CS3505/5020
Software Practice II

A few C# goodies
Homework Help
Demos (vector math, tools for coding, etc)
Generics

public class List<T>
{
    private T[] elements;
    private int count;

    public void Add(T element)
    {
        if (count == elements.Length) Resize(count * 2);
        elements[count++] = element;
    }

    public T this[int index]
    {
        get { return elements[index]; }
        set { elements[index] = value; }
    }

    public int Count
    {
        get { return count; }
    }
}

List<int> intList = new List<int>();

intList.Add(1); // No boxing
intList.Add(2); // No boxing
intList.Add("Three"); // Compile-time error

int i = intList[0]; // No cast required
Generics

- Note, similar in syntax to C++
  - But much, much different in implementation

- Generics allow us to make classes that use parameterized types
  - Instead of using object (everything is an object)
  - Better type checking (adding a string to a List<int> doesn’t work)
  - No need to box and unbox for primitive types
  - Compile time type checking removes the need for run-time checks
    » This leads to better performance.

- Generics apply to classes, interfaces, structs, delegates and methods
  - You can have generic methods without needing a generic class.
Generics

- Types can be constrained
- The where keyword allows us to state that the type that is bound to the type parameter must have the type of the where statement
  - So, if we need to say that a type has to implement an interface, or extend a class, we can enforce that at compile time
- We can also constrain a type parameter to be a class
  - Or interface, struct, etc.

```csharp
class Dictionary<K,V> : IDictionary<K,V>
    where K: IComparable<K>
    where V: IKeyProvider<K>
{
    public void Add(K key, V value) {
        ...
    }
}
```
Generics

- The most obvious use of generics is in container or collection classes
  - But there are plenty of other uses (IComparable<T> is a good example)

- Generics are not quite the same as template classes in C++
  - Generics are only types. C++ can have primitive types as template parameters.
    - (template class C<typename T, int n>)
  - Rules for generics are simpler than C++
  - C# can try to infer the type of a parameterized type (C++ can not).
    ```
    string[] names = Utils.CreateArray<string>("", 10);
    int[] numbers = Utils.CreateArray<int>(-1, 100);
    ```
    ```
    string[] names = Utils.CreateArray("", 10);
    int[] numbers = Utils.CreateArray(-1, 100);
    ```
Generics and Java 1.5

- Java 1.5 did add generics, but there are some differences
  - Java 1.5 made no changes to the VM. As such...
  - Java 1.5 generics can’t have primitive types as a parameterized type. (no List<int> in Java 1.5)
  - Generic information “erased at compile-time” (no generics in reflection, compiled classes)
  - No gain in performance over “object-based” collections.
Generic Summary

- Generics take a bit to get used to, but it’s not too bad.
  - Take advantage of the new generic collection classes when you can
    » System.Collections.Generic
  - Generic versions in a different namespace than the older versions
    » Backward compatibility for older codes
Partial Classes

- After the complexity of generics, partial classes are easy.
- Partial class just allow you to have the code for one class in multiple files
  - Instead of just one.
- Why is this useful?
Partial Classes

```csharp
public partial class Customer
{
    private int id;
    private string name;
    private string address;
    private List<Orders> orders;
}

public partial class Customer
{
    public void SubmitOrder(Order order) {
        orders.Add(order);
    }

    public bool HasOutstandingOrders() {
        return orders.Count > 0;
    }
}
```
Partial Classes

- Are partial classes good to use?
  - Where are your class methods?

- Some engineering soundness lost for convenience.
  - Use at your own risk.
Aside Delegate References

- In 1.x version of C# we would add to a delegate as:
  - `foo += new delegatename(functionbar);`

- In the 2.x version the compiler is able to “do the right thing”
  - `foo += functionbar;`
Nullable Types

- Sometimes, we need to say that something has no value
  - This is fairly easy with objects, because we can use ‘null’ for no value.
  - But, this is very hard with primitive types.

- For this, we can use nullable types
  - Nullable types are a great match for databases, because many databases use nulls.
  - Nullable types can only be value types (primitive types/structs)

- The syntax is pretty simple
  - `int?` is a nullable integer.
  - `Nullable<T>` makes a nullable type out of any value type (primitive or struct)
    » Will not work with reference types (classes).
Are structs a good idea?

Consider Vector2:
- Vector2 a, b;
- a = new Vector2(3, 20);
- b = a;
- a.Normalize();

What is b?
Iterators

- To move through collections, we need to support an Enumerator
- Let’s look at an example. First, a simple list class.
  - Note we use a ListEnumerator helper class.

```csharp
public class List
{
    internal object[] elements;
    internal int count;

    public IEnumerator GetEnumerator()
    {
        return new ListEnumerator(this);
    }
}
```
public class ListEnumerator : IEnumerator
{
    List list;
    int index;

    internal ListEnumerator(List list)
    {
        this.list = list;
        index = -1;
    }

    public bool MoveNext()
    {
        int i = index + 1;
        if (i >= list.count) return false;
        index = i;
        return true;
    }

    public object Current
    {
        get { return list.elements[index]; }
    }
}
List (New Version)

```csharp
public class List
{
    internal object[] elements;
    internal int count;

    public IEnumerator GetEnumerator()
    {
        for (int i = 0; i < count; i++)
        {
            yield return elements[i];
        }
    }
}
```
Iterators and the Yield Statement

• Okay. How can this possibly work?
  – It looks like we are returning just an item of the list.
    » But we loop over the whole thing.
    » Well, actually, it looks like we just return the first item in the list. The loop doesn’t make sense.
  – But, we have a yield return instead of a plain return.
    » Hum…

• The magic is in the yield.
  – You just write the iterator as if you were walking through all the elements.
    » In this case, we are looping through the array.
  – But, for each value of the container you visit, you make a yield return statement that “returns” that value.
  – The C# compiler does the rest for you.
Iterators

- The yield statement is pretty powerful stuff
  - But once you get it, it’s very, very useful.

- But, what about the magic?
  - Well, what happens is that the compiler makes an internal class for you that does the right thing.
  - For now, just accept the magic and be happy.

- For now, this is limited to creating Enumerators
  - The class generated implements the IEnumerator<T> interface.
  - But, if you take a liberal view of what IEnumerator and GetEnumerator is for, you can do some pretty creative stuff.

```csharp
class Test
{
    public IEnumerator GetEnumerator()
    {
        yield return "hello";
        yield return "world";
    }
}
```

Check this out!!
Java 1.5

- Java 1.5 added some features to Java 1.4
  - Generics (similar to C#)
  - Annotations (somewhat like Attributes in C#)
  - Enumerated Types (different than C#, enums are classes)
  - Boxing and Unboxing (no more casting to/from int to Integer)
  - for/in statement (much like the foreach statement)
- Yes, you could say that a lot of this was “borrowed” from C#
  - But that’s how languages evolve.
Project #2 – Pinball simulation

- Simply a simulation for this part of the project.
  - Project #3 will implement game-like features

- Must use XNA!
  - Use the supplied update/draw paradigm

- Additional C# practice (and XNA warmup)
Project #2 - Engineering

- There are several steps to this project:
  - We will identify these in classroom discussion (and edit this slide)
Project #2

- Automate some programming decisions where appropriate.
  - You have a window, you can draw a sprite at the mouse location. Why would you ever position or locate elements by hand?

- Larger products often require programmers to create their own productivity tools
  - One-off tools are common. Write them, use them and throw them away.
Project #2 - Artwork

- Choose:
  - Coordinate system
  - Sprite organization
  - Drawing / artwork takes time. Plan ahead. (Good artwork takes a lot of time!)

- (Classroom discussion filled out these topics.)
Project #2 - Logic

- How will you organize your sprites (and physical objects)?
- Where will the game logic go?
- Do objects update themselves, or is there another class where positions are changed?
- Ramifications…
Project #2 - Collisions

- Collisions are the heart of this project.
  - Without good collisions, you just have bad code.
  - Devote time to getting these right:
    » Expect several generations of collision code
    » Some vector math required, use Vector2 for support
Project #2 - Collisions

- A few vector routines will help you greatly:
  - Distance from object to point
  - Reflection
    - Elastic
    - Energy adding (bumpers)
    - Energy absorbing (sides, other rigid surfaces)
  - Direction
    - Is the ball moving away from an object?
      - Note: my code makes some assumptions that may be incorrect, but you’ll rarely see them.
Demos

- To finish, I’ll do some vector math and demos.