Performance of Functional C# Code

Bart J.F. De Smet
bartde@outlook.com

pluralsight
hardcore dev and IT training

Anonymous Methods

- C# 2.0 shorthand syntax for delegates

```
delegate int BinOp(int a, int b);

class Program
{
  static void Main()
  {
    BinOp f = delegate (int x, int y)
    {
      return x + y;
    };

    Console.WriteLine(f(2, 3));
  }
}
```

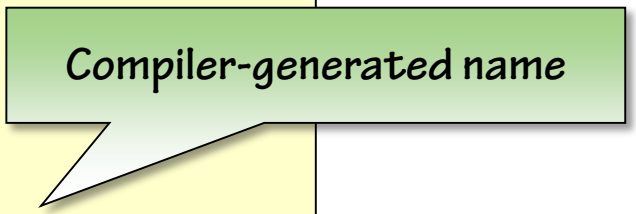
Optional argument list

Statement body

Anonymous Methods

- Compiler-generated method

```
delegate int BinOp(int a, int b);

class Program
{
  static void Main()
  {
    BinOp f = new BinOp(<Main>b__0);

    Console.WriteLine(f(2, 3));
  }

  static int <Main>b__0(int x, int y)
  {
    return x + y;
  }
}
```



Anonymous Methods

- Caching of the delegate

Compiler-generated name

```
class Program
{
  static volatile BinOp CS$<>9__CachedAnonymousMethodDelegate1;

  static void Main()
  {
    if (CS$<>9__CachedAnonymousMethodDelegate1 == null)
      CS$<>9__CachedAnonymousMethodDelegate1 = new BinOp(<Main>b__0);

    BinOp f = CS$<>9__CachedAnonymousMethodDelegate1;

    Console.WriteLine(f(2, 3));
  }
```

Intermezzo – Compiler-generated Names

- **Naming for compiler-generated IL artifacts**
  - Variables, fields, methods, types, etc.
  - Unspeakable names:
    - <> makes it a non-valid C# identifier
    - CS$ known by FxCop not to raise unfixable warnings
    - One character to identify kind of generated name

0	(not used)	8	closure instance	g	initializer local	o	dynamic container
1	iterator state	9	cached delegate	h	transparent ident	p	dynamic call site
2	iterator current	a	iterator instance	i	anon type field	q	dynamic delegate
3	?	b	anon method	j	anon type param	r	(deprecated)
4	hoisted this	c	display class type	k	auto prop field	s	lock taken flag
5	hoisted local	d	iterator class type	l	iterator thread	t	async local
6	?	e	fixed buffer	m	iterator finally	u	async awaiter
7	?	f	anonymous type	n	fabricated methods	v	…

Lambda Expressions

- **Introduced in C# 3.0**
  - Motivated by the LINQ feature set
  - Concise notation for functions with **fat arrow syntax**

```
BinOp f1 = (x, y) => x + y;
BinOp f2 = (int x, int y) => x + y;
BinOp f3 = (int x, int y) => {
  return x + y;
};
```

Parameter type inference

Optional statement body

- **Two representations**
  - Homo-iconic property
    - Homo = same
    - Iconic = representation
  - Assignable to either:
    - Delegate types (shorthand for anonymous methods)
    - Expression tree types (code as data representation)

Closures and Display Classes

- **Capturing local variable from outer scope**
  - Can outlive the stack frame
  - Possible semantics:
    - "By value" – creates a (shallow) copy
    - "By reference" – sharing, read/write access, etc.

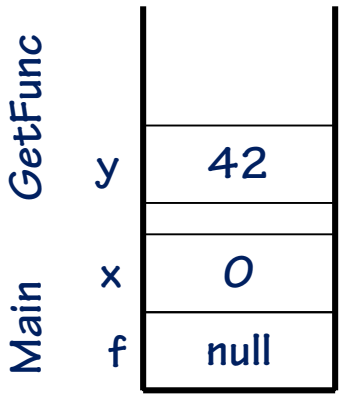
```
int x = 42;

Action a = delegate
{
  Console.WriteLine(x);
};

x = 43;

a(); // Prints 42 or 43?
```

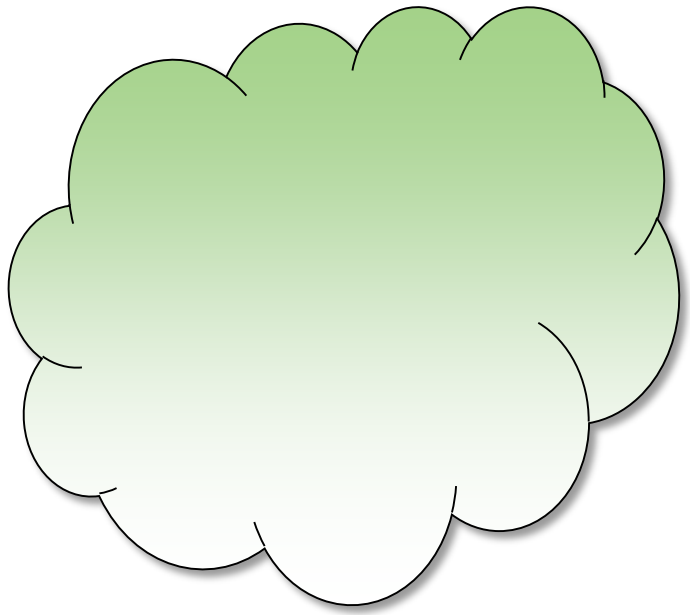
```
// A simple "cell" to hold an object
Tuple<Func<T>, Action<T>> NewProperty<T>()
{
  T x = default(T);

  return Tuple.Create(
    new Func<T>(() => x),
    new Action<T>(value => x = value);
  );
}
```

Closures and Display Classes

- **Lifetime management?**
  - Stack-allocated local variable
  - Heap-allocated delegate

```
static Func<int> GetFunc()
{
  int y = 42;
  return () => y;
}

static void Main()
{
  Func<int> f = GetFunc();
  int x = f();
  Console.WriteLine(x);
}
```



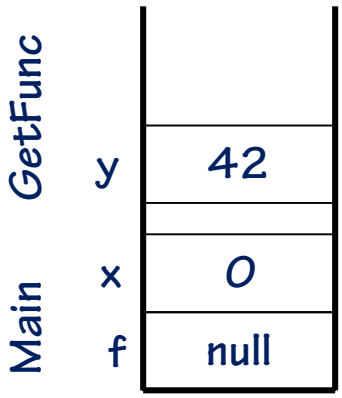


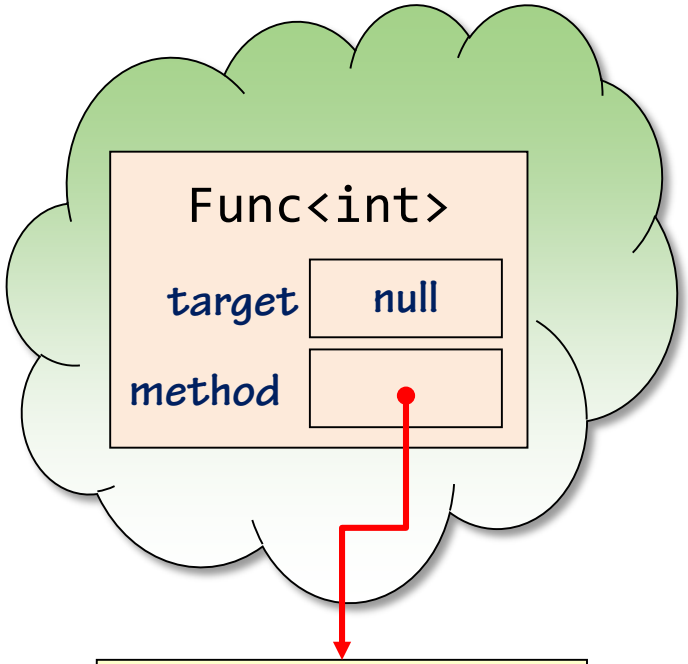
```
int <GetFunc>b__0()
{
   return y;
}
```

Closures and Display Classes

- **Lifetime management?**
  - Stack-allocated local variable
  - Heap-allocated delegate

```
static Func<int> GetFunc()
{
  int y = 42;
  return () => y;
}

static void Main()
{
  Func<int> f = GetFunc();
  int x = f();
  Console.WriteLine(x);
}
```



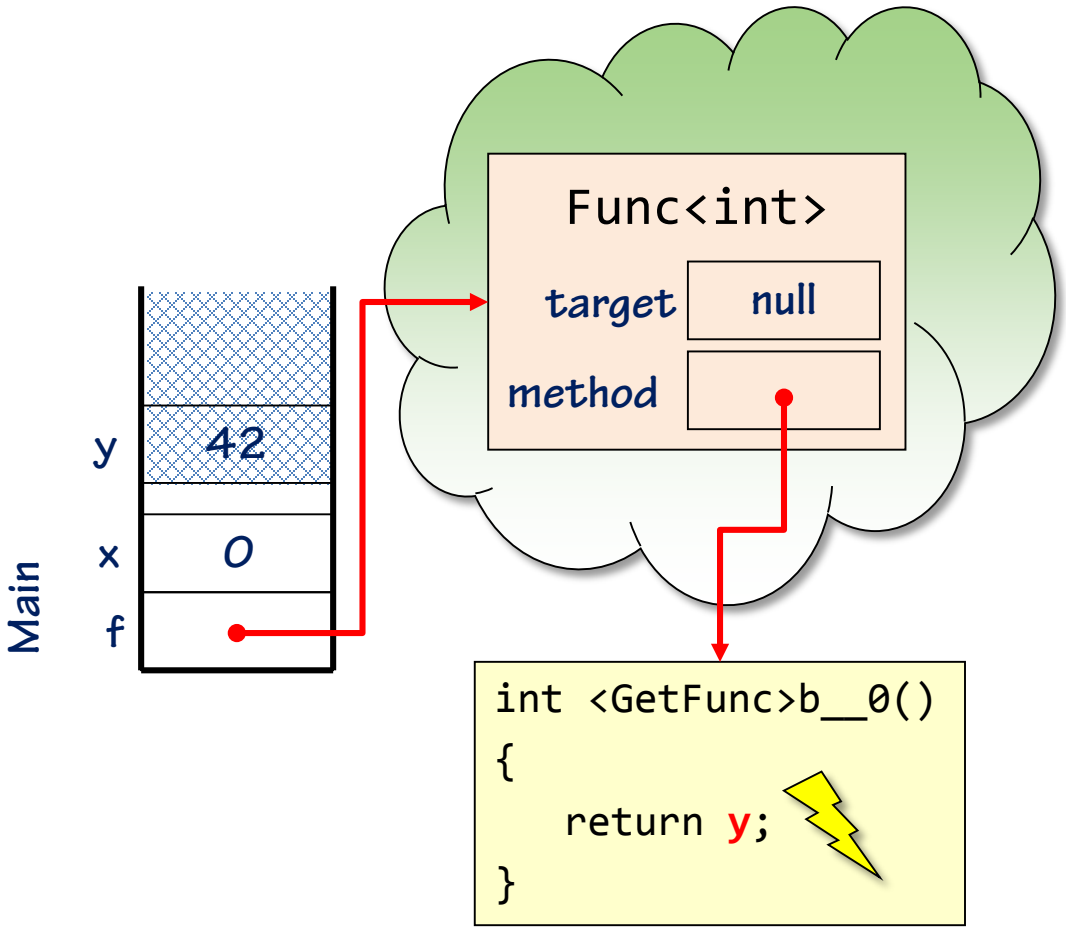



```
int <GetFunc>b__0()
{
   return y;
}
```

Closures and Display Classes

- **Lifetime management?**
  - Stack-allocated local variable
  - Heap-allocated delegate

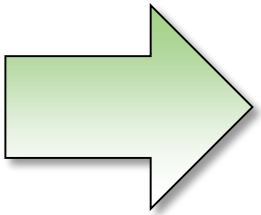
```
static Func<int> GetFunc()
{
  int y = 42;
  return () => y;
}

static void Main()
{
  Func<int> f = GetFunc();
  int x = f();
  Console.WriteLine(x);
}
```



Closures and Display Classes

- **Closures**
  - Concept from functional programming (Peter Landin, 1964)
    - Based on lambda calculus (Alonzo Church, 30s)
  - Function with its "referencing environment"
  - Display classes in C#

```
static Func<int> GetFunc()
{
  int y = 42;
  return () => y;
}
```



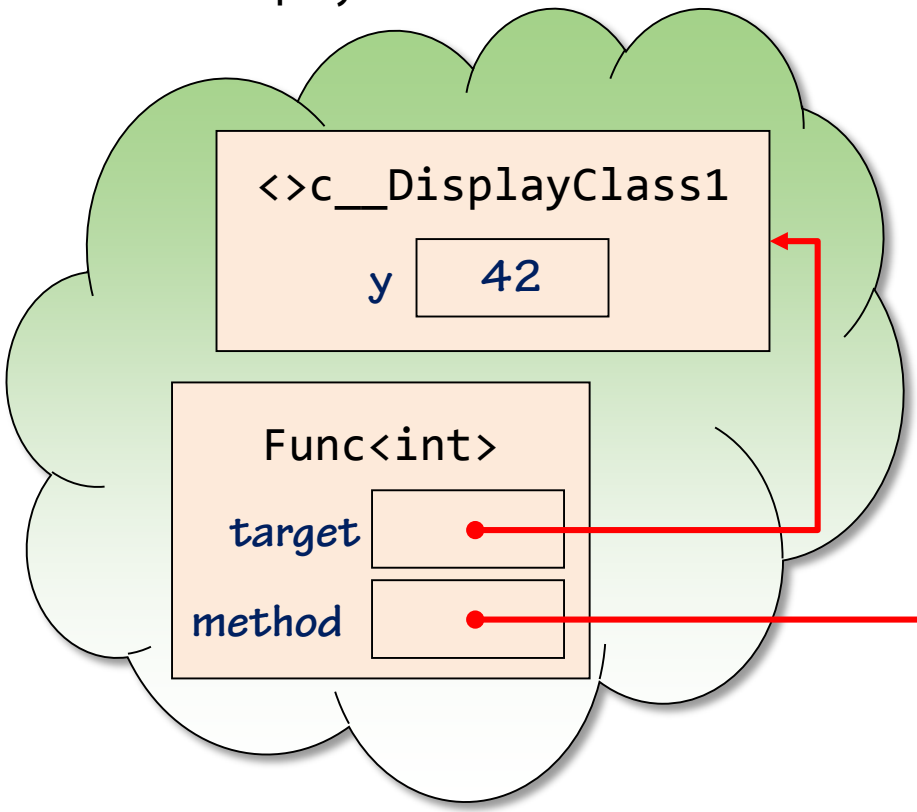
```
class <>c__DisplayClass1
{
  public int y;

  public int <GetFunc>b__0()
  {
    return this.y;
  }
}
```

Hoisted local

```
static Func<int> GetFunc()
{
  var d = new <>c__DisplayClass1();
  d.y = 42;
  return d.<GetFunc>b__0;
}
```

Closures and Display Classes

- **Closures**
  - Concept from functional programming (Peter Landin, 1964)
    - Based on lambda calculus (Alonzo Church, 30s)
  - Function with its "referencing environment"
  - Display classes in C#




```
class <>c__DisplayClass1
{
  public int y;

  public int <GetFunc>b__0()
  {
    return this.y;
  }
}
```

Closures and Display Classes

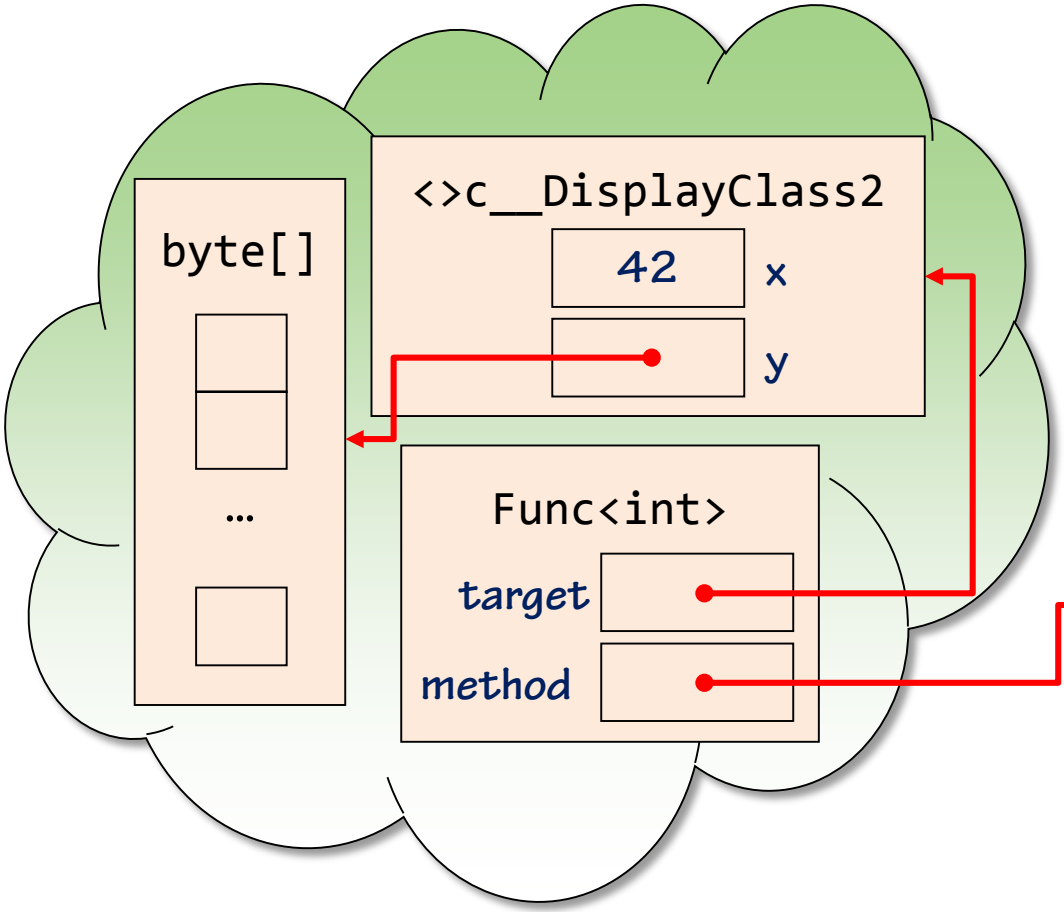
- **Space leaks**
  - Locals hoisted to same display class
  - Delegate keeping display class instance alive

```
static Func<int> GetFunc()
{
  int x = 42;
  byte[] y = new byte[64 * 1024 * 1024];
  Func<int> getSize = () => y.Length;
  // getSize can be collected
  return () => x;
}
```

```
class <>c__DisplayClass2 {
  public int x;
  public byte[] y;

  public int <GetFunc>b__0() {
    return this.y.Length;
  }

  public int <GetFunc>b__1() {
    return this.x;
  }
}
```

Closures and Display Classes

- **Space leaks**
  - Locals hoisted to same display class
  - Delegate keeping display class instance alive




```
class <>c__DisplayClass2 {
  public int x;
  public byte[] y;

  public int <GetFunc>b__0() {
    return this.y.Length;
  }

  public int <GetFunc>b__1() {
    return this.x;
  }
}
```

Closures and Display Classes

- **Scope of foreach loop variables**
  - What's the scope of x?

```
static void Main()
{
  foreach (var x in new[] { 1, 2, 3, 4 })
  {
    Task.Run(() => Console.WriteLine(x));
  }
}
```

Multi-threaded access to x

  - Possible output:
    - Unordered set { 1, 2, 3, 4 } if x is local to the loop
    - Missing values if x is outside the loop
  - Specification of foreach loop implementation matters

Closures and Display Classes

- **Scope of foreach loop variables**
  - Prior to C# 5.0

```
static void Main()
{
  using (var e = new[] { 1, 2, 3, 4 }.GetEnumerator())
  {
    int x;
    while (e.MoveNext())
    {
      x = e.Current;
      Task.Run(() => Console.WriteLine(x));
    }
  }
}
```

Foreach loop pattern

Race (write)

Race (read)

Closures and Display Classes

- **Scope of foreach loop variables**
  - Prior to C# 5.0
  - Manual fix by creating a local "copy"

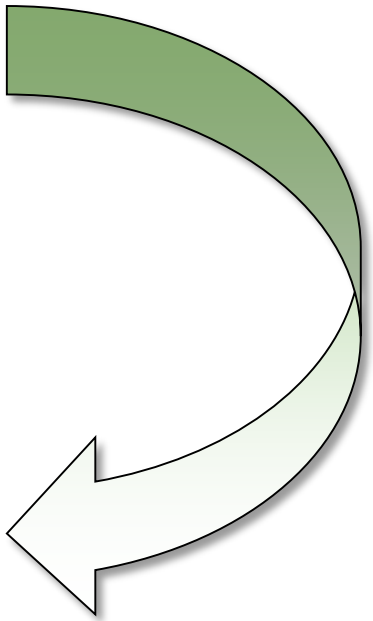
```
static void Main()
{
  foreach (var x in new[] { 1, 2, 3, 4 })
  {
    var y = x;
    Task.Run(() => Console.WriteLine(y));
  }
}
```

One closure instance per iteration

Closures and Display Classes

- **Scope of foreach loop variables**
  - Breaking change in C# 5.0

```
static void Main()
{
  foreach (var x in new[] { 1, 2, 3, 4 })
    Task.Run(() => Console.WriteLine(x));
}
```



C# 5.0

```
static void Main()
{
  foreach (var x in new[] { 1, 2, 3, 4 })
  {
    var y = x;
    Task.Run(() => Console.WriteLine(y));
  }
}
```

Summary

- **Anonymous methods and lambda expressions**
  - Concise notation for functions
  - Shorthand for method declaration and delegate instantiation
    - or expression trees (lambdas only)
  - Delegate instance caching

- **Closures and display classes**
  - Capturing local variable from outer scope
  - Hoisting to heap allocated display class
  - Potential space leaks
  - Scoping in loops
  - Visible in expression trees

- **Tail calls**
  - Recursion instead of loops
  - No tail call support in C# compiler