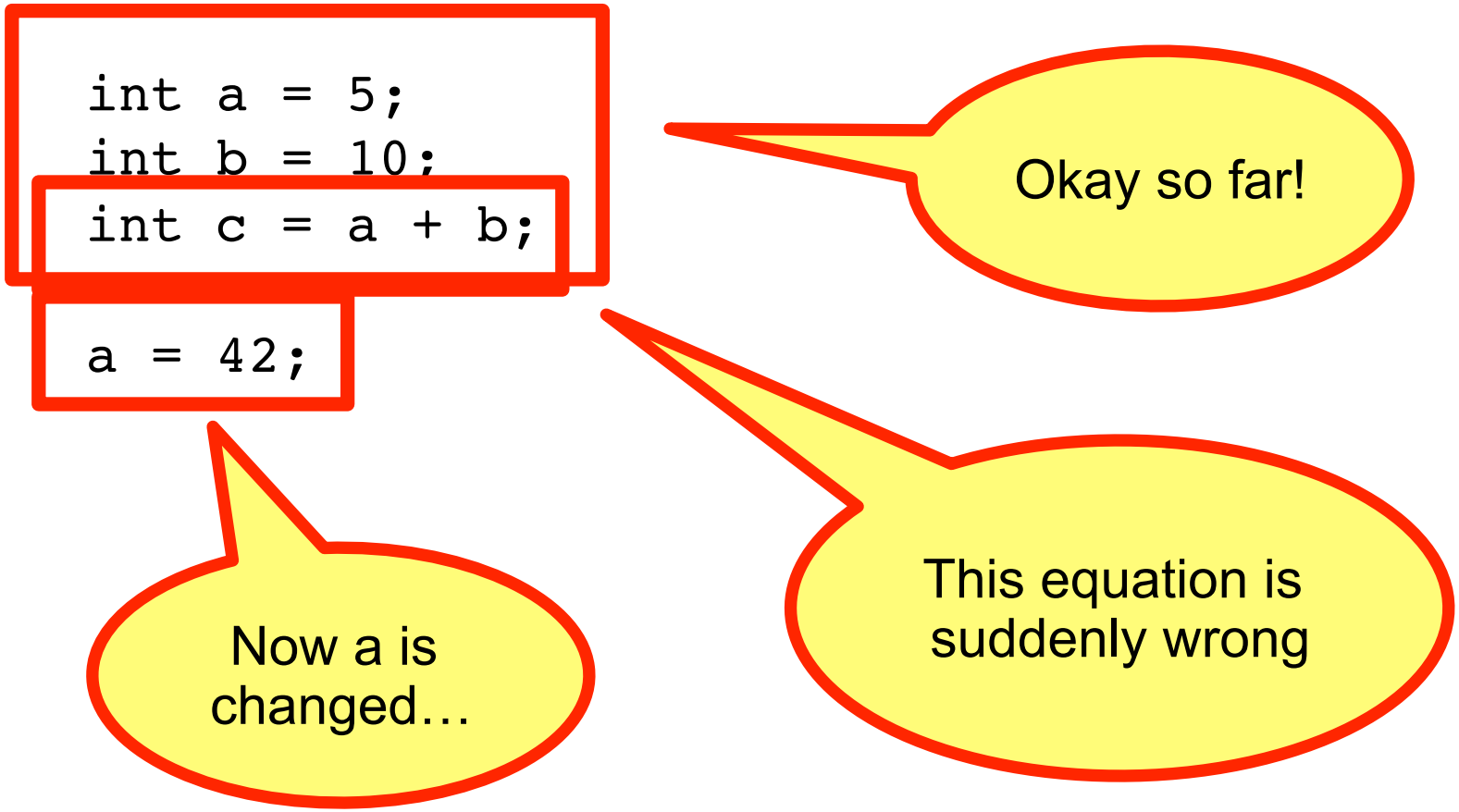
# The advantage of Immutable Data

Oliver Sturm
olivers@devexpress.com

# Agenda

- Basics for and against mutability
- Designing simple and complex data structures with immutability in mind
- Immutability within functions

# (Wrong?) Assumptions



# Change is good…

- That's a paradigm of imperative programming
- State change is regarded as a central idea of computer programming
- For some tasks a state-driven algorithm can be the best solution, especially in imperative languages

# Change is bad…

- Change creates problems
- Think about debugging:
  - "Why is a suddenly 5?"
- Change is an enemy of scalability
  - Issues with shared data
  - Locking & Co. are often used as workarounds

# Idea: stop changing things

- Great idea. But:
- Change is subjective
- Scalability requires change
- Result: Programmers must have a choice of mutability or immutability, with clear distinctions and boundaries

# Simple steps

- Regard variables as immutable
- Declare fields as readonly
- Instead of changing objects, create new instances

# A mutable class

```
class Person {
  private string name;
  public string Name {
    get { return name; }
    set { name = value; }
  }
}
```

# An immutable class

```
class ImmutablePerson {
  private readonly string name;
  public string Name { get { return name; } }

  public ImmutablePerson(string name) {
    this.name = name;
  }

  public ImmutablePerson ChangeName(
    string newName) {
    return new ImmutablePerson(newName);
  }
}
```

# Structs?

- Structs should be implemented immutably
- Problem: structs are value types
  - Special issues with mutable structs
  - Usually inefficient when creating many clones

# Demo

Simple types

# Isolation/Visibility

- Scenarios in concurrency
  1. Threads work for themselves
  2. Each thread works on a part of a larger task
  3. Threads must communicate
- 2 and 3 are often exchangeable on the algorithm level
- Immutable data types guarantee stability of data for a particular “context”

# Cloning objects is a challenge

- Using helper functions for object changes
- But what if there are many fields?
- … and several different changes?
- Creating a copy of an object requires some thought – shallow, deep etc.
- Creating a copy while making changes is harder

# Demo

Cloning objects

# Cloning objects - thoughts

- F# call syntax – very nice
- C# call syntax – not so nice
  - Type inference isn't good enough
  - No built-in syntax for container types
  - Idea: using lambda expressions to specify the change operations – doesn't work because fields are read-only

# Cloning objects – more thoughts

- Assumptions made about implementation of cloneable types
  - Only public read-only fields being used – rather uncommon, and problematic with things like data binding
  - Constructor takes all the fields as parameters – unavoidable when using *readonly*
  - Constructor parameter names are equal to field names
- The F# mechanism works very similarly, but automatic

# Cloning objects – even more thoughts

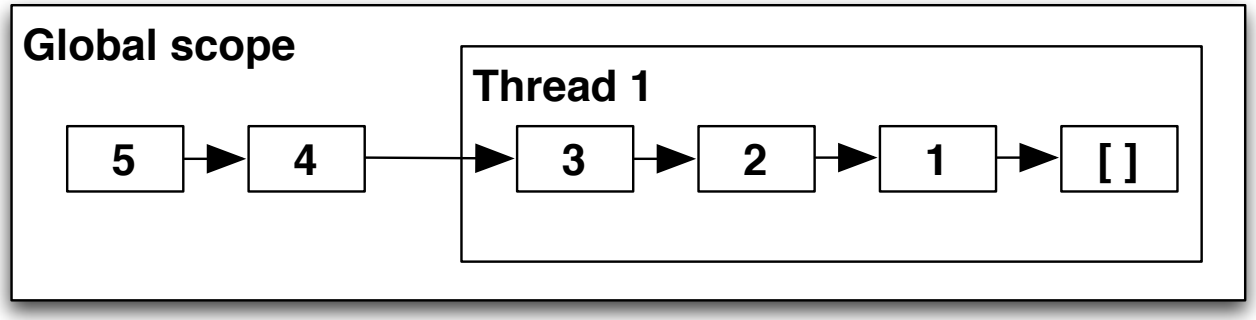
- The mechanism currently takes about 50 times as long as direct construction
- Optimizations:
  - Performance, similar to the accessor caching already used
  - Allow for use of properties, different names, etc, through intelligent lookup, attributes, etc
  - “dynamic”? More usable perhaps, but probably not faster

# And now…

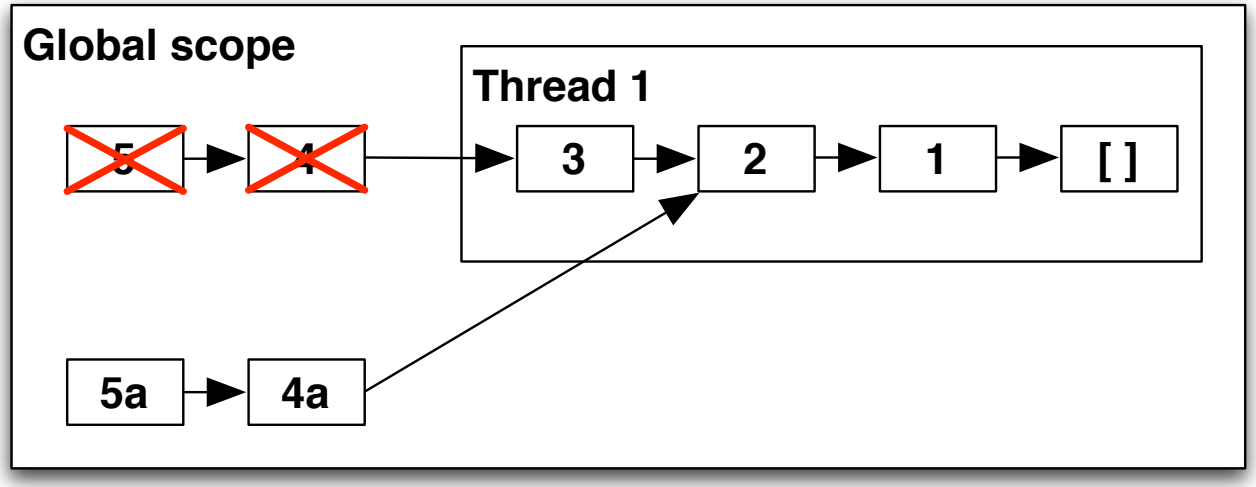
- What if there's more than one object?

# List types

**Adding elements**



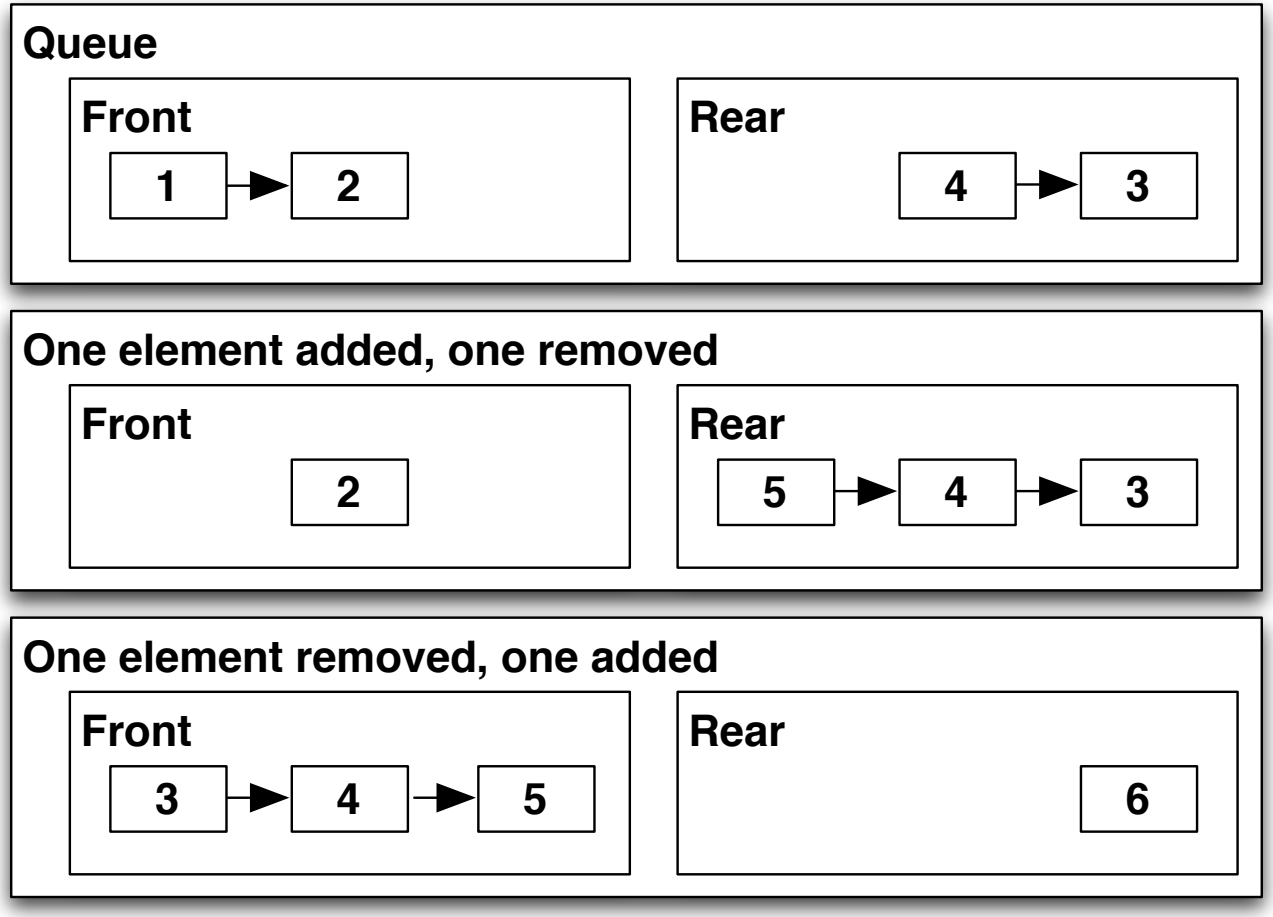

**Removing element 3**



# Demo

List types

# (FIFO) Queue



# Microsoft Immutable Collections

- Nuget package: Microsoft.Bcl.Immutable
- Implementation along the same lines as my own FCSlib
- Set of immutable data structures available
- .NET 4.5 required
- Semantics a bit odd^H^H^Hdifferent

# Immutability within functions

- Reusing variables – not a good idea
- Iterations in C# usually need mutability
- Alternatives:
  - Recursion – restricted in C#
  - Standard higher order functions move the problem away from ***your*** code
  - Clean structure

# Demo

Immutability within functions

# Some things are hard to demo

- Immutable data is an important step when trying to avoid side effects
- General software stability increases together with the simplicity of testing and debugging
- Try it yourself, you'll like it!

# Summary

- For scalability, it is important to coordinate mutable and immutable data cleanly
- Even complex data types can be designed as immutable
- C# can do all this, though some help from the compiler would be nice

Algorithms for List and Queue are ported from Chris Okasaki's book "Purely functional data structures"

# Thank you


Please feel free to contact me about the content anytime.

olivers@devexpress.com