Language mechanics

DEVELOPMENTOR

DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

Objectives

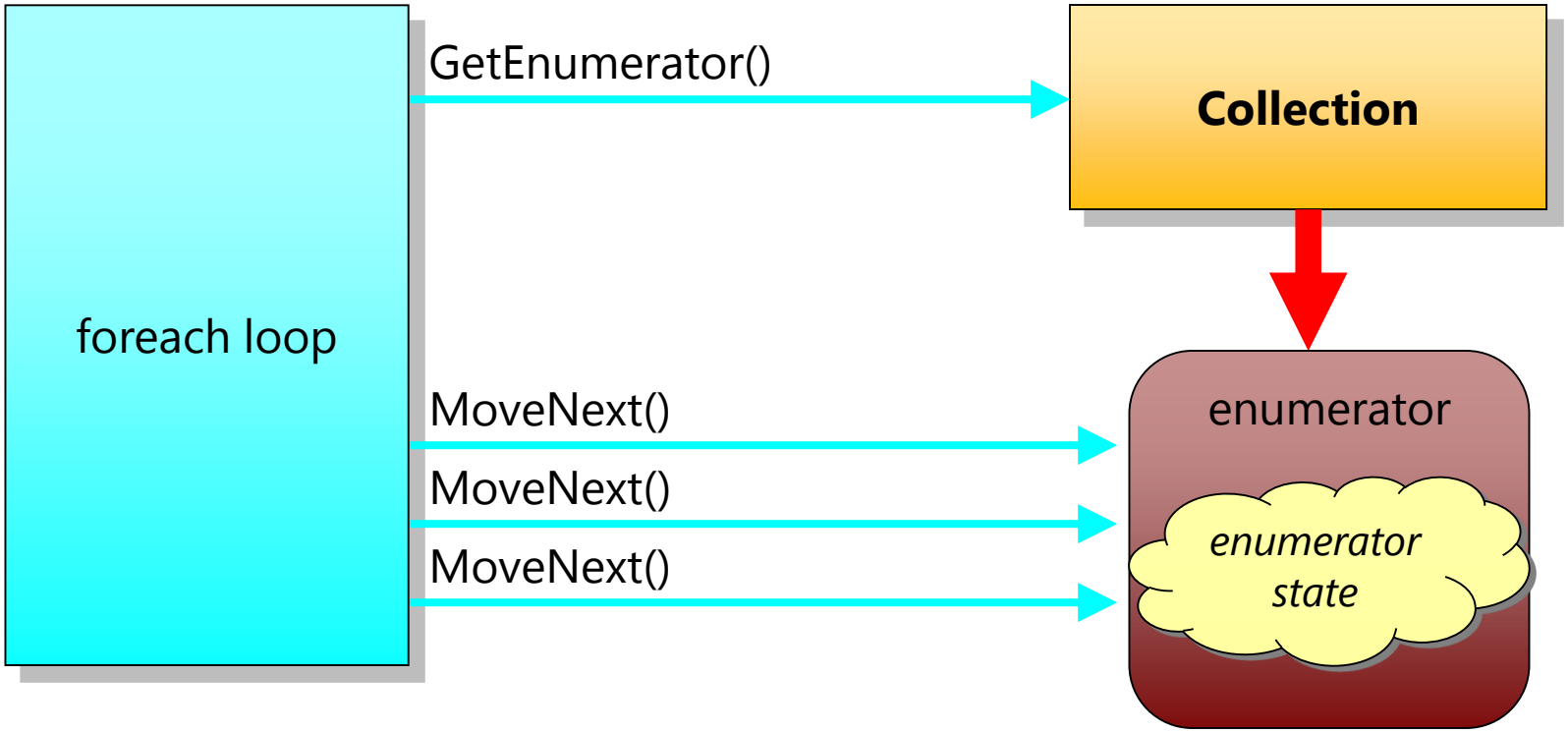
- Language evolution
- C# 2
  - Anonymous methods
  - Iterator methods
- C# 3
  - Foundation for Linq
- C# 4
  - Dynamic

C#1

- Language/Compiler wants to
  - make code easy to read and write
- How ?
  - By hiding underlying mechanics
- C# 1, simple mapping from C# to IL
- Some exceptions
  - event
  - foreach

Foreach

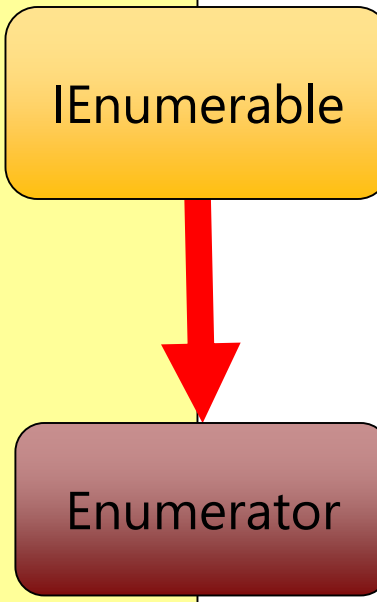
- Initially ask for Enumerator from IEnumerable
  - enumerator tracks client's progress
- Repeatedly calls `MoveNext` on enumerator



Iterators

- Common interface for iteration allows decoupling of
  - Producer and consumer
- Implementing IEnumerable<T> and IEnumerator<T> tedious

```
class X {
  class X_Enumerable : IEnumerable<int> {
     int min; int max; int divisor;
     IEnumerator GetEnumerator() {
       return new X_Enumerator(min, max, divisor);
     }
  }
class X_Enumerator : IEnumerator<int> {
    int current;
    Enumerator(int min, int max, int divisor) { ... }
    bool MoveNext() { ... }
    int Current() { get { return current; } }
    ...
  }
IEnumerable EnumDivisibleBy(int min, int max, int divisor) {
  return new X_Enumerable(min, max, divisor);
}
```



Simplified iterators

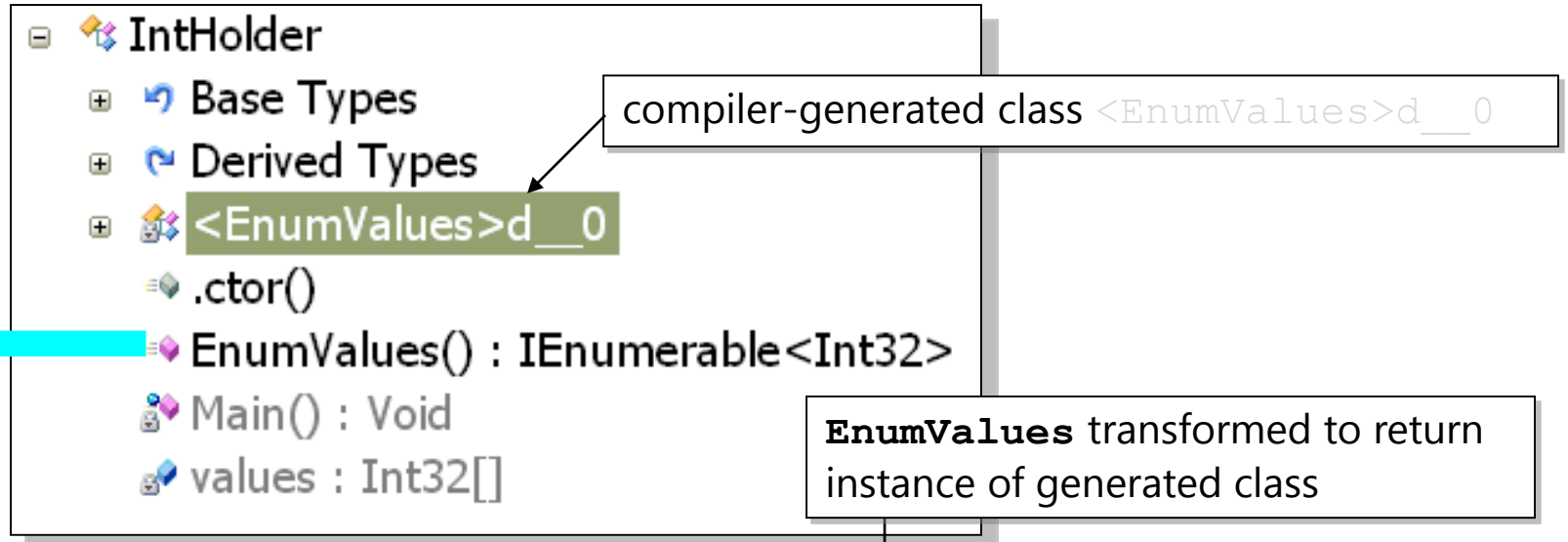
- Better to describe intentions to compiler and let compiler build tedious mechanics
- C# 2 enables this through use of two new keywords
  - yield return and yield break

```
IEnumerable<int> DivisibleBy(int min, int max, int divisor){
  for(int i=min; i<max; i++)
    if(i % divisor == 0)
       yield return i;
}
```

```
foreach(int i in DivisibleBy(0, 50, 7))
  Console.WriteLine(i);
```

```
0
7
14
21
```

Iterator method - compiled result



Issues with Delegates

- In C# 1 , delegate methods often short one line pieces of functionality
  - Method is only intended for specific use
  - Would be more preferable to explicitly wrap up arbitrary blocks of code

```
List<int> primes = new List<int>() { 2, 3, 5, 7, 11, 13, 17, 19 };
primes.RemoveAll(IsSmallPrime);

   . . .

private static bool IsSmallPrime(int prime)
{
  return prime < 11;
}
```

Anonymous methods

- In C#2 you can create delegate instances that can wrap inline blocks of code
  - Intent is clearer
  - Closer to strategy pattern

```
List<int> primes = new List<int>() { 2, 3, 5, 7, 11, 13, 17, 19 };

 primes.RemoveAll(delegate(int prime)
 {
    return prime < 11;
 });
```

Anonymous methods, under the Hood

- Compiler generates static method on callers class
  - wires up a delegate instance to that static method
  - No performance benefit, just easier to code

```
public class Program
{
   [CompilerGenerated]
   private static bool <FilterPrimes>b__1(int prime)
   {
      return (prime < 11);
   }

   . . .

 primes.RemoveAll( <FilterPrimes>b__1 );
}
```

Capturing state

- Anonymous methods allow the use of local variables.
  - Useful when method requires more information than supplied as parameters
- Be careful using this technique with asynchronous programming

```
List<int> primes = new List<int>() { 2, 3, 5, 7, 11, 13, 17, 19 };

int total = 0;
primes.ForEach( delegate(int prime)
{
  total += prime;
});

Console.WriteLine(total);
```

Capturing State, under the hood

- Compiler now creates new class that contains
  - Anonymous method
  - Instance fields for each local variable used inside the anonymous method
- Instance of new type is created on heap
  - Instead of allocated on the stack
  - Reference kept on the stack
- Delegate now wraps instance method

```
<>c__DisplayClass2 CS$<>8__locals3 = new <>c__DisplayClass2();
List<int> primes = new List<int>() { 2,3,5,7,11,13,17,19 }

CS$<>8__locals3.total = 0;

Primes
.ForEach(new Action<int>(CS$<>8__locals3.<FilterPrimes>b__1));

Console.WriteLine(CS$<>8__locals3.total);
```

Lambda Expressions

- C# 3 introduced more concise syntax for anonymous methods
  - Based on lambda calculus
  - Compiler can infer parameter and return types
  - Uses same compiler techniques as anonymous methods
- Ideal for functional style programming

```
List<int> primes = new List<int>(){ 2, 3, 5, 7, 11, 13, 17, 19 };

primes.RemoveAll(prime => prime < 11);
```

More Lambda Expressions

- Lambda method can take
  - No parameters
  - 1 parameter
  - Many parameters
- Lambda body can be
  - Single expression
  - Multi statement

```
() =>

p =>

(lhs,rhs) =>
```

```
(lhs,rhs) => lhs + rhs
```

```
(lhs,rhs) =>
{
    int total = lhs + rhs;
    return total;
}
```

Utility methods

- Adding missing functionality traditionally meant building Utility classes.
  - Usage model not OO like
  - can be hard to discover methods.
- Preference would be to invoke Capitalise as an object method

```
public static class StringUtil {
  public static string Capitialise(string str)
  {
    return str.Substring(0, 1).ToUpper() + str.Substring(1);
  }
}
```

```
Console.WriteLine( StringUtil.Capitialise("andy") );
```

```
Console.WriteLine( "andy".Capitialise() );
```

Extension methods

- Provide a more OO style invocation
  - Static methods inside static classes can be made to look like object methods
  - First parameter of static method used to denote type to extend, prefixed with the this keyword
- Now possible to invoke Capitalise in an OO style
- Extension method class must be in a visible namespace, to be included

```
public static class StringUtil
{
  public static string Capitialise( this string str ){
    return str.Substring(0, 1).ToUpper() +  str.Substring(1);
  }
}
```

```
Console.WriteLine( "andy".Capitialise() );
```

Property Initialisers

- Initialise an objects properties at creation time
  - Not a replacement for constructors
- Single statement initialisation

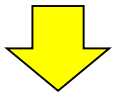
```
public class Person
{
  public string Name { get; set; }
  public int Age { get; set; }
}
```

```
Person andy = new Person { Name = "Andy", Age = 21 };
```

Anonymous types

- Compiler builds type based on shape of data
- Type is not known until runtime
- Must use var keyword to tell compiler to infer type
  - Has few use cases, one being projections in LINQ
  - var keyword can also be used to simplify variable declarations

```
var p = new { Name = "Andy", Age = 21 };
Console.WriteLine(p);
```



```
{ Name = Andy, Age = 21 }
```

```
var cache = new Dictionary<string, List<string>>();
```

Mixing Query Mechanics and Intent

- Traditional query against a collection of objects
  - Mixes mechanics and intent
  - Intent is not as clear as specific query languages
- Query languages like SQL make their intent clearer

```
public List<Person> GetLittlePeople(List<Person> people)
{
   List<Person> littlePeople = new List<Person>();
   foreach (Person p in people){
   if (p.Height < 2.0m)
   {
     littlePeople.Add(p);
   }
  }
  return littlePeople;
}
```

```
SELECT Name, Age From People where Height < 2
```

Language Integrated Query (LINQ)

- C# 3 introduces LINQ
  - Define intent not mechanics
- Query rules
  - All queries must start with a from
  - All queries must end with a select or group

```
var littlePeople = from person in people
                   where person.Height < 2
                   select new { person.Name, person.Age };
```

DIY LINQ

- Separate intent from mechanics
  - Create objects that define the query
  - Write code to execute the query
- Deferred execution
- Utilising extension methods

```
IEnumerable<Person> littlePeopleQuery =
                                people
                                .Where(p => p.Height < 2)

List<Person> littlePeople = new List(littlePeopleQuery);
```

```
public static IEnumerable<T> Where<T>(this IEnumerable<T> items,
                                      Predicate<T> condition){
  foreach (T item in items){
   if (condition(item))
     yield return item;
  }
}
```

- LINQ to objects defines extension methods on IEnumerable<T>
  - IEnumerable<T> is heavily implemented across various object containers
- Extension methods defined inside
  - System.Linq.Enumerable
  - System.Core

```
using System.Linq

List<Person> littlePeople = people
                            .Where(p => p.Height < 2)
                            .OrderBy(p => p.Height)
                            .ToList();
```

Language Integrated Query (LINQ)

- Extension methods go some of the way
- C# language has keywords that map onto a sub set of extension methods
  - Compiler emits code that uses extension methods

```
var littlePeople = from person in people
                   where person.Height < 2
                   select new { person.Name, person.Age };
```

```
var littlePeople = people
                   .Where(p => p.Height < 2)
                   .Select(p => new { Name = p.Name, Age = p.Age});
```

Dynamic Typing

- C# supports dynamic typing
  - Declare variables as **`dynamic`**
- **Principally there to aid interop**
  - **Dynamic languages**
  - **COM**
- Variable "typed" to object it points to

```
dynamic d = 10;
d = 2.4;
d = "Hello";
```

d is int

d is double

d is string

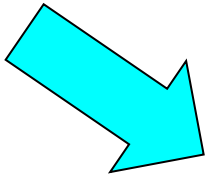
Dynamic Typing and Reflection

- Dynamic makes late bound invocation simpler

```
public double CallAdd(object o, double x, double y)
{
  Type t = o.GetType();
  MethodInfo mi = t.GetMethod("Add",
                    new Type[] { typeof(double), typeof(double) });

  object ret = mi.Invoke(o, new object[] { x, y });

  return (double)ret;
}
```



```
public double CallAdd(object o, double x, double y)
{
  dynamic d = o;

  return d.Add(x, y);
}
```

Dynamic mechanics

- Is Dynamic just compiler generated reflection
  - For that to be true it would have similar performance as reflection
  - It is in fact a lot faster than reflection
- Dynamic utilises the Dynamic Language Runtime (DLR)
  - Created to support languages like Iron Ruby, Iron Python
  - Utilises call site caching
    - reflection to emit delegate invocation code first time type is encountered at a specific call site
    - Utilises cached delegate for all future invocations

Summary

- C# language has evolved along a common theme
  - Hiding mechanics, raising level of abstraction
  - Making code easy and easy to produce and consume
- Understanding what is under the hood helps to
  - Prevent writing stupid code
  - Write code that is still performant
- The story continues in C# 5 ..