Whidbey Enhancements to C#

Jeff Vaughan
MSBuild Team
July 21, 2004
Outline

• Practical
  – Partial types
  – Static classes
  – Extern and the namespace alias qualifier

• Cool (and practical too)
  – Generics
  – Nullable Types
  – Anonymous Methods
Partial types simplify source management.

• Everett C# classes could not span multiple files.
  – Very large files can be unwieldy to work with.
  – Teams without source control can split classes and work concurrently without worrying about merging.
  – Code generators and programmers could conflict when editing some classes

• In Whidbey we can split a class across files by using the `partial` keyword and avoid these problems.
Partial types are combined at compile time.

File1.cs:

```csharp
public partial class Foo{
    void MyHandwrittenFun(){}
}

public partial class Foo{
    void AnotherFun(){}
}
```

File2.cs:

```csharp
public partial class Foo{
    void MyGeneratedFun(){}
}
```

Is equivalent to:

```csharp
public partial class Foo
{
    void MyHandwrittenFun(){}
    void AnotherFun(){}
    void MyGeneratedFun(){}
}
```
We can use static classes to enforce coding conventions.

• A common pattern is to write a class composed of only static members. (e.g. mathematical operators)
• Whidbey C# uses `static class Foo` to denote that `Foo` cannot have instance methods or members.
• The static modifier is checked at compile time.
• The `Environment.HasShutDownStarted` was (accidentally) an instance variable in the 1.0 framework and users couldn't access it; a static class declaration would have caught this.
Everett projects can’t contain two entities with the same name.

• Generally this a good thing.
  – Readers know which entity is being referred to.
  – C# doesn’t need to define shadowing semantics.

• Sometimes it’s useful to allow conflicting names
  – Different versions of a library can’t be linked together.
  – Common names may exist in different code bases.
Externs let us use libraries with name collisions.

• Compiling with `csc /r:X=foo.dll` roots foo’s namespace hierarchy in alias X.
  – Multiple libraries may be placed in the same alias.
  – The command `csc /r:bar.dll` implicitly adds bar’s members to the global namespace.

• The `extern alias X` command lets a source file treat `X` as a namespace root.
The namespace alias qualifier provides access to aliased names.

- The namespace alias qualifier specifies an which namespace to use to resolve a symbol.
- To access the MyPackage.Math.Add method in the OLD alias use
  
  ```
  OLD::MyPackage.Math.Add(a,b);
  ```
- To access the MyPackage.Math.Add method in global namespace use
  
  ```
  global::MyPackage.Math.Add(a,b);
  ```
  or
  
  ```
  using MP = MyPackage;
  MP::Math.Add(a,b);
  ```
Generics are modern extension to traditional type systems.

- A generic structure is a structure that is parameterized over a type or set of types.
- Generics are a fusion of bounded parametric polymorphism and subtyping.
  - Parametric polymorphism: Type variables
  - Bounded parametric polymorphism: Type variables with constraints
  - Subtyping: Types can be swapped when signatures match
Consider a simple stack based evaluator in Everett...

```java
public class Stack {
    object[] items;
    int count;
    public void Push(object item) {...}
    public object Pop() {...}
}

public class Evaluator {
    public Stack s; //stack of ints
    public int step(Token tok) {
        if (tok.IsInt())
            push (s.Push(tok.toInt()));
        if (tok.IsPlus())
            push ((int)(s.Pop())
                  + (int)(s.Pop()));
        ...
    }
}
```
... and notice how things might break.

- We have two downcasts: `(int)s.Pop()`
- Bad values could wind up in our stack
  - Serialization / deserialization
  - Pre-Processing
- If an other code breaks the stack we will get an exception from Evaluator and not from the responsible method.
Everret C# doesn’t provide good ways to safeguard the stack.

- Wrap `Stack` using a specifically typed class or method set (define `IntStack`)
  - Pollutes name space.
  - Lots of redundant code if done frequently.
- Add lots of runtime checks
  - You can change how gracefully you fail, but you can’t stop from failing.
  - We have better things to do with the CPU than run lots of checks.
With generics we can write code...

```java
public class Stack<T>
{
    T[] items;
    int count;
    public void Push(T item) {...}
    public T Pop() {...}
}

public class Evaluator
{
    public Stack<int> s;
    public int step(Token tok)
    {
        if(tok.IsInt())
        {
            push (s.Pop() + s.Pop());
        }
    }
}
```
...that is safer and faster.

- Now `s.Pop()` returns `int`.
- Member `s` has type `stack<int>`.
  - Code containing `s.Push(x)` won’t compile unless `x` is an `int`.
  - The compiler enforces the class semantics.
- No execution time wasted on casts.
Sorting requires another construct.

- The following won’t compile:
  ```java
  public class SortableArray<T>{
      T[] items;  int count;
      public this[int i] {...}
      public void Sort() {
          ... items[i].compareTo(x) ...}
  }
  ```

- And this is not really safe:
  ```java
  public void Sort() { ...
      ((Comparable)items[i]).compareTo(x) ... } 
  ```
Using type constraints we can statically guarantee sort is safe.

```csharp
class SortableArray<T>
    where T: IComparable
{
    T[] items;
    int count;

    public this[int i] {...}

    public void Sort() {
        ...
        items[i].CompareTo(x)
        ...
    }
}
```

- Type variable T is constrained to implement the IComparable interface.
- Constraints may include
  - any number of interfaces.
  - a base class.
  - a default constructor.
We can gain expressive power by generalizing methods.

- Consider writing a function to merge two arrays:

  ```java
  static void merge<T>(SortedArray<T> x, SortedArray<T> y)
  where T: IComparable {...}
  ```

- Note that we must constrain `T` in order for the `SortedArray<T>` to be a valid type.
- In some cases calls to `merge` may omit the explicit type parameter (type inference).
We can use generic interfaces and multiple type parameter.

```java
public interface Indexable<E>
{
    E[] makeIndex();
}

public class Map<K, V> : Indexable<K>
    where K: System.IComparable
    where V: new()
{
    K item = default(K);
    public Map(){}
    public void Sort(){item.CompareTo(item);}
    public K[] makeIndex(){return null;}
}
```
Modern languages have different levels of support for generics.
Implementation efficiencies vary between languages.
Nullable types provide a principled way of dealing with unknown data.

- Nullable type `int?` represents an integer that can be set to null.
- Nullable primitives correspond well to databases and xml readers.
- Nulls propagate through arithmetic operations.
- Type `foo?` and `System.Nullable<foo>` are completely interchangeable.
Nullable exposes few standard operations.

• Properties & Operations ($n, m$ instances of $T?$)
  – Property $n.HasValue$ is true if $n$ contains a value.
  – Property $n.Value$ is null or the contained value.
  – The null coalescing operator ($??$) is defined such that $n ?? m$ is $n$ if $n.HasValue$, or $m$ otherwise.

• Conversions
  – Values of type $T$ can implicitly convert to $T?$.
  – Values of $T?$ must be explicitly converted to $T$.
  – Values of $T??$ and $T?$ can convert implicitly in both directions.
Nullable can be conceptualized as characteristic of a type.

- Making a nullable type “more nullable” by adding many `?`’s doesn’t effect the derived types’ semantics.
  - So `int?`, `int??`, and `int???` all work the same.
  - This provides flexibility when nulling type variables:
Nullable types behave predictably strangely.

```csharp
int? x = 40;
int y = 2;
int? z = null;

int? t = (int)x + y;
int?? u = 312;
int?? v = z;

Console.WriteLine(t);  // 42
Console.WriteLine(z ?? y); // 2
Console.WriteLine((z + y) == null); // True

Console.WriteLine(u); // 312
Console.WriteLine(u.GetType().ToString());

Console.WriteLine(v.HasValue); // False
```
Corner cases make nullable types hard to work with.

• The compiler emits a warning when declaring nullable ref types (e.g. `string?`).
• Nullable ref types don’t allow for reference equality comparisons (e.g. `==`).
• For consistency with databases, logical operations on `bool?` are ternary. (e.g. `null & false == false`)
Nullable types appear to be one step toward more expressive types.

- Foo – Sometype
- Foo! – Nonnull Foo
- Foo? – Nullable Foo
- Foo* – Possibly empty stream of Foos
- Foo+ - Non-empty stream of Foos
Many languages make it easy to manipulate functions.

- First class functions can be passed around.
- Mixing anonymous and named functions increases code readability.
  
  ```
  (fn x => x+1)       fun f(x) = x + 1
  ```
- Closures provide a convenient way to build predicates.
  
  ```
  let val t = ... in
    filter (fn v => v = t) [1, 2, 3]
  end
  ```
- Lisp, Scheme, SML, Haskell, and Python provide these features.
Everett C# delegates are hard to use.

- All methods needed to be named.
  - Delegate use and definition are often separated by many lines.
  - One time use delegates pollute the local name space and may confuse readers.
- Lack of variable capture means any delegate state needed must be declared as class members.
- Delegate use was too verbose.
Whidbey adds syntactic sugar for manipulating methods.

- Delegate object creation is now implicit.
  
  - Everret:
    ```csharp
    MyEvent += new delegateType(myMethod);
    ```

  - Whidbey:
    ```csharp
    MyEvent += myMethod;
    ```

- Delegates can be anonymous.
  ```csharp
  MyEvent += delegate(int foo)
  {
    return foo+1;
  };
  ```
A logger using an Everett delegate is messy...

```csharp
public delegate void Printer(string s);

public class Logger
{
    Printer logPrinter;
    public Logger(Printer p) { logPrinter = p; }
    public void LogString(string s) { logPrinter(s); }
}

string myPrefix = ">>";

void MyPrinter(string s) { Console.WriteLine(myPrefix + s); }

public void Test()
{
    Logger l = new Logger(new Printer(MyPrinter));
    l.LogString("hello world");
}
```
... but Whidbey helps us clean up.

```csharp
public delegate void Printer(string s);

public class Logger
{
    Printer logPrinter;
    public Logger(Printer p){logPrinter = p;}
    public void LogString(string s){logPrinter(s);}
}

public void Test()
{
    string myPrefix = ">>";
    Logger l = new Logger(delegate (string s)
    {
        Console.WriteLine(myPrefix + s);
    });
    l.LogString("hello world");
}
```
The Whidbey logger is significantly more readable.

- C# is lexical scoping, and, in the Whidbey code, it’s immediately obvious that `myPrefix` will not change during execution.
- Readers of the Whidbey code will not encounter `MyPrinter` at the top level.
- Delegate objects no longer need to be created using `new`. This works for named delegates too!
Useful Links

• CSharp Spec
  http://msdn.microsoft.com/vcsharp/team/language/

• Eric Gunnerson’s Blog
  http://weblogs.asp.net/ericgu/

• Brandon Bray’s Blog
  http://weblogs.asp.net/branbray/

• Interview explaining C# design decisions
  http://www.artima.com/intv/generics.html

• Microsoft Research on types
  http://research.microsoft.com/~emeijer/Papers/XS.pdf