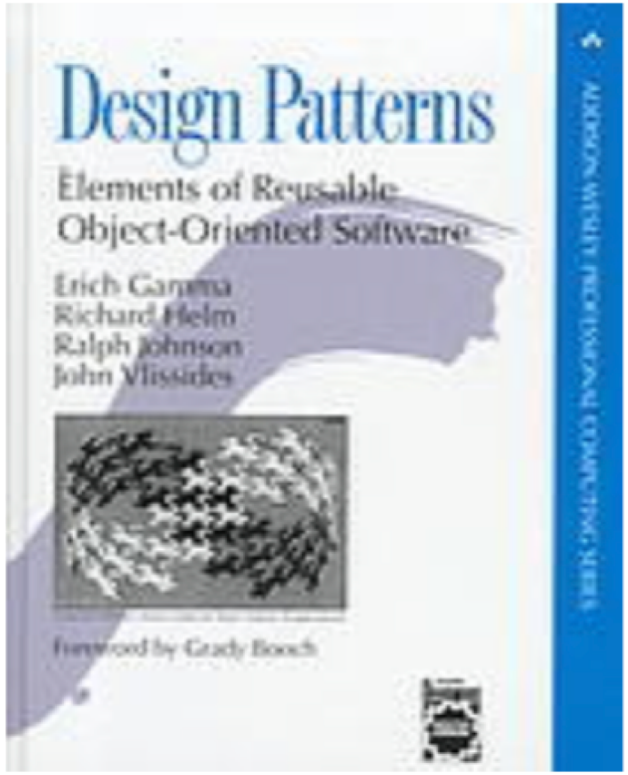
History of Patterns?

Patterns date back to late 1970s but...

Became really popular with the “Gang of Four” book – Thisbookdescribedobjectorientedpatterns
– Published in 1994

Famous patterns: (as defined by me) – Decorator

– Observer – Adaptor – Factory
– Singleton – Prototype – Iterator



Why patterns?

- Patterns help “structure” your code
- Sharing code becomes easier – “I’m using a singleton here...”
- Patterns have been “proven” and are understood so if you follow one you benefit from common knowledge
- Languages often “support” a particular pattern – Ex: decorator, map, reducer
- Optimization can be done without the programmers knowledge
  - Ex: parallelize a map call, copy of modify factory

Parallel Patterns

The following additional parallel patterns can be used for “structured parallel programming”

- Speculative selection
- Map
- Recurrence
- Scan
- Reduce
- Fork/join
- Map-Reduce
- Partition
- Master/Worker
- Pipeline
- Divide and Conquer
- Segmentation
- Stencil
- Search/match
- Gather
- Merge scatter
- Permutation scatter
- Atomic

Functional Programming & Concurrency

- What is the purpose of synchronization?
  - To avoid conflicting updates of shared data
- Shared mutable data are hard to understand and use correctly
  - inefficient on multicore machines
- Functional programming
  - No updates to shared data

Instead: copying, partial sharing, intermediate data structures, message passing, ...

Agenda

- History and influence of Functional Programming
- What is Functional Programming
- Why Functional Programming

Twitter Paper

Your Server as a Function

Marius Eriksen

Twitter Inc.

marius@twitter.com

Abstract

Building server software in a large-scale setting, where systems exhibit a high degree of concurrency and environmental variability, is a challenging task to even the most experienced programmer. Efficiency, safety, and robustness are paramount—goals which have traditionally conflicted with modularity, reusability, and flexibility.

We describe three abstractions which combine to present a pow-

Services Systems boundaries are represented by asynchronous functions called *services*. They provide a symmetric and uniform API: the same abstraction represents both clients and servers.

Filters Application-agnostic concerns (e.g. timeouts, retries, authentication) are encapsulated by *filters* which compose to build services from multiple independent modules.

http://monkey.org/~marius/funsrv.pdf

Twitter Paper

Your Server as a Function

Servers in a large-scale setting are required to process tens of thousands, if not hundreds of thousands of requests concurrently; they need to handle partial failures, adapt to changes in network conditions, and be tolerant of operator errors. As if that weren't enough, harnessing off-the-shelf software requires interfacing with a heterogeneous set of components, each designed for a different purpose. These goals are often at odds with creating modular and reusable software [6].

We present three abstractions around which we structure our server software at Twitter. They adhere to the style of functional programming—emphasizing immutability, the composition of first-class functions, and the isolation of side effects—and combine to present a large gain in flexibility, simplicity, ease of reasoning, and robustness.

Abstract

Building server software
hibit a high degree of con
a challenging task to eve
ficiency, safety, and robu
traditionally conflicted w
We describe three abst

presented by asynchronous
vide a symmetric and uni-
represents both clients and

(e.g. timeouts, retries, au-
ters which compose to build
modules.

History

λ

1930

1958 LISP

1962 APL

1966 ISWIM

1973 ML

1977 FP

1986 Erlang

1988 Haskell

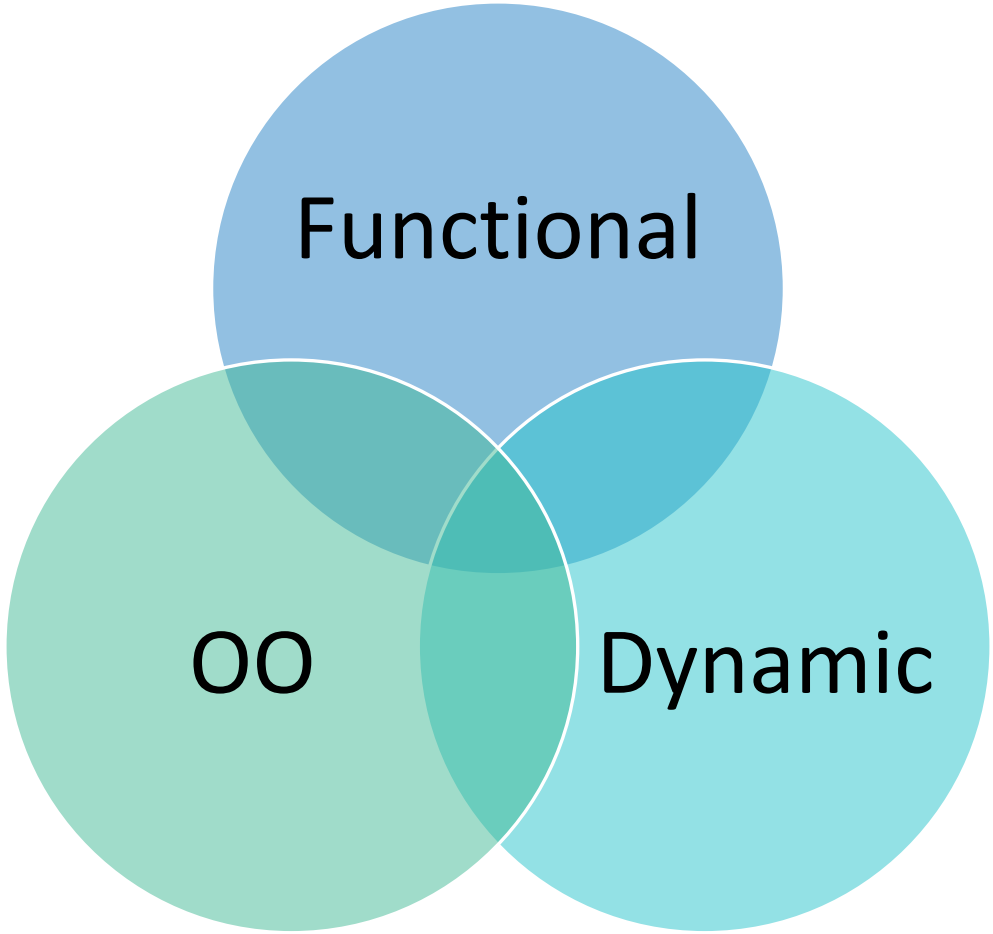
1996 Ocaml

2003 Scala

2005 F#

2007 Clojure

Languages go multi-paradigm



Early 1994 support for lambda, filter, map and reduce in Python

Introduced List Comprehension in C# 3.0 with LINQ and Java 8 with Streams

Introduced full support for lambda in Java 8, C#, VB.NET and C++

What is "Functional Programming?"

Functional programming is just a style, is a programming paradigm that treats computation as the evaluation of functions and avoids state and mutable data…

"Functions as primary building blocks" (first-class functions)

programming with *"immutable"* variables and assignments, no *state*

- Programs work by returning values instead of modifying data

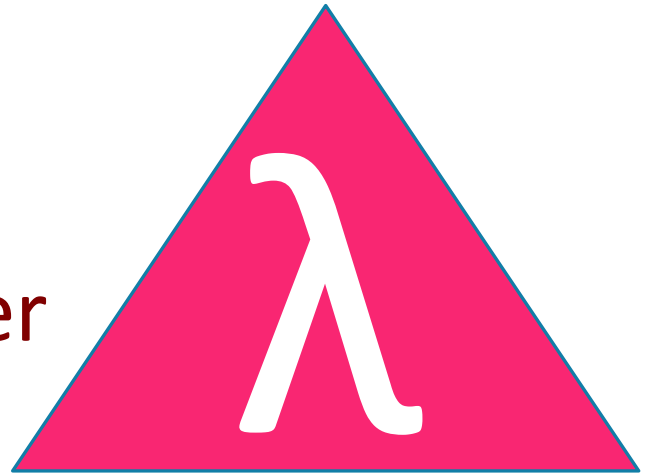
What is "Functional Programming?"

Functional programming is so called because a program consists entirely of functions.

— John Hughes, Why Functional Programming Matters

Declarative

First class
Higher-Order
Functions



Immutability

… is all about functions! Identifying an abstraction and building a function, use existing functions to build more complex abstractions, pass existing functions to other functions to build even more complex abstractions

Functional Programming

It's declarative, stateless, side-effects free and immutable

describes what we want, not how we want it

"Functional programming allows developers to describe what they want to do, rather than forcing them to describe how they want to do it."

Anders Hejlsberg, C# creator

Functional Programming

It's declarative, stateless, side-effects free and immutable

describes what we want, not how we want it

system relies only on inputs, not external state

Functional Programming

It's declarative, stateless, side-effects free and immutable

describes what we want, not how we want it

system relies only on inputs, not external state

once value has been set, we can't override its value

Return new value, instead of altering existing ones

… but why?

… why FP?

Motivation for Function Programming

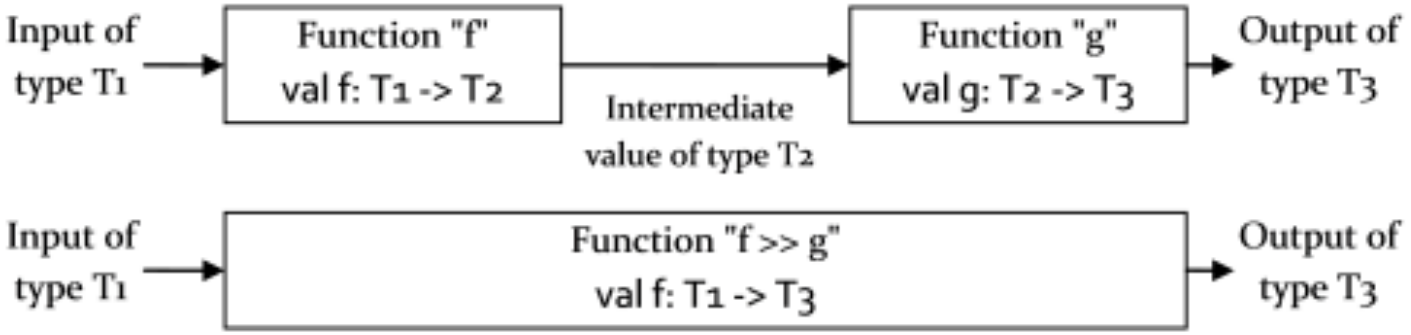
- Functional Programming promotes Composition and Modularity
- Declarative programming style
- Concurrency, FP is easy to parallelize
- Conciseness, less code (no null checking)
- Simplicity in life is good: cheaper, easier, faster, better
  - We typically achieve simplicity in software in two ways:
    - By raising the level of abstraction
    - Increasing modularity
- Better composition and modularity == reuse
- The Multi-core (r)evolution! Concurrency without pain

Functional Composition

Function Composition

Function Composition - Building with composition. Composition is the 'glue' that allows us build larger systems from smaller ones. This is the very heart of the functional style. Almost every line of code is a composable expression. Composition is used to build basic functions, and then functions that use those functions, and so on.

Composition — in the form of passed parameters plus first-class functions



Function Composition Operators

The forward pipe operator:

```
let (|>) arg func =          Type: 'a -> ('a -> 'b) -> 'b
     func arg
```

Allows you to reorder function calls to put the last argument first:

```
let three = 1 |> add 2
```

Produces function
`int -> int`

Invokes `(int -> int)`
with 1

Function Composition

```
open System

let evenNumbersAsWords =
    [ 1 .. 15 ]
    |> Seq.filter (fun i -> i % 2 = 0)
    |> Seq.map (fun i -> i.ToWords())
    |> Seq.reduce (sprintf "%s, %s")
```

Creates an F# List with the integers 1 to 15

Partial application of the Seq.filter

F# Lambda

Partial application of sprintf

Map Reduce!

Immutability!

evenNumbersAsWords:
"two, four, six, eight, ten, twelve, fourteen"

Currying and Partial Application

C#

```
public static Func<int, int> Add(int x)
{
    return y => x + y;
}

public static Func<int, int> Add1() => Add(1)

int valueA = Add  (1)(3); // Add  (1:x) (3:y)
int valueB = Add1 (3);    // Add1 (3:y)
```

Currying and Partial Application

F#

```
let add x y = x + y
let add1 = add 1
let value = add1 2
```

Avoid Side Effects

FP Preachings!

Avoid Side-Effects!

- Do not modify variables passed to them
- Do not modify any global variable

Avoid Mutation

Reasoning about your code

```
int Sum(List<int> values) { … }

List<int> values = new List<int>{ 1,2,3,4,5 };

int result1 = Sum(values); // 15

int result2 = Sum(values); // ??
```

Reasoning about your code

```
int Sum(List<int> values) {

    int sum = 0;
    for(int i = 0;i < arr.Length; i++){
        sum += values[i];
        values[i] = 0;

    }
}
```

Reasoning about your code

```
int Sum(List<int> values) { … }

List<int> values = new List<int>{ 1,2,3,4,5 };

int result1 = Sum(values); // 15

int result2 = Sum(values); // ??

int Sum(IEnumerable<int> values) { … }
```

Side Effects

Side effects aren't bad…

Unwanted Side Effects are Evil!

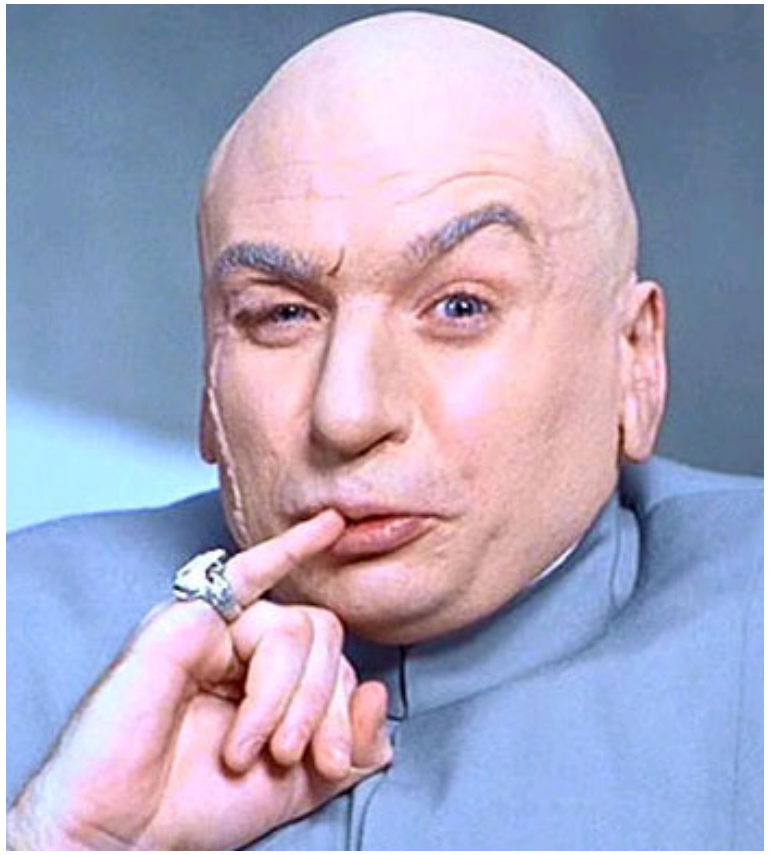
Side effects aren't bad…

unwanted side effects are EVIL

and the root of many bugs

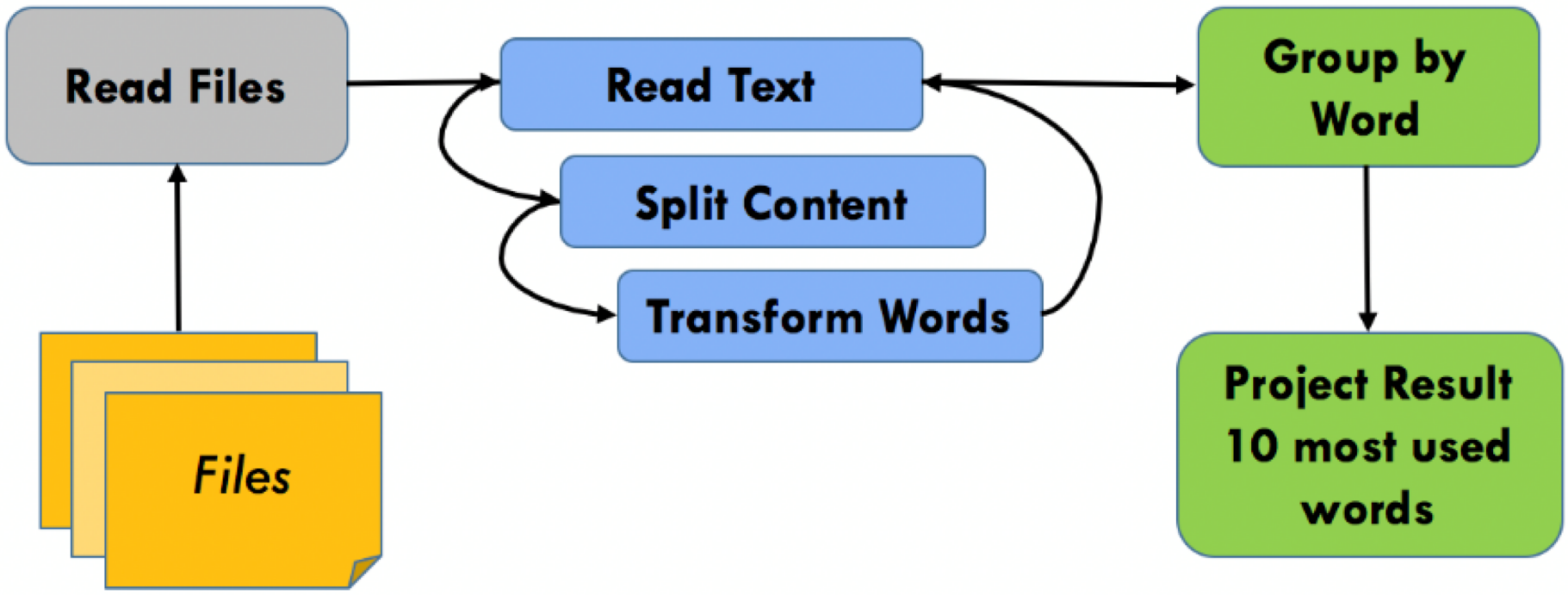
One of the biggest advantages with functional programming is that the order of execution of "side effect free functions" is not important.

… "side effect free functions" are easy to parallelize



Avoiding side effects with pure functions

Parallel words counter with side effects



Parallel words counter with side effects

```
public static Dictionary<string, int> WordsCounter(string source)
{
    var wordsCount =
            (from filePath in
                Directory.GetFiles(source, "*.txt").AsParallel()
                from line in File.ReadLines(filePath)
                from word in line.Split(' ')
                select word.ToUpper())
        .GroupBy(w => w)
        .OrderByDescending(v => v.Count()).Take(10); //#C
    return wordsCount.ToDictionary(k => k.Key, v => v.Count());
}
```

Why removing without side effects

- Easy to reason about the correctness of your program
- Easy to compose functions for creating new behavior
- Easy to be isolated and, therefore, easy to test and less bug prone
- Easy to be executed in parallel; because pure functions don’t have external dependencies, their order of execution (evaluation) doesn’t matter

LAB :

Avoiding side effects with pure function

Parallel Words Counter with side effects

```
static Dictionary<string, int> PureWordsPartitioner
                              (IEnumerable<IEnumerable<string>> content) =>
    (from lines in content.AsParallel()
        from line in lines
        from word in line.Split(' ')
        select word.ToUpper())
               .GroupBy(w => w)
              .OrderByDescending(v => v.Count()).Take(10)
               .ToDictionary(k => k.Key, v => v.Count());

static Dictionary<string, int> WordsPartitioner(string source) {
    var contentFiles =
       from filePath in Directory.GetFiles(source, "*.txt")
             select File.ReadLines(filePath);

    return PureWordsPartitioner(contentFiles);
}
```

Functional Concepts introduction

Function definition

A function always gives the **same output** value for a **given input** value

A function has no **side effects**

C#

```
public static int add(int a, int b)
{
    return a + b;
}
```

F#

```
let add a b = a + b
```

Function definition

C#

```
public static T1 First<T1, T2>((T1, T2) t)
{
    return t.Item1;
}
```

F#

```
let first (x, y) = x
```

Function as first class

Functions like **any values**

```
Func<int, int, int> add = (a, b) => a + b;
```

C#

- Function can be **passed as parameter** to other function

```
IEnumerable<T> Where<T> (IEnumerable<T>, Func<T, bool>)
```

C#

- Function can be **return** from other function

```
Func<int, int, int> GetCalcStrategy()
```

C#

Function as first class

C#

```
public static IEnumerable<T2> Select<T1, T2>(
    this IEnumerable<T1> xs,
    Func<T1, T2> f)
{
    foreach(var x in xs)
    {
        yield return f(x);
    }
}

var newList = list.Select(x => x + 1);
```

Function as first class

F#

```
let map (f : 'a -> 'b) (xs : 'a seq) =
    seq { for x in xs do
              yield f x }

let newList = [1..10]
              |> map (fun x -> x + 1)
```

Function as first class

C#

```
public static IEnumerable<T2> Select<T1, T2>(
    this IEnumerable<T1> xs,
    Func<T1, T2> f)
```

F#

```
let map (f : 'a -> 'b) (xs : 'a seq)
```

Basic F# syntax

Curly braces are not used to delimit blocks of code. Instead, **indentation** is used

Whitespace is used to separate parameters rather than commas.

Currying and Partial Application

Currying is the process of rewriting function with multiple parameters to a series of functions, each with only one parameter.

Currying is done **automatically** by the compiler.

Partial application is when function applied with some of the parameters and return new function

Currying and Partial Application

C#

```
public static Func<int, int> Add(int x)
{
    return y => x + y;
}

public static Func<int, int> Add1() => Add(1)

int valueA = Add  (1)(3); // Add  (1:x) (3:y)
int valueB = Add1 (3);    // Add1 (3:y)
```

Currying and Partial Application

F#

```
let add x y = x + y
let add1 = add 1
let value = add1 2
```

Values

Assign **name** to **value**

Not like C# **variables**

Usually **values** is **separated** from **behavior**

```
let num = 1
let str = "Hii"
let person = Person()
```

F#

Recursion

Used a lot in functional programming

Used instead of loop

Simplify traversing complex data structure

Tail Call Optimization

F#

```
let rec factorial n =
    if n = 0 then
        1
    else n * factorial (n - 1)
```

Recursion

F#

```
let tailRecFactorial n =
    let rec fact n acc =
        if n = 0 then
            acc
        else
            fact (n - 1) acc * n
    fact n 1
```

The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.

-- Edsger Dijkstra