Attaining Truly Functional Functions

Zoran Horvat
CEO AT CODING HELMET
@zoranh75 http://csharpmentor.com

Understanding Partial Function Application

$$\text{scale}(factor, x) \rightarrow factor * x$$

$$\text{double}(x) \rightarrow \text{scale}(2, x)$$

$$\text{scale}(3,4) \rightarrow 12$$

$$\text{apply}(\text{scale}, 3,4) \rightarrow 12$$

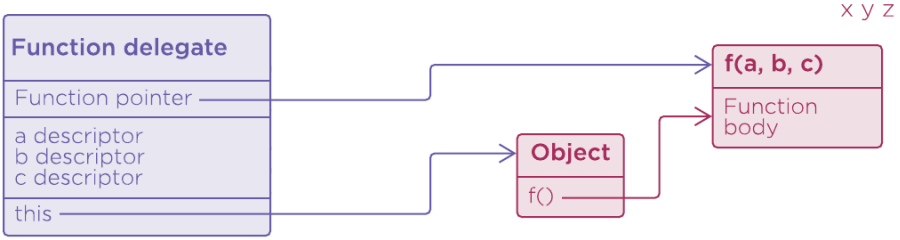
$$\text{apply}(\text{scale}, factor, x) \rightarrow factor * x$$

$$\text{apply}(\text{scale}, factor) \rightarrow f(x)$$

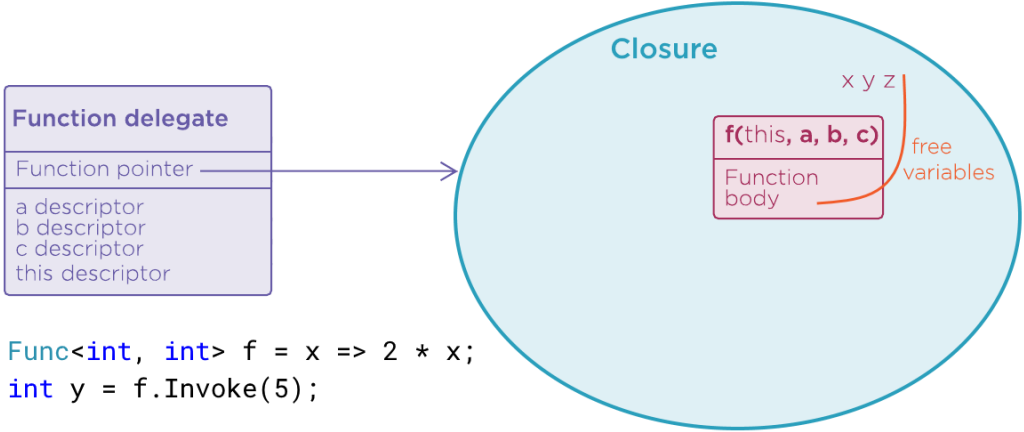
$$\text{apply}(\text{scale}, 2) \rightarrow \text{double}$$

$$\text{apply}(\text{apply}(\text{scale}, 2), 3) \rightarrow \text{apply}(\text{double}, 3) \rightarrow 6$$

Delegates and Closures

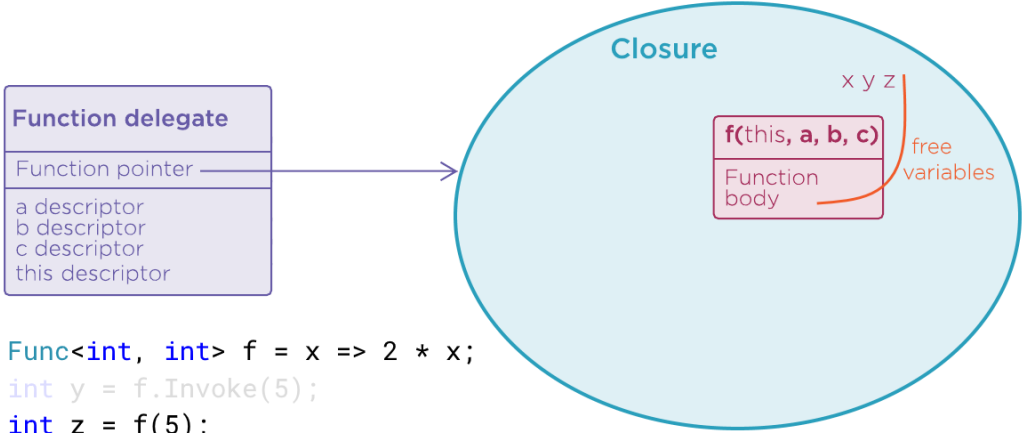


Delegates and Closures




```
Func<int, int> f = x => 2 * x;
int y = f.Invoke(5);
```

Delegates and Closures




```
Func<int, int> f = x => 2 * x;
int y = f.Invoke(5);
int z = f(5);
```

Lexical vs. Dynamic Scoping

```
namespace Demo                                  scope
{
  class Program                                 scope
  {
    void Main()                                 scope
    {
      int factor = 2;
      int scale(x) => factor * x;               scope
    }
  }
}
```

Lexical vs. Dynamic Scoping

```
namespace Demo
{
  class Program
  {
    void Main()
    {
      int factor = 2;
      int scale(x) => factor * x;
    }
  }
}
```

bind

Lexical vs. Dynamic Scoping

```
namespace Demo
{
  class Program
  {
    void Main()
    {
      int factor = 2;
      int scale(x) => factor * x;
    }
  }
}
```

bind

bind

Lexical vs. Dynamic Scoping

```
namespace Demo
{
  class Program
  {
    void Main()
    {
      int factor = 2;
      int scale(x) => factor * x;
    }
  }
}
```

bind

bind

As opposed to dynamic scoping:

```
void theCaller()
{
  int factor = 2;
  scale(3);
}
```

call

```
int scale(int x)
{
  return x * factor;
}
```

Lexical vs. Dynamic Scoping

```
namespace Demo
{
  class Program
  {
    void Main()
    {
      int factor = 2;
      int scale(x) => factor * x;
    }
  }
}
```

bind

bind

As opposed to dynamic scoping:

Not C#!

```
void theCaller()
{
  int factor = 2;
  scale(3);
}

    int scale(int x)
    {
      return x * factor;
    }
```

Summary

Partial function application in F# and C#

- Built into functional languages
- In C# modelled via Func delegates
- Some help comes from lambda syntax

Closures in C#

- Compiler rewrites code
- Closure becomes a proper object
- Free variables are fields in the closure

Summary

Mutable free variables

- Execution environment observes the same free variable as the function
- Changes to free variable observed by the lambda

Advice

- Avoid changing free variables
- Keep free variables immutable

Next module:
Treating All Objects as Values