CS3505/5020
Software Practice II

C#
Vector Review
Homework Help
Decimal primitive

- Use lowercase ‘m’ to denote decimal numbers
- Stored internally as integers with a base 10 decimal point.
- Ideal for money or other base 10 values.
- Decimal point is remembered:

```csharp
Decimal x, y;
x = 4.5m;
y = 4.500m;
Console.WriteLine(x);  // Outputs 4.5
Console.WriteLine(y);  // Outputs 4.500

y = Decimal.Round(y, 2);
Console.WriteLine(y);  // Outputs 4.50
```
Structs

- Similar to classes, but
  - User-defined value type
  - Always inherits from object
  - High performance
  - Stack allocated

- Ideal for lightweight objects
  - int, float, double, etc., are all structs
  - User-defined “primitive” types
    » Complex, point, rectangle, color, rational

- Multiple interface inheritance

- Same members as class

- Member access
  - public, internal, private

- Instantiated with new operator

- Structs are “final” – can’t inherit from them
Allocation: Structs vs. Classes

```c
struct SPoint { int x, y; ... }
class CPoint { int x, y; ... }

SPoint sp = new SPoint(10, 20);
CPoint cp = new CPoint(10, 20);
```
Classes and Structs - Similarities

- Both are user-defined types
- Both can implement multiple interfaces
- Both can contain
  - Data
    » Fields, constants, events, arrays
  - Functions
    » Methods, properties, indexers, operators, constructors
  - Type definitions
    » Classes, structs, enums, interfaces, delegates
public class Car : Vehicle {
    public enum Make { GM, Honda, BMW }
    Make make;
    string vid;
    Point location;
    Car(Make m, string vid; Point loc) {
        this.make = m;
        this.vid = vid;
        this.location = loc;
    }
    public void Drive() {
        Console.WriteLine("vroom");
    }
}

Car c =
    new Car(Car.Make.BMW,
            "JF3559QT98",
            new Point(3,7));
c.Drive();
public struct Point
{
    int x, y;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    public int X { get { return x; } set { x = value; } }
    public int Y { get { return y; } set { y = value; } }
}

Point p = new Point(2, 5);
p.X += 100;
int px = p.X;     // px = 102
** Enums

- Enums are first class
- Enums are typesafe
- Values are optional

```csharp
enum Suit {
    Clubs = 0,
    Diamonds = 1,
    Hearts,
    Spades
}

... Suit s = Suit.Clubs;
Console.WriteLine(s); //-> Clubs
Console.WriteLine((int)s); //-> 0
Suit s2 = Suit.Spades;
Console.WriteLine((int)s2); //-> 3
```
Statements
Variables and Constants

- Within the scope of a variable or constant it is an error to declare another variable or constant with the same name.

```c
{
    int x;
    {
        int x;  // Error: can't hide variable x
    }
}
```
Passing Arguments

- **Java**
  - All by value -- Primitives by value, objects by value of reference
- **C++**
  - By value with copy constructors for objects, reference by specification
- **C#**
  - Primitives by value; Objects by ref; Value can be by ref by saying “ref”
  - “out” – just like “ref” except initial value ignored, and MUST be assigned

```csharp
using System;
public class RefClass {
  public static void Main(string[] args) {
    int total = 20;
    Console.WriteLine("Original value of 'total': {0}", total);
    // Call the Add method
    Add (10, ref total);
    Console.WriteLine("Value after Add() call: {0}", total);
  }
  public static void Add (int i, ref int result) {
    result += i;
  }
}
```
Properties

- Formalized getter/setter model of Java

```java
public class Animal {
    private String name;
    public String Species {
        get { return name; }
        set { name = value; } // Notice magic variable value
    }
}
Animal animal = new Animal();
animal.Species = "Lion"; // Set the property
str = animal.Species; // Get the property value string

- Compiles to get_Species/set_Species for languages that don’t have properties yet
Indexers

- Give array like behavior to any class
- Define property on `this` and add square brackets

```java
public class Skyscraper {
    Story[] stories;
    public Story this [int index] {
        get { return stories[index]; }
        set { if (value!=null) {
            stories[index] = value;
        }
    }
}

SkyScraper searsTower = new SkyScraper();
searsTower[155] = new Story("Observation Deck");
searsTower[0] = new Story("Entrance");
```

Does not HAVE to be an int, but often that is what you want.
Operator Overloading

- Just like in C++

```java
public static complex operator+(complex lhs, complex rhs)
```
try {
    throw new FooException("Oops!");
} catch (FooException e) {
    ... Handle exception ......;
}

} catch {
    ... Catch all other exceptions ...;
}

} finally {
    ... clean up, even if no exception occurred...;
}
Delegates

- C++ and others have function pointers
  - Java does not
- C# does with the delegate
  - A delegate is a reference type that defines a method signature
  - Method signature is both return type and argument types
- When instantiated, a delegate holds one or more methods
  - Essentially an object-oriented function pointer

```csharp
delegate void myDelegate(int a, int b);
myDelegate operation = new myDelegate(Add);
operation += new myDelegate(Multiply);
operation(1,2);
```

Will actually call both the Add AND the Multiply methods
Delegates
Multicast Delegates

- A delegate can hold and invoke multiple methods
  - Multicast delegates must contain only methods that return void, else there is a run-time exception
- Each delegate has an invocation list
  - Methods are invoked sequentially, in the order added
- The += and -= operators are used to add and remove delegates, respectively
- += and -= operators are thread-safe
delegates
Multicast Delegates

delegate void SomeEvent(int x, int y);
static void Foo1(int x, int y) {
    Console.WriteLine("Foo1");
}
static void Foo2(int x, int y) {
    Console.WriteLine("Foo2");
}
public static void Main() {
    SomeEvent func = new SomeEvent(Foo1);
    func += new SomeEvent(Foo2);
    func(1, 2); // Fool and Foo2 are called
    func -= new SomeEvent(Foo1);
    func(2, 3); // Only Foo2 is called
}
Events
Overview

- Event handling is a style of programming where one object notifies another that something of interest has occurred
  - A publish-subscribe programming model
- Events allow you to tie your own code into the functioning of an independently created component
- Events are a type of “callback” mechanism
**Events**

**Overview**

- **Events are well suited for user-interfaces**
  - The user does something (clicks a button, moves a mouse, changes a value, etc.) and the program reacts in response
  - In many systems, the event loop comes with the user interface

- **Many other uses, e.g.**
  - Time-based events
  - Asynchronous operation completed
  - Email message has arrived
  - A web session has begun
**Events Overview**

- C# has native support for events
  - One of the first real languages to do this
- Based upon delegates *(COOL IDEA)*
- An event is essentially a field holding a delegate
- However, public users of the class can only register delegates
  - They can only call `+=` and `-=`
  - They can’t invoke the event’s delegate
- Multicast delegates allow multiple objects to register with the same event
Events
Example: Component-Side

- Define the event signature as a delegate

```csharp
public delegate void EventHandler(object sender, EventArgs e);
```

- Define the event and firing logic

```csharp
public class Button {
    public event EventHandler Click;

    protected void OnClick(EventArgs e) {
        // This is called when button is clicked
        if (Click != null) Click(this, e);
    }
}
```
Events
Example: User-Side

- Define and register an event handler

```csharp
public class MyForm: Form {
    Button okButton;

    static void OkClicked(object sender, EventArgs e) {
        ShowMessage("You pressed the OK button");
    }

    public MyForm() {
        okButton = new Button(...);
        okButton.Caption = "OK";
        okButton.Click += new EventHandler(OkClicked);
    }
}
```
using System;
using System.Threading;
class Test {

    static void printA () {
        while (true) { Console.Write("A");}
    }

    static void printB () {
        while (true) { Console.Write("B");}
    }

    public static void Main () {
        Thread a = new Thread(new ThreadStart(printA));
        Thread b = new Thread(new ThreadStart(printB));
        a.Start(); b.Start();
    }
}
Locks and Critical Sections

Typically `this` to protect instance variable

Statements that you want to run as a critical section

```csharp
public class CheckingAccount {
    decimal balance;
    public void Deposit(decimal amount) {
        lock (this) {
            balance += amount;
        }
    }
    public void Withdraw(decimal amount) {
        lock (this) {
            balance -= amount;
        }
    }
}
```
**XML Comments**

**Overview**

- Java has javadoc
- C# lets you embed XML comments that document types, members, parameters, etc.
  - Denoted with triple slash: ///
- XML document is generated when code is compiled with /doc argument
- Comes with predefined XML schema, but you can add your own tags too
  - Some are verified, e.g. parameters, exceptions, types
# XML Comments Overview

<table>
<thead>
<tr>
<th>XML Tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;summary&gt;</code>, <code>&lt;remarks&gt;</code></td>
<td>Type or member</td>
</tr>
<tr>
<td><code>&lt;param&gt;</code></td>
<td>Method parameter</td>
</tr>
<tr>
<td><code>&lt;returns&gt;</code></td>
<td>Method return value</td>
</tr>
<tr>
<td><code>&lt;exception&gt;</code></td>
<td>Exceptions thrown from method</td>
</tr>
<tr>
<td><code>&lt;example&gt;</code>, <code>&lt;c&gt;</code>, <code>&lt;code&gt;</code></td>
<td>Sample code</td>
</tr>
<tr>
<td><code>&lt;see&gt;</code>, <code>&lt;seealso&gt;</code></td>
<td>Cross references</td>
</tr>
<tr>
<td><code>&lt;value&gt;</code></td>
<td>Property</td>
</tr>
<tr>
<td><code>&lt;paramref&gt;</code></td>
<td>Use of a parameter</td>
</tr>
<tr>
<td><code>&lt;list&gt;</code>, <code>&lt;item&gt;</code>, ...</td>
<td>Formatting hints</td>
</tr>
<tr>
<td><code>&lt;permission&gt;</code></td>
<td>Permission requirements</td>
</tr>
</tbody>
</table>
class XmlElement {
    /// <summary>
    /// Returns the attribute with the given name and namespace
    /// </summary>
    /// <param name="name">The name of the attribute</param>
    /// <param name="ns">The namespace of the attribute, or null if the attribute has no namespace</param>
    /// <return>The attribute value, or null if the attribute does not exist</return>
    /// <seealso cref="GetAttr(string)"/>
    public string GetAttr(string name, string ns) {
        ...
    }
}
Attributes Overview

- It’s often necessary to associate information (metadata) with types and members, e.g.
  - Documentation URL for a class
  - Transaction context for a method
  - XML persistence mapping

- Attributes allow you to decorate a code element (assembly, module, type, member, return value and parameter) with additional information
Attributes
Overview

```csharp
[HelpUrl("http://SomeUrl/APIDocs/SomeClass")]
class SomeClass {
    [Obsolete("Use SomeNewMethod instead")]
    public void SomeOldMethod() {
        ...
    }

    public string Test([SomeAttr()] string param1) {
        ...
    }
}
```
Attributes
Overview

• Attributes are superior to the alternatives
  – Modifying the source language
  – Using external files, e.g., .IDL, .DEF

• Attributes are extensible
  – Attributes allow to you add information not supported by C# itself
  – Not limited to predefined information

• Built into the .NET Framework, so they work across all .NET languages
  – Stored in assembly metadata
# Attributes Overview

- Some predefined .NET Framework attributes

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Browsable</td>
<td>Should a property or event be displayed in the property window</td>
</tr>
<tr>
<td>Serializable</td>
<td>Allows a class or struct to be serialized</td>
</tr>
<tr>
<td>Obsolete</td>
<td>Compiler will complain if target is used</td>
</tr>
<tr>
<td>ProgId</td>
<td>COM Prog ID</td>
</tr>
<tr>
<td>Transaction</td>
<td>Transactional characteristics of a class</td>
</tr>
</tbody>
</table>
Performance

- Lots of performance studies
- One that seems competent is (but this is a couple years old):

- Some numbers:
  - Maximum memory usage:
    - Java - 163 MB
    - C# - 111 MB
    - Cpp - 98 MB

- Performance summary (lots of tests)
  - Cpp is the fastest, except the STL "hashmaps" (11 wins against C#)
  - C# is quit fast (problems with exception handling, matrix multiply and nested loops)
  - Java is slower, but hash maps are very fast and the exception handling is also very strong.
  - C++ 11 wins against C#
  - Java 5 wins against C#
  - C# 9 wins against Java
  - C# 2 wins against C++
Summary

- C# has lots of nice features
- Clearly C# was built after C++ and Java and fixes problems and issues
- C# language feels more like C++
- C# environment is more Java like
- Easy to learn language
- Your work will be in learning the libraries (.Net Framework and XNA libraries)
- More advanced C# includes:
  - Generics
  - Partial classes
  - Anonymous delegates
  - LINQ
  - ...

Summary

- It is easy to be productive in C# without learning all the details
- Good students will exceed my knowledge by the end of the semester (it’s not a C# course).
Vectors

- Represent a direction and magnitude combined.
  - Can be 2D, 3D, or higher – ordered sets of basis magnitudes
  - Usually ordered groups of numbers, but true vectors can be composed of any type
  - Superior to slope and length representation

- 2D Examples:
  - (4, -3)
  - (3, 4)
Vectors

- Can be easily generated from pairs of points
  - Subtract the source from destination

- (4, -3) ->

Points are not vectors?!
- Confusingly, we sometimes treat a point as a displacement from the origin. The vector tells us the relative position of a point.
- Vectors are not associated with a location
Vectors

- Can be combined (added) to form new vectors
  - Vector sum represents the ‘net’ direction or displacement

  - \((4, -3) + (3, 4) = (7, 1)\)

- Can be added to a point to find another point at the specified displacement.
  - \((7, 1) + (12, 5) = (19, 6)\)
Vectors

- Can be scaled to indicate increased displacement

- \((4, -3) \times 2 = (8, -6)\)
Vectors

- Used for many, many graphics and simulation operations:
  - Camera angles
  - Velocities and accelerations
  - Coordinate basis, normals
Vectors

- Can be normalized
  - A normalized vector has a length of 1.0
- Can be easily rotated with a matrix multiply
- Similarities in vector direction are exposed by the ‘dot’ product
  - Aids in changing coordinate spaces
  - Aids in reflections, bounces, intersections, etc.
- More on Thursday
**Discussion Tomorrow**

- Do the discussion tomorrow
  - Next two projects will be building a more capable game than the cat/mouse one that you will build tomorrow
  - Note – if you have any cats (or mice), you may use a *small* GIF/JPG/BMP image in the discussion 😊
  - The notes may take longer than 50 minutes to read through, so use discussion time to resolve questions.