

Game Engineering: 2D

CS420-2010F-02

Intro to CSharp

David Galles

Department of Computer Science
University of San Francisco

02-0: C++ v. Java

- We will be coding in C# for this class
- Java is very similar to C#, with some exceptions:
 - Minor, syntactic differences
 - Performance optimizations in C#
 - Static compilation vs. virtual functions

02-1: Hello World in C#

- Other than output library differences, and Main starting with a capital “M”, C# and Java “Hello World” are the same

```
class Hello
{
    public static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");
    }
}
```

02-2: Compilation Units

- Java:
 - Each class is in its own file
 - Class name == file name
 - Can have main in each class for testing
- C#
 - Many classes in each file
 - Can be named anything
 - Can have Main in each class for testing
 - But, at compile time, need to specify which Main to use

02-3: Compilation Units

Program MyProgram.cs

```
class C1
{
    public static void Main(string args[])
    {
        System.Console.WriteLine("In C1");
    }
}
class C2
{
    public static void Main(string args[])
    {
        System.Console.WriteLine("In C2");
    }
}
```

02-4: Compilation Unit

```
% csc.exe /main:C1 myProgram.cs
```

```
% csc.exe /main:C2 myProgram.cs
```

02-5: Inheritance Syntax

- C# uses C++-style inheritance syntax:

```
class Point
{
    // Body of Point
}

class Circle : Point
{
    // Body of Circle
}
```

02-6: C# Inheritance

- Like Java, single inheritance only
- Also like java, can implement multiple interfaces
 - Most of the gains of multiple inheritance
 - Don't have the downsides of multiple inheritance (what are they?)

02-7: Inheritance Syntax

- Use the same syntax for inheriting and implementing interfaces
- By convention, interface names start with “I”

```
class MySubClass : MySuperClass, IComparable
{
    // Body of Point
}
```

```
class Circle : Point
{
    // Body of Circle
}
```

02-8: Inheritance Syntax

- Advantage of Java syntax over C# syntax for inheritance / implementing interfaces?
- Why does C# use the syntax that it does?

02-9: Namespaces

- You're using a large library of code in your project
- You define a new class "foo"
- The class "foo" already in the library
 - Oops!
- What can you do?

02-10: Namespaces

- You're using a large library of code in your project
- You define a new class "foo"
- The class "foo" already in the library
- What can you do?
 - Create long names for each of your classes
 - Namespaces!

02-11: Namespaces

- Enclose your class in a namespace

```
namespace <name>
{
    // class definition
}
```

02-12: Namespaces

```
namespace Geom {  
  
class Point  
{  
    public Point(float initialX = 0, float initialY = 0) { ... }  
    public float GetX() { ... }  
    public float GetY() { ... }  
    public void SetX(float newX) { ... }  
    public void SetY(float newY) { ... }  
    public void Print();  
  
    private float x;  
    private float y;  
}  
}
```

02-13: Namespaces

- Any class defined within the namespace “foo” can access any other class defined within the same namespace (even from different files)
- Outside the namespace, you can access a class in a different namespace using the syntax `<namespace>.<classname>`

02-14: Namespaces

```
namespace Geom
{

class Rectangle
{
    public Rectangle(float x1, float y1, float x2, float y2);
    public Point GetUpperleft();
    public Point GetLowerRight();

    private Point mUpperLeft;
    private Point mLowerRight;
}

}
```


02-15: Namespaces

```
class Rectangle
{
public:
    Rectangle(float x1, float y1, float x2, float y2);
    Geom.Point GetUpperleft();
    Geom.Point GetLowerRight();

private:
    Geom.Point mUpperLeft;
    Geom.Point mLowerRight;
}
```

02-16: More Namespaces

- Namespaces can nest

```
namespace foo {  
    namespace bar {  
        class MyClass { ... }  
    }  
    class MyClass2  
    {  
        public bar.MyClass c;  
    }  
}  
...  
foo.bar.MyClass x;
```

02-17: “Using” Namespaces

- Many of the standard objects (Object, String) are in the System namespace
 - System.String, System.Object
- Using System. everywhere can get a little cumbersome
- We certainly don't want to put our code in the System namespace!
- using to the rescue

02-18: “Using” Namespaces

```
using System;

class Example
{
    public static void main(String args[])
    {
        Console.WriteLine("Hello World");
    }
}

using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Storage;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;

// Code can use classes/etc defined in each
// nested namespace
```

02-19: Namespaces

- C# Namespaces are very similar to C++ namespaces
- Somewhat similar to Java packages
 - Java packages require mirrored directory structure
 - No such requirement for C# (or C++) namespaces
 - In general, C# (and C++) don't require any relationship between filenames and filepaths (unlike Java)

02-20: Access Directives

- public: Anyone can access
- private: Can only be accessed within class
- protected: Like C++ protected, can be accessed within the class and within subclasses
- internal: Like java protected, can be accessed within the class, and from any class within the same assembly.
- Default == private

02-21: Stack vs. Heap

- Java:
 - Primitives (int, float, boolean) are stored on the stack
 - Complex data structures (arrays, classes) are stored on the heap
- C#
 - Primitives are stored on the stack
 - structs are stored on the stack
 - Classes are stored on the heap

02-22: C# structs

- A C# struct is similar to a class, with a few exceptions:
 - Structs are stored on the stack
 - Default constructor “zeroes out” all fields – can’t override the default constructor (*can* write a constructor that takes arguments)
 - No inheritance for structs
 - *Even though you call new to “create” structs, they are not stored on the heap**. Calling new on a struct just calls the constructor on memory already allocated on the stack

02-23: C# structs

```
struct SPoint
{
    public SPoint(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    public int x;
    public int y;
}

class CPoint
{
    public CPoint(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    public int x;
    public int y;
}
```

02-24: C# structs

- Difference between SPoint and CPoint
 - SPoints are stored on the stack, CPoints stored on the heap
 - SPoints are passed by value (entire structure is copied), CPoints are called by reference (just a pointer is passed)
 - Can't Inherit from SPoint (can from SClass)

02-25: C# structs

```
class Test
{
    static void incrementX(SPoint p)
    {
        p.x++;
    }
    static void incrementX(CPoint p)
    {
        p.x++;
    }

    public static void Main(String args[])
    {
        SPoint p1 = new SPoint(1,1);
        CPoint p2 = new CPoint(1,1);
        increment(p1); increment(p2);
        // Value of p1.x?    p2.x?
    }
}
```

02-26: C# structs

- We can control the exact layout of fields in a C# struct
- Even have fields overlap (C unions)

```
using System.Runtime.InteropServices;
[StructLayout(LayoutKind.Explicit)]
struct TestUnion
{
    [FieldOffset(0)]
    public int intValue;
    [FieldOffset(0)]
    public double doubleValue;
    [FieldOffset(0)]
    public char charValue;
}
```

02-27: C# structs

- We can control the exact layout of fields in a C# struct
- Even have fields overlap (C unions)

```
using System.Runtime.InteropServices;
[StructLayout(LayoutKind.Explicit)]
struct TestUnion
{
    [FieldOffset(0)]
    public int int1Value;
    [FieldOffset(4)]
    public int int2Value
    [FieldOffset(0)]
    public double doubleValue;
    [FieldOffset(8)]
    public char charValue;
}
```

02-28: C# structs

- We can have structs within structs

```
struct Point
{
    public float x;
    public float y;
}
```

```
struct Rectangle
{
    public Point upperLeft;
    public Point lowerRight;
}
```

02-29: C# structs

- The following won't compile – why not?

```
struct LinkedListNode
{
    public LinkedListNode next;
    public int data;
}
```

02-30: Getters and Setters

- Often a bad idea to give direct (public) access to instance variables
- Make instance variables private, create getters and setters to get and set the values
- C# provides some syntactic sugar for getters and setters, called Properties, that *look* like accessing public variables, but behave like method calls

02-31: Properties

```
class GetSetTest
{
    ...
    public float Height
    {
        get
        {
            return mHeight;
        }
        set
        {
            mHeight = value;
        }
    }
    private float mHeight;
}

GetSetTest c = new GetSetTest();
x = c.Height;
c.Height = 3.0f;
```

02-32: Properties With Side Effects

```
class GetSetTest
{
    ...
    public float Height
    {
        get
        {
            return mHeight;
        }
        set
        {
            mHeight = value;
            mNumTimesSet++;
        }
    }
    private float mHeight;
    private mNumTimesSet;
}
```

02-33: Read only Properties

```
class ReadOnlyProperty
{
    ...
    public float Height
    {
        get
        {
            return mHeight;
        }
    }
    private float mHeight;
}
```

02-34: Write only Properties

```
class WriteOnlyProperty
{
    ...
    public float Height
    {
        set
        {
            mHeight = value;
        }
    }
    private float mHeight;
}
```

02-35: Auto-Implemented Properties

- Properties that just access variables (without side effects) are pretty verbose for what they do
- C# allows for autogenerated properties – you give the name of the property, and the variable is created and manipulated behind the scenes

```
class AutoGenProperty
{
    ...
    public float Height {get; set; }
}
```

02-36: Virtual Functions

- In Java, all methods are virtual
 - Every method call requires extra dereference
 - Always get the correct method
- In C#, methods are, by default, static
 - Determine at *compile time* which code to call
 - Advantages? Disadvantages?

02-37: Virtual Functions

```
class Base
{
    public void p1() { printf("p1 in Base\n");}
    public virtual void p2() { printf("p2 in Base\n");}
}

class Subclass : public Base
{
    public void p1() { printf("p1 in Subclass\n");}
    public override void p2() { printf("p2 in Subclass\n");}
}

// Some later block of code:
{
    Base b1 = new Base();
    Subclass s1 = new Subclass();
    Base b2 = s1;
    b1->p1();    b1->p2();
    b2->p1();    b2->p2();
    s1->p1();    s1->p2();
}
```

02-38: Pass by Reference

- C# allows you to pass a parameter by *reference*
- Actually pass a pointer to the object, instead of the object itself

02-39: Pass by Reference

```
void foo(int x, ref int y)
{
    x++;
    y++;
}
```

```
public static void Main(String args[])
{
    int a = 3;
    int b = 4;
    foo(a, ref b);
    Console.WriteLine("a = " + a + " b = " + b)
}
```

Output:

a = 3, b = 5

02-40: C# Generics

- C# are nearly identical to Java generics from a user point of view
- Also very similar to C++ templates
 - Implementation is different, (type erasure vs. code copying vs mixture), but that's beyond the scope of this class
- Stack.cs example

02-41: C# Enumerators (Iterators)

- C# iterators are similar to Python iterators
- foreach statement, iterate over anything that implements System.Collections.IEnumerable
- Standard collections (including built-in arrays) all implement IEnumerable

```
int[] MyArray = {1, 2, 3, 4, 5};  
foreach (int i in MyArray)  
{  
    Console.WriteLine(i.ToString());  
}
```

02-42: C# Enumerators (Iterators)

- The “foreach” construct is just a bit of syntactic sugar for a class that implements a IEnumerable interface
 - IEnumerable interface defines GetEnumerator method that returns an object that implements IEnumerator interface
 - IEnumerator defines Current, MoveNext, Reset

02-43: C# Enumerators (Iterators)

```
List<int> myList = new List<int>();  
myList.Add(1); myList.Add(2); myList.Add(3);  
IEnumerator<int> itr = myList.GetEnumerator();  
while(itr.MoveNext())  
{  
    Console.WriteLine(itr.Current);  
}
```

02-44: C# Enumerators (Iterators)

- To make your collection iterable, needs to implement the IEnumerable interface
 - `Systems.Collections.IEnumerator`
`GetEnumerator()`
- Interface `Systems.Collections.IEnumerator`:
 - `object Current` get;
 - `bool MoveNext()`;
 - `void Reset()`;
- Pretty much how Java does iterators

02-45: Writing C# Enumerables

- Better way to write Enumerators:
 - GetEnumerator contains a loop that goes through each element in the collection
 - call “yield return” on each element

02-46: Writing C# Enumerables

```
using System.Collections;
class EnumTest : IEnumerable
{
    // Constructor, etc
    int[] data;

    IEnumerator GetEnumerator()
    {
        for (int i = 0; i < data.Length; i++)
        {
            yield return data[i];
        }
    }
}
```


02-47: Writing C# Enumerables

```
using System.Collections;
class EnumTest : IEnumerable
{
    // Constructor, etc
    int[] data1;
    int[] data2;

    IEnumerator GetEnumerator()
    {
        foreach (int elem in data1)
        {
            yield return elem;
        }
        foreach (int elem in data2)
        {
            yield return elem;
        }
    }
}
```

02-48: Writing C# Enumerables

- Binary Search Tree example

02-49: Unsafe Code

- Hard to write OS code in Java – don't get direct access to memory
- Easier in C# – can use *unsafe* code
 - C-style pointers
 - Most run-time checks disabled
 - Full Control to shoot yourself in the foot
 - *Only within code blocks marked as unsafe*

02-50: Unsafe Code

- Within unsafe code you can play with C style pointers
- Take address of object
 - Need to use fixed construct for address of elements on the heap
- Example: `Unsafe.cs`

02-51: Unsafe Code

- All sorts of fun things you can do with unsafe:

```
static void Main(string[] args)
{
    int* fib = stackalloc int[100];
    int* p = fib;
    *p++ = *p++ = 1;
    for (int i=2; i<100; ++i, ++p)
        *p = p[-1] + p[-2];
    for (int i=0; i<10; ++i)
        Console.WriteLine (fib[i]);
}
```

02-52: **Static Classes**

- If a class is declared “static”:
 - Can only contain static members
 - Cannot be instantiated
 - Cannot be subclassed
- Useful for creating libraries of functions, like Math

02-53: Static Classes

```
static class Math
{
    const double pi = 3.1415926535
    static double sin(double theta) { ... }
    static double cos(double theta) { ... }
    ...
}
```

02-54: Operator Overloading

- Let's say you are writing a complex number class
 - Want standard operations: addition, subtraction, etc
 - Write methods for each operation that you want
- It would be nice to use built-in operators

```
Complex c1 = new Complex(1,2);  
Complex c2 = new Complex(3,4);  
Complex c3 = c1 + c2;
```


02-55: Operator Overloading

- In C# you can overload operators
- Essentially just “syntactic sugar”
- Really handy for things like vector & matrix math
 - math math libraries make heavy use of operator overloading
- See C# Complex code example
- Aside: Why no operator overloading in Java?

02-56: Constants

- Any instance variable that is declared const needs to be given a value at compile time
- const variables are implicitly static
- Any instance variable that is declared readonly needs to be initialized in the constructor, can't be changed

```
class Constants
{
    const float pi = 3.14159f; // Implicitly static
    static readonly float cake = 10;
    readonly startTime;
    public Constants()
    {
        startTime = DateTime.Now.Millisecond;
    }
}
```

02-57: #region

- Can denote code blocks that you want to be able to collapse and expand in Visual Studio using #region
- Not terribly useful (and if you are using #region heavily, might be time to refactor your code!), but inserted in some generated code

```
#region helperFunctions
// Number of helper functions you want to be able to hide
#endregion
```

- Examples in visual studio

02-58: Delegates

- C# version of function pointers
- Used in (among other places) event driven programming

02-59: Delegates

```
class DelegateTest
{
    delegate int inc(int x);

    static int testDel(delegate inc, int val) {
        return inc(val);
    }
    static int addOne(int x) {
        return x + 1;
    }
    static int addTwo(int x) {
        return x + 2;
    }
    static void Main(string[] args) {
        console.WriteLine(testDel(new inc(addOne), 3));
        console.WriteLine(testDel(new inc(addTwo), 3));
    }
}
```

02-60: Delegates

- From website:
 - TestDelegate.cs
 - TestDelegate2.cs
- Delegates are not just function pointers – store the entire context of function being called
 - Can pass non-static methods as delegates (C++ requires static)
 - More memory overhead than C++ function pointers

02-61: **Parallel constructs**

- Come back to parallel constructs when we cover parallel programming, later in the semester

02-62: Odds & Ends

- Other minor differences between Java and C#
- If you come across an unfamiliar concept / keyword, google is your friend
- Example: sealed