Lesson 5-B
Lazy/Deferred Execution

Visual C# 2012 How to Program

Introduction

- Lazy in programming is *not* a four-letter word…
- Doing things lazily can be (and usually is) a good practice
- A tale of two students:
  - *A professor assigns a large assignment on Monday, due the following Monday. Student A goes home that night and works all night on it making sure it's perfect so that it will be completely off of their mind. Student B procrastinates, planning on finishing it at the last second the following Sunday night.*
  - *Tuesday, the professor walks in and announces that the whole thing was a mistake and no assignment is actually due. Which student's approach ended up being better?*
  - Moral: You will waste less time if you only do the exact work you have to do when you know you have to do it. Wait until you are sure it is required.

Yield Return

- Material taken from: *https://www.kenneth-truyers.net/2016/05/12/yield-return-in-c/*
- Yield return is a good example of the concept of deferred execution in C#
- What is an IEnumerable<T>?
  - An interface that ensures there will be an iterator which can be used to go through the collection (implicitly used by foreach loops, remember)
  - As we noted in the last lesson, LINQ queries, that return IEnumerables, use deferred execution
    - This means an IEnumerable is not necessarily a collection sitting somewhere in memory, it's merely something you can use to get the values of a collection by means of an iterator, those values may not actually exist until you access them
    - In your own functions, this is accomplished with **yield return**

Yield Return

- Material taken from: *https://www.kenneth-truyers.net/2016/05/12/yield-return-in-c/*
- Yield return means that a function will stop executing and return a value, but save its state so that it can be resumed from the same point of execution if called again.

```
IEnumerable<int> GetSomeNumbers()
{
   yield return 1;
   yield return 2;
   yield return 3;
}
…
foreach(var number in GetSomeNumbers())
   Console.WriteLine(number);
```

- Step through this code in a debugger to see the actual execution

Yield Return

- Material taken from: *https://www.kenneth-truyers.net/2016/05/12/yield-return-in-c/*
- If you want to understand the effect, imagine we had a class like this:

```
Class NumberGenerator()
{
    private int i = 0;
    public int getNextNum()
    {
        return i++;
    }
}
```

- So a function with a yield return statement is like an object that remembers state between calls
- The difference is, you can't iterate over the values the object returns…

Yield Return

- Material taken from: *https://www.kenneth-truyers.net/2016/05/12/yield-return-in-c/*
- A classic illustration of this is the ability to write an "infinite" loop without actually executing it infinitely
- Consider a function to return a list of numbers 1 – 5:

```
IEnumerable<int> GetSomeNumbers()
{
   var i = 0;
   var list = new List<int>();
   while (i < 5)
        list.add(++i);
   return list;
}
…
foreach(var number in GetSomeNumbers())
   Console.WriteLine(number);
```

- This code does what you would expect it to do, displaying numbers 1 – 5

Yield Return

- Material taken from: *https://www.kenneth-truyers.net/2016/05/12/yield-return-in-c/*
- A classic illustration of this is the ability to write an "infinite" loop without actually executing it infinitely
- Now consider a version that uses yield return:

```
IEnumerable<int> GetSomeNumbers()
{
   var i = 0;
   while (i < 5)
        yield return ++i;
}
…
foreach(var number in GetSomeNumbers())
   Console.WriteLine(number);
```

- This code accomplishes the same thing as the last version, but the execution is completely different. Rather than creating a full list at one time and returning it, we generate the numbers one at a time as needed.

Yield Return

- Material taken from: *https://www.kenneth-truyers.net/2016/05/12/yield-return-in-c/*
- A classic illustration of this is the ability to write an "infinite" loop without actually executing it infinitely
- Now suppose we made both versions infinite:

```
IEnumerable<int> GetSomeNumbers()
{
    var i = 0;
    var list = new List<int>();
    while (true)
           list.add(++i);
    return list;
}
…
foreach(var number in GetSomeNumbers().Take(5))
    Console.WriteLine(number);
```

```
IEnumerable<int> GetSomeNumbers()
{
    var i = 0;
    while (true)
           yield return ++i;
}
…
foreach(var number in GetSomeNumbers().Take(5))
    Console.WriteLine(number);
```

- Take(5) takes the first 5 numbers in the IEnumerable, and it uses an iterator with deferred execution to do so (otherwise the example wouldn't work)
- So the version on the left enters an infinite loop, never returning the list, while the version on the right works because there is never a list to return, it simply gives back the numbers as requested, but uses while(true) so that there is no upper limit to what can be returned

Yield Return

- Material taken from: *https://www.kenneth-truyers.net/2016/05/12/yield-return-in-c/*
- Exercise:
  - Suppose we want to study Fibonacci numbers and look for the first one that has some property. How many Fibonacci numbers do we need to generate? Of course, we don't know, that's the whole point.
    - So we can have a function that generates Fibonaccis and studies them in the function, or calls a different function
    - Or use some global state
    - But a more elegant solution is to have a function that gives an IEnumerable of *all* Fibonacci numbers, and iterate through it as if it were any other collection
  - Write this function…

Multiple Execution

- Material taken from: *https://www.kenneth-truyers.net/2016/05/12/yield-return-in-c/*
- A potential downside of deferred execution in LINQ queries is the risk of running the same query twice

```
IEnumerable<Invoice> GetInvoices() {
    for(var i = 1; i<11; i++)
        yield return new Invoice() {Amount = i * 10};
}

void DoubleAmounts(IEnumerable<Invoice> invoices) {
    foreach(var invoice in invoices)
        invoice.Amount = invoice.Amount * 2;
}

var invoices = GetInvoices();
DoubleAmounts(invoices);

Console.WriteLine(invoices.First().Amount);
```

- What do we expect the output to be?
  - You may say 20, but it actually is 10. `invoices` is not an actual collection in memory, it is an iterator that can work through a series of numbers, generating them as requested. The loop in `DoubleAmounts` iterates, then all values are lost, and then the call to `invoices.First()` starts the iteration all over again.

Multiple Execution

- Material taken from: *https://www.kenneth-truyers.net/2016/05/12/yield-return-in-c/*
- A potential downside of deferred execution in LINQ queries is the risk of running the same query twice

```
IEnumerable<Invoice> GetInvoices() {
    for(var i = 1; i<11; i++)
        yield return new Invoice() {Amount = i * 10};
}

void DoubleAmounts(IEnumerable<Invoice> invoices) {
    foreach(var invoice in invoices)
        invoice.Amount = invoice.Amount * 2;
}

var invoices = GetInvoices().ToList();
DoubleAmounts(invoices);

Console.WriteLine(invoices.First().Amount);
```

- Contrast with this code where we put the values in a list..

Multiple Execution

- Material taken from: *https://www.kenneth-truyers.net/2016/05/12/yield-return-in-c/*
- A potential downside of deferred execution in LINQ queries is the risk of running the same query twice

```
var HighEarners = Employees.Where(e => e.Salary > 100000);

foreach(var emp in HighEarners)
    // do something with all the high earners

...

foreach(var emp in HighEarners)
    // do something with all the high earners
```

- This is a classic inefficiency, Resharper will warn you about this
- You will actually perform the task of extracting a filtered collection twice in this code
- The solution is to get a list instead of an IEnumerable

```
var HighEarners = Employees.Where(e => e.Salary > 100000).ToList();
```

- There are two reasons why you would NOT do this:
  - There is a possibility that the result of the query is different in the two locations in code
  - Memory constraints require working through a "stream" of employees rather than loading them all into memory at one time (yield return is related to the field of Big Data for this reason)

Multiple Execution

- Material taken from: *https://www.kenneth-truyers.net/2016/05/12/yield-return-in-c/*
- But remember this causes code to run *more* efficiently in many cases also

```
var dollarPrices = FetchProducts().Take(10)
                                  .Select(p => p.CalculatePrice())
                                  .OrderBy(price => price)
                                  .Take(5)
                                  .Select(price => ConvertToDollars(price));
```

- If FetchProducts did not use deferred execution it would process however many products there are, even though we only need 10 of them for our purposes here
- With deferred execution, Take(10) iterates only over the first 10 values of the IEnumerable returned by FetchProducts() and the others are not examined

Multiple Execution

- Material taken from: *https://www.kenneth-truyers.net/2016/05/12/yield-return-in-c/*
- The illustration of "custom iteration" in the webpage listed above is worth looking at, if we do not have time in class it's a good idea to see it on your own

IEnumerable and Immutability

- Understanding deferred execution helps us understand the IEnumerable interface a little better
- The interface revolves around the ability to iterate over the collection
  - For example, you can use the IEnumerable in a foreach loop
- Therefore, methods like Count, Sum, Where, ToList, all make sense, they can all be accomplished by means of iteration
- But there are no methods to change the collection, such as Add or Remove
- A List can have such methods, because it is a static collection, but an IEnumerable may only be an iterator that creates the items as it goes, you can't modify such a "collection"

IEnumerable and Immutability

- This allows a simple technique for information hiding
- Suppose we have a private collection such as a `List<T>` in our class
  - `private List<int> itemsList;`
- And we want to make these items *visible* outside of the class, but not allow any modifications to the `List` itself
- One simple solution is to create a property that returns the `List` as an `IEnumerable`
  - `public IEnumerable<int> Items => itemsList;`
- You are returning the `List`, which is an `IEnumerable<int>`, but the *type* you are returning is `IEnumerable<int>` so the code that accesses this property can only iterate over the list, it cannot access methods like `Add` and `Remove`
  - Presumably the caller can cast it if they know this is what you're doing
- Of course you always have to remember that if the elements are objects then the iterator is giving the objects and they can be modified, but if you only care about protecting the integrity of the `List` itself then this method will work
- Also, if performance is an issue you have to remember that the property will not allow random access

Deferred Execution – LINQ Inefficiencies

- Let's return to this example from the previous lesson:
- `var vals = values.Where(v => lst.All(v2 => v != v2));`
- vs.
- `var vals = values.Where(v => !lst.Contains(v));`
- If `lst` is sorted, we can check for whether or not it contains an element in O(log n) time, or if it's a hash set we can check in O(1) time, assuming the `Contains` method is written efficiently for that data structure
  - So Total time for searching all values is O(n log n) or O(n)
- But if we use a LINQ extension method such as `All`, it will possibly execute using an iterator (because of deferred execution), so it will not take advantage of any potential efficiency and it will require a full $O(n^2)$ running time
- This is also why calling a property such as `lst.Count` is, in theory, more efficient than a method such as `lst.Count()`
  - Though it's possible that `Count()` is written to use `Count`
  - If you are passing in a lambda expression of course you need the method

Deferred Execution – Memory Management

- Recall this example from the previous lesson:
- 1)
```
Emps.Where(e => !e.Hired).ToList()
                .ForEach(e => Emps.Remove(e));
```

- 2)
```
Emps = Emps.Where(e => e.Hired).ToList();
```
- Is there a difference?
- Suppose you had a separate IEnumerable:

```
var salaries = Emps.Select(e => e.Salary);
```

- And then "changed" Emps using one of the two methods above, would there be a difference?

20.2 Motivation for Generic Methods

- Overloaded methods are often used to perform similar operations on different types of data.
- To understand the motivation for generic methods, let's begin with an example (Fig. 20.1) that contains three overloaded `DisplayArray` methods (lines 23–29, lines 32–38 and lines 41–47).
- These methods display the elements of an int array, a `double` array and a `char` array, respectively.
- Soon, we'll reimplement this program more concisely and elegantly using a single generic method

```
1  // Fig. 20.1: OverloadedMethods.cs
2  // Using overloaded methods to display arrays of different types.
3  using System;
4
5  class OverloadedMethods
6  {
7     static void Main( string[] args )
8     {
9        // create arrays of int, double and char
10       int[] intArray = { 1, 2, 3, 4, 5, 6 };
11       double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
12       char[] charArray = { 'H', 'E', 'L', 'L', 'O' };
13
14       Console.WriteLine( "Array intArray contains:" );
15       DisplayArray( intArray ); // pass an int array argument
16       Console.WriteLine( "Array doubleArray contains:" );
17       DisplayArray( doubleArray ); // pass a double array argument
18       Console.WriteLine( "Array charArray contains:" );
19       DisplayArray( charArray ); // pass a char array argument
20    } // end Main
21
```

Fig. 20.1 | Using overloaded methods to display arrays of different types. (Part 1 of 3.)

```
22      // output int array
23      private static void DisplayArray( int[] inputArray )
24      {
25         foreach ( int element in inputArray )
26            Console.Write( element + " " );
27
28         Console.WriteLine( "\n" );
29      } // end method DisplayArray
30
31      // output double array
32      private static void DisplayArray( double[] inputArray )
33      {
34         foreach ( double element in inputArray )
35            Console.Write( element + " " );
36
37         Console.WriteLine( "\n" );
38      } // end method DisplayArray
```

Fig. 20.1 | Using overloaded methods to display arrays of different types. (Part 2 of 3.)

```
39
40      // output char array
41      private static void DisplayArray( char[] inputArray )
42      {
43         foreach ( char element in inputArray )
44            Console.Write( element + " " );
45
46         Console.WriteLine( "\n" );
47      } // end method DisplayArray
48   } // end class OverloadedMethods
```

```
Array intArray contains:
1 2 3 4 5 6

Array doubleArray contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7

Array charArray contains:
H E L L O
```

Fig. 20.1 | Using overloaded methods to display arrays of different types. (Part 3 of 3.)

20.3 Generic-Method Implementation

- Figure 20.3 reimplements the app of Fig. 20.1 using a generic `DisplayArray` method (lines 24–30).
- Note that the `DisplayArray` method calls in lines 16, 18 and 20 are identical to those of Fig. 20.1, the outputs of the two apps are identical and the code in Fig. 20.3 is 17 lines *shorter* than that in Fig. 20.1.
- As illustrated in Fig. 20.3, generics enable us to create and test our code once, then *reuse* it for many different types of data.
- This demonstrates the expressive power of generics.

```
1  private static void DisplayArray( T[] inputArray )
2  {
3     foreach ( T element in inputArray )
4        Console.Write( element + " " );
5
6     Console.WriteLine( "\n" );
7  } // end method DisplayArray
```

Fig. 20.2 | `DisplayArray` method in which actual type names are replaced by convention with the generic name `T`. Again, this code will *not* compile.

```
1  // Fig. 20.3: GenericMethod.cs
2  // Using overloaded methods to display arrays of different types.
3  using System;
4  using System.Collections.Generic;
5
6  class GenericMethod
7  {
8     public static void Main( string[] args )
9     {
10       // create arrays of int, double and char
11       int[] intArray = { 1, 2, 3, 4, 5, 6 };
12       double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
13       char[] charArray = { 'H', 'E', 'L', 'L', 'O' };
14
15       Console.WriteLine( "Array intArray contains:" );
16       DisplayArray( intArray ); // pass an int array argument
17       Console.WriteLine( "Array doubleArray contains:" );
18       DisplayArray( doubleArray ); // pass a double array argument
19       Console.WriteLine( "Array charArray contains:" );
20       DisplayArray( charArray ); // pass a char array argument
21    } // end Main
22
```

Fig. 20.3 | Using a generic method to display arrays of different types. (Part 1 of 2.)

```
23      // output array of all types
24      private static void DisplayArray< T >( T[] inputArray )
25      {
26         foreach ( T element in inputArray )
27            Console.Write( element + " " );
28
29         Console.WriteLine( "\n" );
30      } // end method DisplayArray
31   } // end class GenericMethod
```

```
Array intArray contains:
1 2 3 4 5 6

Array doubleArray contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7

Array charArray contains:
H E L L O
```

Fig. 20.3 | Using a generic method to display arrays of different types. (Part 2 of 2.)

Common Programming Error 20.1

If you forget to include the type-parameter list when declaring a generic method, the compiler will not recognize the type-parameter names when they're encountered in the method. This results in compilation errors.

Good Programming Practice 20.1

It's recommended that type parameters be specified as individual capital letters. Typically, a type parameter that represents the type of an element in an array (or other collection) is named `E` for "element" or `T` for "type."

Common Programming Error 20.2

If the compiler cannot find a single nongeneric or generic method declaration that's a best match for a method call, or if there are multiple best matches, a compilation error occurs.

20.4 Type Constraints

`IComparable<T>` Interface

- It's possible to compare two objects of the *same* type if that type implements the generic interface `IComparable<T>` (of namespace `System`).
- A benefit of implementing interface `IComparable<T>` is that `IComparable<T>` objects can be used with the *sorting* and *searching* methods of classes in the `System.Collections.Generic` namespace—we discuss those methods in Chapter 21.
- The structures in the Framework Class Library that correspond to the simple types *all* implement this interface.

20.4 Type Constraints (cont.)

Specifying Type Constraints

- Even though `IComparable` objects can be compared, they cannot be used with generic code by default, because not all types implement interface `IComparable<T>`.
- However, we can restrict the types that can be used with a generic method or class to ensure that they meet certain requirements.
- This feature—known as a type constraint—restricts the type of the argument supplied to a particular type parameter.

20.4 Type Constraints (cont.)

- Figure 20.4 declares method `Maximum` (lines 20–34) with a type constraint that requires each of the method's arguments to be of type `IComparable<T>`.
- This restriction is important, because not all objects can be compared.
- However, all `IComparable<T>` objects are guaranteed to have a `CompareTo` method that can be used in method `Maximum` to determine the largest of its three arguments.

```
1  // Fig. 20.4: MaximumTest.cs
2  // Generic method Maximum returns the largest of three objects.
3  using System;
4
5  class MaximumTest
6  {
7     public static void Main( string[] args )
8     {
9        Console.WriteLine( "Maximum of {0}, {1} and {2} is {3}\n",
10          3, 4, 5, Maximum( 3, 4, 5 ) );
11       Console.WriteLine( "Maximum of {0}, {1} and {2} is {3}\n",
12          6.6, 8.8, 7.7, Maximum( 6.6, 8.8, 7.7 ) );
13       Console.WriteLine( "Maximum of {0}, {1} and {2} is {3}\n",
14          "pear", "apple", "orange",
15          Maximum( "pear", "apple", "orange" ) );
16    } // end Main
17
18    // generic function determines the
19    // largest of the IComparable objects
20    private static T Maximum< T >( T x, T y, T z )
21       where T : IComparable< T >
22    {
23       T max = x; // assume x is initially the largest
```

Fig. 20.4 | Generic method `Maximum` returns the largest of three objects. (Part 1 of 2.)

```
24
25          // compare y with max
26          if ( y.CompareTo( max ) > 0 )
27             max = y; // y is the largest so far
28
29          // compare z with max
30          if ( z.CompareTo( max ) > 0 )
31             max = z; // z is the largest
32
33          return max; // return largest object
34       } // end method Maximum
35    } // end class MaximumTest
```

```
Maximum of 3, 4 and 5 is 5

Maximum of 6.6, 8.8 and 7.7 is 8.8

Maximum of pear, apple and orange is pear
```

Fig. 20.4 | Generic method `Maximum` returns the largest of three objects. (Part 2 of 2.)

20.4 Type Constraints (cont.)

- C# provides several kinds of type constraints.
- A class constraint indicates that the type argument must be an object of a specific base class or one of its subclasses.
- An interface constraint indicates that the type argument's class must implement a specific interface.
- The type constraint in line 21 is an interface constraint, because `IComparable<T>` is an interface.

20.4 Type Constraints (cont.)

- You can specify that the type argument must be a reference type or a value type by using the reference-type constraint (`class`) or the value-type constraint (`struct`), respectively.
- Finally, you can specify a constructor `constraint—new()`—to indicate that the generic code can use operator new to create new objects of the type represented by the type parameter.

20.4 Type Constraints (cont.)

- If a type parameter is specified with a constructor constraint, the type argument's class must provide a `public` parameterless or default constructor to ensure that objects of the class can be created without passing constructor arguments; otherwise, a compilation error occurs.
- It's possible to apply multiple constraints to a type parameter.

20.4 Type Constraints (cont.)

- To do so, simply provide a comma-separated list of constraints in the `where` clause.
- If you have a class constraint, reference-type constraint or value-type constraint, it must be listed first—only one of these types of constraints can be used for each type parameter.
- Interface constraints (if any) are listed next.
- The constructor constraint is listed last (if there is one).

20.5 Overloading Generic Methods

- A generic method may be overloaded.
- Each overloaded method must have a unique signature (as discussed in Chapter 7).
- A class can provide two or more generic methods with the same name but *different* method parameters.
- A generic method can be overloaded by nongeneric methods with the same method name.
- When the compiler encounters a method call, it searches for the method declaration that best matches the method name and the argument types specified in the call.

20.6 Generic Classes

- With a generic class, you can use a simple, concise notation to indicate the actual type(s) that should be used in place of the class's type parameter(s).
- At compilation time, the compiler ensures your code's type safety, and the runtime system replaces type parameters with type arguments to enable your client code to interact with the generic class.

20.6 Generic Classes (cont.)

- One generic `Stack` class, for example, could be the basis for creating many `Stack` classes (e.g., "`Stack` of `double`," "`Stack` of `int`," "`Stack` of `char`," "`Stack` of `Employee`").
- Figure 20.5 presents a generic `Stack` class declaration.
- This class should not be confused with the class `Stack` from namespace `System.Collections.Generics`.

```
1  // Fig. 20.5: Stack.cs
2  // Generic class Stack.
3  using System;
4
5  class Stack< T >
6  {
7     private int top; // location of the top element
8     private T[] elements; // array that stores stack elements
9
10    // parameterless constructor creates a stack of the default size
11    public Stack()
12       : this( 10 ) // default stack size
13    {
14       // empty constructor; calls constructor at line 18 to perform init
15    } // end stack constructor
16
17    // constructor creates a stack of the specified number of elements
18    public Stack( int stackSize )
19    {
20       if ( stackSize > 0 ) // validate stackSize
21          elements = new T[ stackSize ]; // create stackSize elements
22       else
23          throw new ArgumentException( "Stack size must be positive." );
```

Fig. 20.5 | Generic class Stack. (Part 1 of 3.)

```
24
25          top = -1; // stack initially empty
26       } // end stack constructor
27
28       // push element onto the stack; if unsuccessful,
29       // throw FullStackException
30       public void Push( T pushValue )
31       {
32          if ( top == elements.Length - 1 ) // stack is full
33             throw new FullStackException( string.Format(
34                "Stack is full, cannot push {0}", pushValue ) );
35
36          ++top; // increment top
37          elements[ top ] = pushValue; // place pushValue on stack
38       } // end method Push
39
40       // return the top element if not empty,
41       // else throw EmptyStackException
42       public T Pop()
43       {
44          if ( top == -1 ) // stack is empty
45             throw new EmptyStackException( "Stack is empty, cannot pop" );
46
```

Fig. 20.5 | Generic class Stack. (Part 2 of 3.)

```
47          --top; // decrement top
48          return elements[ top + 1 ]; // return top value
49       } // end method Pop
50    } // end class Stack
```

Fig. 20.5 | Generic class `Stack`. (Part 3 of 3.)

20.6 Generic Classes (cont.)

- Classes `FullStackException` (Fig. 20.6) and `EmptyStackException` (Fig. 20.7) each provide a parameterless constructor, a one-argument constructor of exception classes (as discussed in Section 13.8) and a two-argument constructor for creating a new exception using an existing one.
- The parameterless constructor sets the default error message while the other two constructors set custom error messages.

```
1  // Fig. 20.6: FullStackException.cs
2  // FullStackException indicates a stack is full.
3  using System;
4
5  class FullStackException : Exception
6  {
7     // parameterless constructor
8     public FullStackException() : base( "Stack is full" )
9     {
10       // empty constructor
11    } // end FullStackException constructor
12
13    // one-parameter constructor
14    public FullStackException( string exception ) : base( exception )
15    {
16       // empty constructor
17    } // end FullStackException constructor
18
19    // two-parameter constructor
20    public FullStackException( string exception, Exception inner )
21       : base( exception, inner )
22    {
23       // empty constructor
24    } // end FullStackException constructor
25 } // end class FullStackException
```

Fig. 20.6 | FullStackException indicates a stack is full.

```
1  // Fig. 20.7: EmptyStackException.cs
2  // EmptyStackException indicates a stack is empty.
3  using System;
4
5  class EmptyStackException : Exception
6  {
7     // parameterless constructor
8     public EmptyStackException() : base( "Stack is empty" )
9     {
10       // empty constructor
11    } // end EmptyStackException constructor
12
13    // one-parameter constructor
14    public EmptyStackException( string exception ) : base( exception )
15    {
16       // empty constructor
17    } // end EmptyStackException constructor
18
19    // two-parameter constructor
20    public EmptyStackException( string exception, Exception inner )
21       : base( exception, inner )
22    {
23       // empty constructor
24    } // end EmptyStackException constructor
25 } // end class EmptyStackException
```

Fig. 20.7 | `EmptyStackException` indicates a stack is empty.

20.6 Generic Classes (cont.)

- Now, let's consider an app (Fig. 20.8) that uses the `Stack` generic class.

```
1  // Fig. 20.8: StackTest.cs
2  // Testing generic class Stack.
3  using System;
4
5  class StackTest
6  {
7     // create arrays of doubles and ints
8     private static double[] doubleElements =
9        new double[]{ 1.1, 2.2, 3.3, 4.4, 5.5, 6.6 };
10    private static int[] intElements =
11       new int[]{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };
12
13    private static Stack< double > doubleStack; // stack stores doubles
14    private static Stack< int > intStack; // stack stores int objects
15
16    public static void Main( string[] args )
17    {
18       doubleStack = new Stack< double >( 5 ); // stack of doubles
19       intStack = new Stack< int >( 10 ); // stack of ints
20
21       TestPushDouble(); // push doubles onto doubleStack
22       TestPopDouble(); // pop doubles from doubleStack
23       TestPushInt(); // push ints onto intStack
24       TestPopInt(); // pop ints from intStack
25    } // end Main
```

Fig. 20.8 | Testing generic class Stack. (Part 1 of 7.)

```
26
27      // test Push method with doubleStack
28      private static void TestPushDouble()
29      {
30         // push elements onto stack
31         try
32         {
33            Console.WriteLine( "\nPushing elements onto doubleStack" );
34
35            // push elements onto stack
36            foreach ( var element in doubleElements )
37            {
38               Console.Write( "{0:F1} ", element );
39               doubleStack.Push( element ); // push onto doubleStack
40            } // end foreach
41         } // end try
42         catch ( FullStackException exception )
43         {
44            Console.Error.WriteLine();
45            Console.Error.WriteLine( "Message: " + exception.Message );
46            Console.Error.WriteLine( exception.StackTrace );
47         } // end catch
48      } // end method TestPushDouble
49
```

Fig. 20.8 | Testing generic class Stack. (Part 2 of 7.)

```
50      // test Pop method with doubleStack
51      private static void TestPopDouble()
52      {
53         // pop elements from stack
54         try
55         {
56            Console.WriteLine( "\nPopping elements from doubleStack" );
57
58            double popValue; // store element removed from stack
59
60            // remove all elements from stack
61            while ( true )
62            {
63               popValue = doubleStack.Pop(); // pop from doubleStack
64               Console.Write( "{0:F1} ", popValue );
65            } // end while
66         } // end try
67         catch ( EmptyStackException exception )
68         {
69            Console.Error.WriteLine();
70            Console.Error.WriteLine( "Message: " + exception.Message );
71            Console.Error.WriteLine( exception.StackTrace );
72         } // end catch
73      } // end method TestPopDouble
```

Fig. 20.8 | Testing generic class Stack. (Part 3 of 7.)

```
74
75      // test Push method with intStack
76      private static void TestPushInt()
77      {
78         // push elements onto stack
79         try
80         {
81            Console.WriteLine( "\nPushing elements onto intStack" );
82
83            // push elements onto stack
84            foreach ( var element in intElements )
85            {
86               Console.Write( "{0} ", element );
87               intStack.Push( element ); // push onto intStack
88            } // end foreach
89         } // end try
90         catch ( FullStackException exception )
91         {
92            Console.Error.WriteLine();
93            Console.Error.WriteLine( "Message: " + exception.Message );
94            Console.Error.WriteLine( exception.StackTrace );
95         } // end catch
96      } // end method TestPushInt
97
```

Fig. 20.8 | Testing generic class `Stack`. (Part 4 of 7.)

```
98      // test Pop method with intStack
99      private static void TestPopInt()
100     {
101        // pop elements from stack
102        try
103        {
104           Console.WriteLine( "\nPopping elements from intStack" );
105
106           int popValue; // store element removed from stack
107
108           // remove all elements from stack
109           while ( true )
110           {
111              popValue = intStack.Pop(); // pop from intStack
112              Console.Write( "{0} ", popValue );
113           } // end while
114        } // end try
115        catch ( EmptyStackException exception )
116        {
117           Console.Error.WriteLine();
118           Console.Error.WriteLine( "Message: " + exception.Message );
119           Console.Error.WriteLine( exception.StackTrace );
120        } // end catch
121     } // end method TestPopInt
122  } // end class StackTest
```

Fig. 20.8 | Testing generic class Stack. (Part 5 of 7.)

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5 6.6
Message: Stack is full, cannot push 6.6
   at Stack`1.Push(T pushValue) in
      c:\examples\ch22\Fig22_05_08\Stack\Stack\Stack.cs:line 36
   at StackTest.TestPushDouble() in
      c:\examples\ch22\Fig22_05_08\Stack\Stack\StackTest.cs:line 39
```

Fig. 20.8 | Testing generic class `Stack`. (Part 6 of 7.)

```
Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Message: Stack is empty, cannot pop
   at Stack`1.Pop() in
      c:\examples\ch22\Fig22_05_08\Stack\Stack\Stack.cs:line 47
   at StackTest.TestPopDouble() in
      c:\examples\ch22\Fig22_05_08\Stack\Stack\StackTest.cs:line 63

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10 11
Message: Stack is full, cannot push 11
   at Stack`1.Push(T pushValue) in
      c:\examples\ch22\Fig22_05_08\Stack\Stack\Stack.cs:line 36
   at StackTest.TestPushInt() in
      c:\examples\ch22\Fig22_05_08\Stack\Stack\StackTest.cs:line 87

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Message: Stack is empty, cannot pop
   at Stack`1.Pop() in
      c:\examples\ch22\Fig22_05_08\Stack\Stack\Stack.cs:line 47
   at StackTest.TestPopInt() in
      c:\examples\ch22\Fig22_05_08\Stack\Stack\StackTest.cs:line 111
```

Fig. 20.8 | Testing generic class `Stack`. (Part 7 of 7.)

20.6 Generic Classes (cont.)

- Figure 20.9 declares generic method `TestPush` (lines 33–54) to perform the same tasks as `TestPushDouble` and `TestPushInt` in Fig. 20.8—that is, `Push` values onto a `Stack<T>`.
- Similarly, generic method `TestPop` (lines 57–79) performs the same tasks as `TestPopDouble` and `TestPopInt` in Fig. 20.8—that is, `Pop` values off a `Stack<T>`.

```
1  // Fig. 20.9: StackTest.cs
2  // Testing generic class Stack.
3  using System;
4  using System.Collections.Generic;
5
6  class StackTest
7  {
8     // create arrays of doubles and ints
9     private static double[] doubleElements =
10       new double[] { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6 };
11    private static int[] intElements =
12       new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };
13
14    private static Stack< double > doubleStack; // stack stores doubles
15    private static Stack< int > intStack; // stack stores int objects
16
17    public static void Main( string[] args )
18    {
19       doubleStack = new Stack< double >( 5 ); // stack of doubles
20       intStack = new Stack< int >( 10 ); // stack of ints
21
22       // push doubles onto doubleStack
23       TestPush( "doubleStack", doubleStack, doubleElements );
```

Fig. 20.9 | Testing generic class Stack. (Part 1 of 5.)

```
24          // pop doubles from doubleStack
25          TestPop( "doubleStack", doubleStack );
26          // push ints onto intStack
27          TestPush( "intStack", intStack, intElements );
28          // pop ints from intStack
29          TestPop( "intStack", intStack );
30       } // end Main
31
32       // test Push method
33       private static void TestPush< T >( string name, Stack< T > stack,
34          IEnumerable< T > elements )
35       {
36          // push elements onto stack
37          try
38          {
39             Console.WriteLine( "\nPushing elements onto " + name );
40
41             // push elements onto stack
42             foreach ( var element in elements )
43             {
44                Console.Write( "{0} ", element );
45                stack.Push( element ); // push onto stack
46             } // end foreach
47          } // end try
```

Fig. 20.9 | Testing generic class Stack. (Part 2 of 5.)

```
48          catch ( FullStackException exception )
49          {
50             Console.Error.WriteLine();
51             Console.Error.WriteLine( "Message: " + exception.Message );
52             Console.Error.WriteLine( exception.StackTrace );
53          } // end catch
54       } // end method TestPush
55
56       // test Pop method
57       private static void TestPop< T >( string name, Stack< T > stack )
58       {
59          // pop elements from stack
60          try
61          {
62             Console.WriteLine( "\nPopping elements from " + name );
63
64             T popValue; // store element removed from stack
65
```

Fig. 20.9 | Testing generic class Stack. (Part 3 of 5.)

```
66            // remove all elements from stack
67            while ( true )
68            {
69               popValue = stack.Pop(); // pop from stack
70               Console.Write( "{0} ", popValue );
71            } // end while
72         } // end try
73         catch ( EmptyStackException exception )
74         {
75            Console.Error.WriteLine();
76            Console.Error.WriteLine( "Message: " + exception.Message );
77            Console.Error.WriteLine( exception.StackTrace );
78         } // end catch
79      } // end TestPop
80   } // end class StackTest
```

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5 6.6
Message: Stack is full, cannot push 6.6
   at Stack`1.Push(T pushValue)
      in c:\examples\ch22\Fig22_09\Stack\Stack\Stack.cs:line 36
   at StackTest.TestPush[T](String name, Stack`1 stack, IEnumerable`1 elements)
      in c:\examples\ch22\Fig22_09\Stack\Stack\StackTest.cs:line 45
```

Fig. 20.9 | Testing generic class `Stack`. (Part 4 of 5.)

```
Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Message: Stack is empty, cannot pop
   at Stack`1.Pop() in c:\examples\ch22\Fig22_09\Stack\Stack\Stack.cs:line 47
   at StackTest.TestPop[T](String name, Stack`1 stack) in
      c:\examples\ch22\Fig22_09\Stack\Stack\StackTest.cs:line 69

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10 11
Message: Stack is full, cannot push 11
   at Stack`1.Push(T pushValue) in
      c:\examples\ch22\Fig22_09\Stack\Stack\Stack.cs:line 36
   at StackTest.TestPush[T](String name, Stack`1 stack, IEnumerable`1 elements)
      in c:\examples\ch22\Fig22_09\Stack\Stack\StackTest.cs:line 45

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Message: Stack is empty, cannot pop
   at Stack`1.Pop() in c:\examples\ch22\Fig22_09\Stack\Stack\Stack.cs:line 47
   at StackTest.TestPop[T](String name, Stack`1 stack) in
      c:\examples\ch22\Fig22_09\Stack\Stack\StackTest.cs:line 69
```

Fig. 20.9 | Testing generic class `Stack`. (Part 5 of 5.)

Custom Extension Methods

- C# allows us to define our own extension methods on native types
- Suppose we want to check if a string meets a certain format, such as is it a Touro ID
- We could write a method:

```
static bool IsStringTouroId(string s)
{
        // Just for demo purposes, this is not precise and should be done with a Regex
        return s.Substring(0, 4) == "T000" && s.Length == 9;
}
```

- And then we can test `if (IsStringTouroId(str))`
- A nice piece of syntactic sugar allows us to write `str.IsTouroId()` instead:
- Add a static class and add the following extension method:

```
public static bool IsTid(this string s)
{
    // Just for demo purposes, this is not precise and should be done with a Regex
    return s.Substring(0, 4) == "T000" && s.Length == 9;
}
```

- The `this` keyword makes this method available on all strings as an extension method
- We can also accept other params as well (in-class demo)

Custom Extension Methods

- C# allows us to define our own extension methods on native types
- If we want to define an extension method to complement LINQ, we may want it to use deferred execution
- The following method takes an IEnumerable<int> and returns an Ienumerable<int> of ints that had a higher value than their predecessor in the collection, demo of use in class

```
public static IEnumerable<int> AllIncreases(this IEnumerable<int> vals)
{
            int prev, curr;
            prev = vals.First();
            foreach (var val in vals.Skip(1))
            {
                curr = val;
                if (curr > prev)
                    yield return curr;
                prev = curr;
            }
 }
```

- This example is greatly enhanced if you use generics, and also if you overload and accept a lambda, demo in class

Deferred Execution

- Two more examples of deferred execution:
- In class demo: Track edges of a graph and add a custom extension “Getsert” method to a dictionary
- This illustrates the strong connection between functional programming and deferred execution, giving a function instead of a variable allows the variable to only be created when it’s actually necessary
- In class demo 2: Giving each node in a tree a property representing its level or distance from the root
- When do we calculate these values?
- Doing it all at one time could be a bottleneck and possibly wasteful, solving each node recursively on demand is redundant
- Instead we use something like a singleton where we set the value once recursively the first time it’s called

12.8 Operator Overloading

- You can overload most operators to make them sensitive to the context in which they are used.

Class `ComplexNumber`

- Class `ComplexNumber` (Fig. 12.17) overloads the plus (+), minus (-) and multiplication (*) operators to enable programs to add, subtract and multiply instances of class `ComplexNumber` using common mathematical notation.

```
1  // Fig. 12.17: ComplexNumber.cs
2  // Class that overloads operators for adding, subtracting
3  // and multiplying complex numbers.
4  using System;
5
6  public class ComplexNumber
7  {
8     // read-only property that gets the real component
9     public double Real { get; private set; }
10
11    // read-only property that gets the imaginary component
12    public double Imaginary { get; private set; }
13
14    // constructor
15    public ComplexNumber( double a, double b )
16    {
17       Real = a;
18       Imaginary = b;
19    } // end constructor
20
```

Fig. 12.17 | Class that overloads operators for adding, subtracting and multiplying complex numbers. (Part 1 of 3.)

```
21      // return string representation of ComplexNumber
22      public override string ToString()
23      {
24         return string.Format( "({0} {1} {2}i)",
25            Real, ( Imaginary < 0 ? "-" : "+" ), Math.Abs( Imaginary ) );
26      } // end method ToString
27
28      // overload the addition operator
29      public static ComplexNumber operator+ (
30         ComplexNumber x, ComplexNumber y )
31      {
32         return new ComplexNumber( x.Real + y.Real,
33            x.Imaginary + y.Imaginary );
34      } // end operator +
35
36      // overload the subtraction operator
37      public static ComplexNumber operator- (
38         ComplexNumber x, ComplexNumber y )
39      {
40         return new ComplexNumber( x.Real - y.Real,
41            x.Imaginary - y.Imaginary );
42      } // end operator -
43
```

Fig. 12.17 | Class that overloads operators for adding, subtracting and multiplying complex numbers. (Part 2 of 3.)

```
44      // overload the multiplication operator
45      public static ComplexNumber operator* (
46         ComplexNumber x, ComplexNumber y )
47      {
48         return new ComplexNumber(
49            x.Real * y.Real - x.Imaginary * y.Imaginary,
50            x.Real * y.Imaginary + y.Real * x.Imaginary );
51      } // end operator *
52   } // end class ComplexNumber
```

Fig. 12.17 | Class that overloads operators for adding, subtracting and multiplying complex numbers. (Part 3 of 3.)

12.8 Operator Overloading (Cont.)

- Keyword `operator`, followed by an operator symbol, indicates that a method overloads the specified operator.
- Methods that overload binary operators must take two arguments—the first argument is the *left* operand, and the second argument is the *right* operand.
- Overloaded operator methods must be `public` and `static`.

Software Engineering Observation 12.7

Overload operators to perform the same function or similar functions on class objects as the operators perform on objects of simple types. Avoid nonintuitive use of operators.

Software Engineering Observation 12.8

At least one parameter of an overloaded operator method must be a reference to an object of the class in which the operator is overloaded. This prevents you from changing how operators work on simple types.

12.8 Operator Overloading (Cont.)

Class `ComplexNumber`

- Class `ComplexTest` (Fig. 12.18) demonstrates the overloaded operators for adding, subtracting and multiplying `ComplexNumber`s.

```
1  // Fig. 12.18: ComplexTest.cs
2  // Overloading operators for complex numbers.
3  using System;
4
5  public class ComplexTest
6  {
7     public static void Main( string[] args )
8     {
9        // declare two variables to store complex numbers
10       // to be entered by user
11       ComplexNumber x, y;
12
13       // prompt the user to enter the first complex number
14       Console.Write( "Enter the real part of complex number x: " );
15       double realPart = Convert.ToDouble( Console.ReadLine() );
16       Console.Write(
17          "Enter the imaginary part of complex number x: " );
18       double imaginaryPart = Convert.ToDouble( Console.ReadLine() );
19       x = new ComplexNumber( realPart, imaginaryPart );
20
```

Fig. 12.18 | Overloading operators for complex numbers. (Part 1 of 3.)

```
21          // prompt the user to enter the second complex number
22          Console.Write( "\nEnter the real part of complex number y: " );
23          realPart = Convert.ToDouble( Console.ReadLine() );
24          Console.Write(
25             "Enter the imaginary part of complex number y: " );
26          imaginaryPart = Convert.ToDouble( Console.ReadLine() );
27          y = new ComplexNumber( realPart, imaginaryPart );
28
29          // display the results of calculations with x and y
30          Console.WriteLine();
31          Console.WriteLine( "{0} + {1} = {2}", x, y, x + y );
32          Console.WriteLine( "{0} - {1} = {2}", x, y, x - y );
33          Console.WriteLine( "{0} * {1} = {2}", x, y, x * y );
34       } // end method Main
35    } // end class ComplexTest
```

Fig. 12.18 | Overloading operators for complex numbers. (Part 2 of 3.)

```
Enter the real part of complex number x: 2
Enter the imaginary part of complex number x: 4

Enter the real part of complex number y: 4
Enter the imaginary part of complex number y: -2

(2 + 4i) + (4 - 2i) = (6 + 2i)
(2 + 4i) - (4 - 2i) = (-2 + 6i)
(2 + 4i) * (4 - 2i) = (16 + 12i)
```

Fig. 12.18 | Overloading operators for complex numbers. (Part 3 of 3.)

Operator Overloading

- Another nice customization is the ability to allow your class to be indexed with [ ]

```
class TreeNode
{
    public string Id { get; set; }
    public int Value { get; set; }
    public List<TreeNode> Children = new List<TreeNode>();

    public TreeNode(string id, int value)
    {
        Id = id;
        Value = value;
    }

    public TreeNode this[string id]
    {
        get { return Children.FirstOrDefault(c => c.Id == id); }
    }
}
```

- We can access children nodes using their Ids as a subscript