

ECE4893A/CS4803MPG: MULTICORE AND GPU PROGRAMMING FOR VIDEO GAMES



Introduction to C#



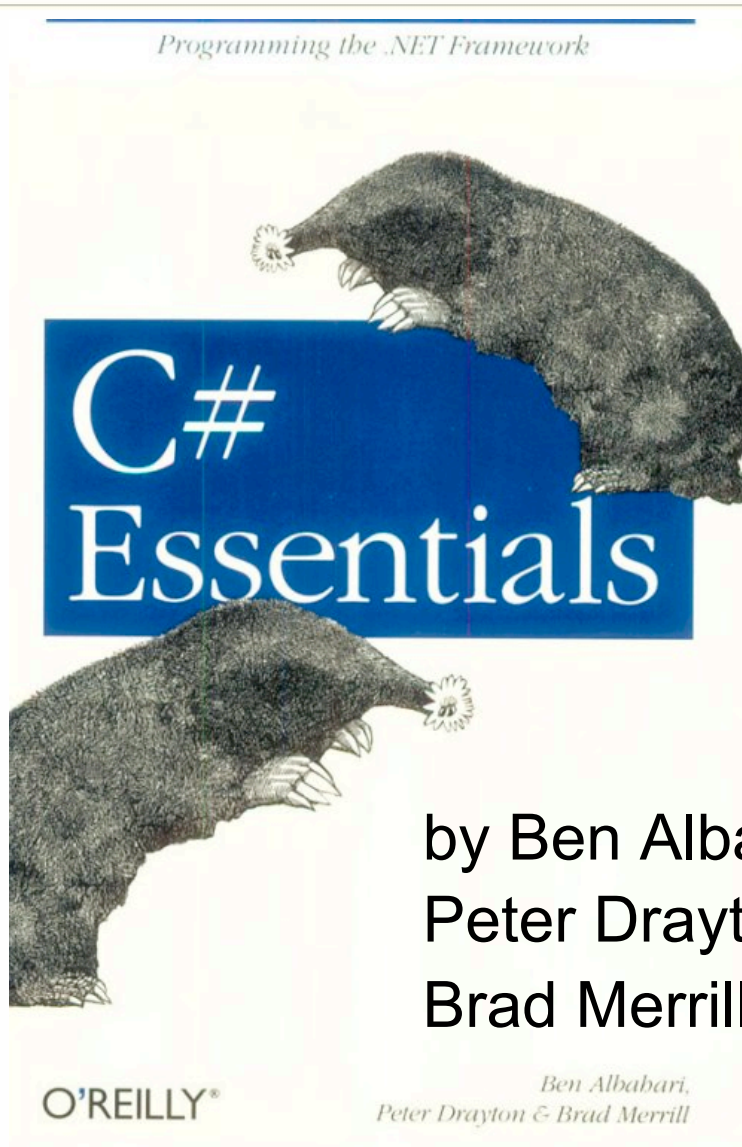
Prof. Aaron Lanterman

School of Electrical and Computer Engineering

Georgia Institute of Technology



References



Great article:

Jesse Liberty, “Top ten traps in C# for C++ programmers”

www.ondotnet.com/pub/a/oreilly/dotnet/news/programmingCsharp_0801.html

Great slide set:

“Introduction to C# slides” from Jim Whitehead’s “Game Design Experience”

Classes and objects

- A class combines together
 - Data
 - Class variables
 - Behavior
 - Methods
- A key feature of object-oriented languages
 - Procedural languages, such as C, did not require clustering of data and behavior

From Jim Whitehead's "Introduction to C# slides"
from his "Game Design Experience" class
Creative Commons Attribution 3.0

Class/instance distinction

- Class defines variables & methods
- Need to create instances of the class, called objects, to use variables & methods
- Exception: static methods and variables

Hello World example

```
class Hello
{
    static void Main()
    {
        // Use the system console object
        System.Console.WriteLine("Hello, World!");
    }
}
```

From Jim Whitehead's "Introduction to C# slides"
from his "Game Design Experience" class
Creative Commons Attribution 3.0

Key language features

- “Unified” object system
 - Every type is an “object,” even primitives
- Single inheritance
- Interfaces
 - Specify methods & interfaces, but no implementation
- Structs
 - A restricted, lightweight (efficient) type
- Delegates
 - Expressive typesafe function pointer
 - Useful for strategy and observer design patterns

From Jim Whitehead’s “Introduction to C# slides”
from his “Game Design Experience” class
Creative Commons Attribution 3.0

Defining a class

[attributes] [access-modifiers] class identifier [:base-class [,interface(s)]] { class-body }

```
class A
{
    int num = 0;    // a simple variable

    A(int initial_num) { num = initial_num; }
                      // set initial value of num
}
```

- Attributes: used to add metadata to a class (can be ignored)
- Access modifiers: one of
 - **public, private, protected, internal, protected internal**
- Base-class
 - Indicates (optional) parent for inheritance
- Interfaces
 - Indicates (optional) interfaces that supply method signatures that need to be implemented in the class
- Class-body
 - Code for the variables and methods of the class

From Jim Whitehead's "Introduction to C# slides"
from his "Game Design Experience" class
Creative Commons Attribution 3.0

Inheritance

- Operationally
 - If class B inherits from base class A, it gains all of the variables and methods of A
 - Class B can optionally add more variables and methods
 - Class B can optionally change the methods of A
- Uses
 - Reuse of class by specializing it for a specific context
 - Extending a general class for more specific uses
- Interfaces
 - Allow reuse of method definitions of interface
 - Subclass must implement method definitions

From Jim Whitehead's "Introduction to C# slides"
from his "Game Design Experience" class
Creative Commons Attribution 3.0



Georgia Institute of Technology

cking, Flickr
www.flickr.com/photos/spotsgot/1500855/

Inheritance example

```
class A {  
    public void display_one() {  
        System.Console.WriteLine("From A");  
    }  
}  
  
class B : A {  
    public void display_two() {  
        System.Console.WriteLine("From B, child of A");  
    }  
}  
  
class App {  
    static void Main() {  
        A a = new A();           // Create instance of A  
        B b = new B();           // Create instance of B  
  
        a.display_one();          // I come from A  
        b.display_one();          // I come from A  
        b.display_two();          // I come from B, child of A  
    }  
}
```

From Jim Whitehead's "Introduction to C# slides"
from his "Game Design Experience" class
Creative Commons Attribution 3.0

Visibility

- A class is a container for data and behavior
- Often want to control over which code:
 - Can read & write data
 - Can call methods
- Access modifiers:
 - Public
 - No restrictions; members visible to any method of any class
 - Private
 - Members in class A marked private only accessible to methods of class A
 - Default visibility of class variables (but is good to state this explicitly)
 - Protected
 - Members in class A marked protected accessible to methods of class A and subclasses of A

From Jim Whitehead's "Introduction to C# slides"
from his "Game Design Experience" class
Creative Commons Attribution 3.0

Visibility example

```
class A {  
    public int  
        num_slugs;  
    protected int  
        num_trees;  
    ...  
}  
  
class B : A {  
    private int  
        num_tree_sitters;  
    ...  
}  
  
class C { ... }
```

- Class A can see:
 - num_slugs: is public
 - num_trees: is protected, but is defined in A
- Class B can see:
 - num_slugs: is public in A
 - num_trees: is protected in parent A
 - num_tree_sitters: is private, but is defined in B
- Class C can see:
 - num_slugs: is public in A
 - Can't see:
 - num_trees: protected in A
 - num_tree_sitters: private in B

From Jim Whitehead's "Introduction to C# slides"
from his "Game Design Experience" class
Creative Commons Attribution 3.0

Constructors

- Use “new” to create a new object instance
 - This causes the “constructor” to be called
- A constructor is a method called when an object is created
 - C# provides a default constructor for every class
 - Creates object but takes no other action
 - Typically classes have explicitly provided constructor
- Constructor
 - Has same name as the class
 - Can take arguments
 - Usually public, though not always
 - Singleton design pattern makes constructor private to ensure only one object instance is created

Type system (1)

- Value types: Directly contain data
 - Intrinsic types and structs
 - “Passed by value” (copied)
 - Cannot be null
 - Allocated on the stack (unless part of a reference type)
- Reference types: Contain references to objects
 - Classes and interfaces, and “boxed” value types
 - “passed by reference” (implicit pointer)
 - May be null
 - Variables sit on the stack, but hold a pointer to an address on the heap; the “real object” is allocated on heap

Slide adapted from “Introduction to C#”, Anders Hejlsberg

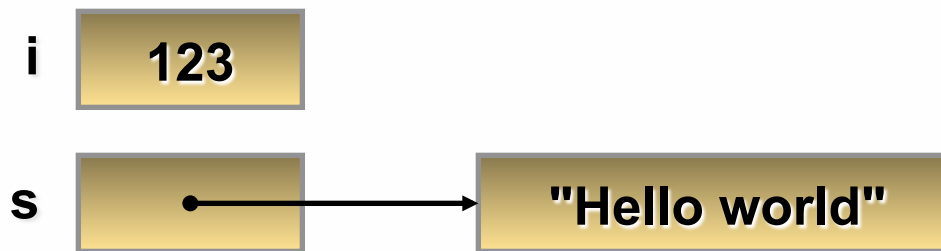
www.ecma-international.org/activities/Languages/Introduction%20to%20Csharp.ppt

From Jim Whitehead’s “Introduction to C# slides”
from his “Game Design Experience” class
Creative Commons Attribution 3.0



Type system (2)

```
int i = 123;  
string s = "Hello world";
```



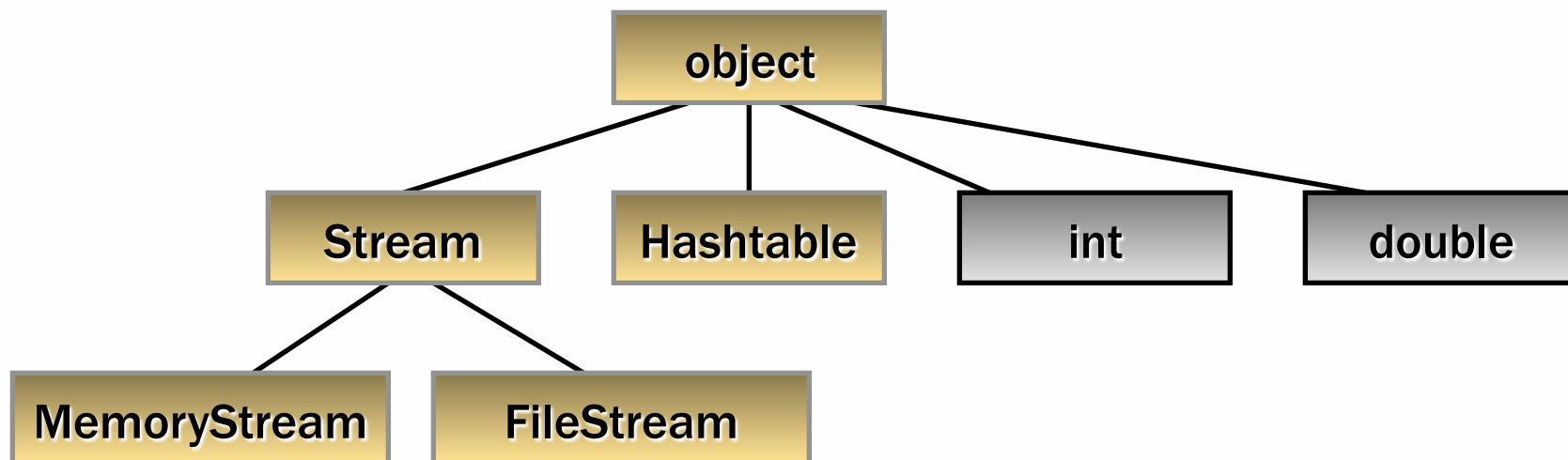
Slide adapted from “Introduction to C#”, Anders Hejlsberg

www.ecma-international.org/activities/Languages/Introduction%20to%20Csharp.ppt

From Jim Whitehead’s “Introduction to C# slides”
from his “Game Design Experience” class
Creative Commons Attribution 3.0

Unified type system

- All types ultimately inherit from **object**
 - **classes, enums, arrays, delegates, structs, ...**
- An implicit conversion exists from *any* type to type **object**

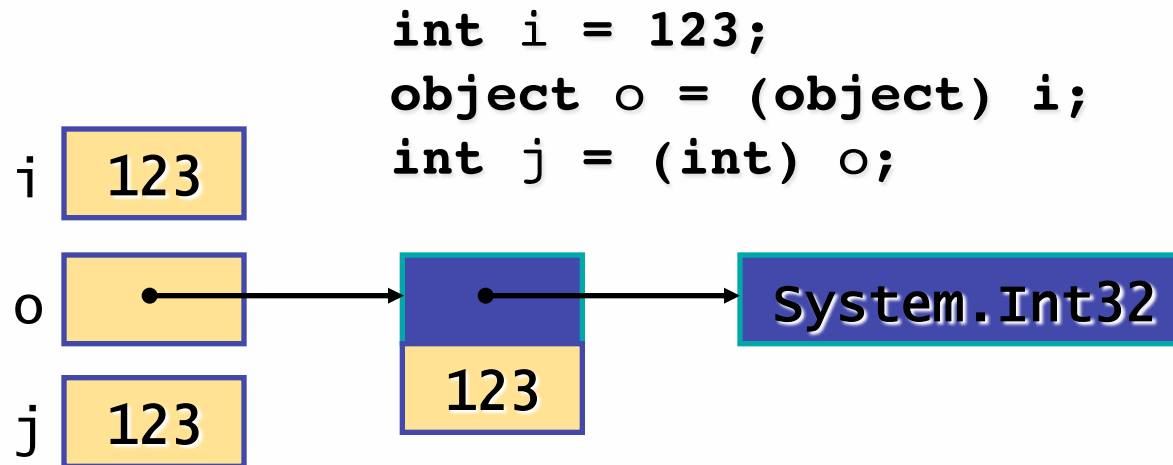


From Jim Whitehead's "Introduction to C# slides"
from his "Game Design Experience" class
Creative Commons Attribution 3.0

Slide from "Introduction to C#",
Anders Hejlsberg

Boxing and unboxing

- **Boxing**
 - Process of converting a value type to the type **object**
 - Wraps value inside a **System.Object** and stores it on the managed heap
 - Can think of this as allocating a “box”, then copying the value into it
- **Unboxing**
 - Extracts the value type from the object
 - Checks type of box, copies value out



From Jim Whitehead's "Introduction to C# slides"
from his "Game Design Experience" class
Creative Commons Attribution 3.0

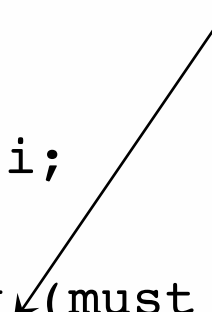
Boxing and unboxing example

```
using System;
public class UnboxingTest
{
    public static void Main()
    {
        int i = 123;

        //Boxing
        object o = i;

        // unboxing (must be explicit)
        int j = (int) o;
        Console.WriteLine("j: {0}", j);
    }
}
```

If o is null or not an int an
InvalidCastException is thrown



Predefined types

- C# predefined types
 - Reference **object, string**
 - Signed **sbyte, short, int, long**
 - Unsigned **byte, ushort, uint, ulong**
 - Character **char** (2 byte, Unicode)
 - Floating point **float, double, decimal**
 - Logical **bool**
- Predefined types are simply aliases for system-provided types
 - For example, **int == System.Int32**

From Jim Whitehead's "Introduction to C# slides"
from his "Game Design Experience" class
Creative Commons Attribution 3.0

Slide from "Introduction to C#",

Anders Hejlsberg



www.ecma-international.org/activities/Languages/Introduction%20to%20Csharp.ppt

Pop quiz: C

- What is the value of b after this code is run (assume C code)?

```
a = 7;  
b = 3;  
if (a = 5)  
{  
    b = 10;  
}
```

Booleans in C#

- In C, 0 is false, “anything else” is true
- In C#, this code will give a compile time error
 - C# has distinct Boolean values, true and false

```
a = 7;  
b = 3;  
if (a = 5)  
{  
    b = 10;  
}
```

Enumerations (1)

```
enum Grades
{
    gradeA = 94,
    gradeAminus = 90,
    gradeBplus = 87,
    gradeB = 84
}
```

- Base type can be any integral type (**ushort**, **long**) except for **char**
- Defaults to **int**
- Must cast to **int** to display in **WriteIn**
 - Example: **(int)g.gradeA**

From Jim Whitehead's "Introduction to C# slides"
from his "Game Design Experience" class
Creative Commons Attribution 3.0

Enumerations (2)

- Defaults to start at zero

```
enum Days {Sat, Sun, Mon, Tue, Wed,  
           Thu, Fri};
```

- Can override initial value

```
enum Days {Sat=1, Sun, Mon, Tue, Wed,  
           Thu, Fri};
```

Decimal type

- A fixed precision number up to 28 digits plus decimal point
- Useful for money calculations
- 300.5m
- Suffix “m” or “M” indicates decimal



tackyspoons, Flickr
www.flickr.com/photos/tackyspoons/812710409/

From Jim Whitehead's "Introduction to C# slides"
from his "Game Design Experience" class
Creative Commons Attribution 3.0

Variables

```
int remaining = 0;  
string name;  
float myfloat = 0.5f;  
bool zombified = true;
```

```
const int freezingPoint = 32;
```

- Variables must be initialized or assigned to before first use
- Class members take a visibility operator beforehand (private by default)
- Constants cannot be changed

Structs vs. classes

- Structs are value types
 - More efficient when used in arrays
 - Less efficient when used in collections
 - Collections expect reference types, so structs must be “boxed” - boxing has overhead
 - Support properties, methods, fields, and operators...
 - ...but not inheritance or destructions
- Classes are reference types
 - May be more efficient when used in collections

Reference parameters

- C, C++, and C# allow a function to only return one value
- In C++ and C, you can get around this by passing in pointers
- In C#:
 - Reference types in the parameter list may be changed by the function
 - To let a function change a value type in the parameter list, can use an explicit `ref` keyword:

`ref` must be
used in both
declaration
and call

`public void Changer(ref int x)`

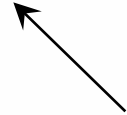
`Aaron.Changer(ref int aaronx);`

Variables must be initialized

```
public void Changer(ref int x)
```

```
int aaronx;
```

```
Aaron.Changer(ref int aaronx);
```



C# will give a compile-time error
since aaronx has not been
initialized

In general, variables in C# must be assigned
before being passed into a function

A clunky workaround

```
public void Changer(ref int x)
```

```
int aaronx = 0;
```

```
Aaron.Changer(ref int aaronx);
```

The out keyword

```
public void Changer(out int x)
```

```
int aaronx;
```

```
Aaron.Changer(out int aaronx);
```



out keyword like **ref**, except it tells C# that it's OK for the value to be undefined

C# will demand that you assign `aaronx` before the function returns!

C# Finalizers (1)

```
~MyClass()  
{  
    // your code to release unmanaged resources  
    // used by object  
}
```

is syntactic sugar for

```
MyClass.Finalize()  
{  
    // your code to release unmanaged  
    // resources used by object  
    base.Finalize();  
}
```

Your finalizer should not try to deal with other C# reference objects - only deal with unmanaged resources!

C# Finalizers (2)

- Finalizer will be called when the .NET garbage collector decides to call it
 - You don't get to decide when it's called
- Only define a finalizer if you really need one
 - Calling it involves some overhead

C# arrays are objects

Java: **int** arr1[];

C#: **int**[] arr1;

arr1 = **new int**[5];

arr1 = **new int**[5]{10,20,30,40,50};

int[] arr2 = **new int**[5] {10,20,30,40,50};

int[] arr2 = {10,20,30,40,50};

Multi-dimensional arrays

```
string[,] bingo;
```

```
bingo = new string[3,2]  {{“A”,“B”},  
    {“C”,“D”},{“E”,“F”}};
```

```
bingo = new string[,]  {{“A”,“B”},  
    {“C”,“D”},{“E”,“F”}};
```

```
string[,] bingo = {{“A”,“B”},{“C”,“D”},  
    {“E”,“F”}};
```

Jagged arrays

- Arrays of arrays

```
int[] arr =  
new int[] [  
    {new int[] {10,11,12}, new int[] {13, 14,  
    15, 16, 17}}];
```

Array iteration

```
int[] arr = {16, 17, 18};  
foreach (int x in arr)  
{  
    System.Console.WriteLine(x.ToString());  
}
```

- Works on arrays and collections
- List is read-only in the loop
- Can't change x in the loop

foreach details

“In C#, it is not strictly necessary for a collection class to inherit from **IEnumerable** and **IEnumerator** in order to be compatible with **foreach**; as long as the class has the required **GetEnumerator**, **MoveNext**, **Reset**, and **Current** members, it will work with **foreach**.”

Switch statement

```
const int raining = 1;
const int snowing = 0;
int weather = snowing;
switch (weather) {
    case snowing:
        System.Console.WriteLine("It is snowing!");
        goto case raining;
    case raining:
        System.Console.WriteLine("I am wet!");
        break;
    default:
        System.Console.WriteLine("Weather OK");
        break;
}
```

- Alternative to **if**
- Typically use **break**
- Can use **goto** to continue to another case