CS3505/5020
Software Practice II

Game loops
C#
HW 1 – Game Loop

- How do games work?
  - sprites move across screen
  - fast enough so you don’t see flicker

- Game loop controls this
  - it can either draw the sprites as fast as it can
  - or it draws them on a set period of time (like once every 1/60th of a second)

- Advantages and disadvantages of both

- Mapping into a C# form is tricky
  - Three solutions: timer, thread, invalidate paint method
  - You can choose any of them
Contrast Timed vs. Infinite

- Timed is periodic, Infinite is ‘as fast as you can’
- If you want to move something from point a to point b, how do you do it in either one?
  - What else do you need to know?
- What if you just have a velocity?
  - What is velocity defined as?

- Understanding this is a key insight for this assignment
Key steps in this assignment

- Choose your visual representations, set up the form
- Respond to user events (enable appropriate form elements)
- Create ball class, set up objects
- Implement game loop
- Add physics / motion

- Incremental development is paramount
The C# Language

- Brief overview of key concepts and differences with Java and C++
- Some cool new features
Design Goals of C#

The Big Ideas

- Component-orientation
  - Events, methods, properties
- Everything is an object
  - Yes, this is different from Java and C++ and is POWERFUL
- Robust and durable software
- Preserving your investment
  - Easy to integrate with lots of “stuff”
Design Goals of C#
Component-Orientation

- C# is the first “Component-Oriented” language in the C/C++ family
- What is a component?
  - An independent module of reuse and deployment
  - Coarser-grained than objects
    (objects are language-level constructs)
  - Includes multiple classes
  - Often language-independent
  - In general, component writer and user don’t know each other, don’t work for the same company, and don’t use the same language
Design Goals of C#
Component-Orientation

- Component concepts are first class
  - Properties, methods, events
  - Design-time and run-time attributes
  - Integrated documentation using XML

- Enables “one-stop programming”
  - In some languages, header files or IDL are used to describe component interfaces
  - In C#, this is built in

- Thus, no .h/.cpp split
  - No forward declarations required

- Also, no globals. (Yay)
Design Goals of C#
Everything is an Object

- Traditional views
  - C++, Java™: Primitive types are “magic” and do not interoperate with objects (but perform better – no heap allocation)
  - Smalltalk, Lisp: Primitive types are objects, but at some performance cost

- C# unifies with minimal performance cost
  - Deep simplicity throughout system

- Improved extensibility and reusability
  - New primitive types: Decimal, …
  - Collections, etc., work for all types
Design Goals of C#  
Robust and Durable Software

- Garbage collection
  - No memory leaks and stray pointers
- Exceptions
- Type-safety
  - No uninitialized variables, no unsafe casts
- Versioning addressed
- Avoid common errors
  - E.g. if (x = y) ...
- One-stop programming
  - Fewer moving parts
  - See version 3.0 features!!
Design Goals of C#
Preserving Your Investment

- **C++ Heritage**
  - Namespaces, pointers (in unsafe code), unsigned types, etc.
  - Some changes, but no unnecessary sacrifices
  - Feels more like C++ than Java in terms of language, but feels more like Java in terms of environment

- **Interoperability**
  - What software is increasingly about
  - C# talks to XML, SOAP, COM, DLLs, and any .NET Framework language

- **Increased productivity**
  - Short learning curve
  - Millions of lines of C# code in .NET
Program Structure

- Physical organization
  - Types are defined in files (each type in 1 or more files – partial classes)
    » Partial classes!!
  - Files are compiled into modules
  - Modules are grouped into assemblies
  - Assemblies usually 1 to 1 with modules
Files, Preprocessor, Namespace

- **C++ style comments** (// and /* */)
- **Files in C++ and Java**
  - C++: header and cpp file, multiple classes per file allowed
  - Java: one file with one class per file and file named same class
- **C#**
  - One file (.cs extension)
  - Any number of class definitions per file allowed
- **C# has C++ preprocessor concept (#define, etc.)**
- **Java uses the package concept, C# uses namespace**
  - Allow multiple namespaces per file
- **Also have alias: using foo = namespace.namespace.class;**
- **C# always uses . ; no more . or -> or ::**

```java
package <Package name>;
import <package hierarchy>.<class name>;
class Customer {
    ...
}
```

```csharp
using <namespace hierarchy>.<class name>;
namespace <namespace name> {
    class Customer {
        ...
    }
}
```
Types
Unified Type System

- **Value types**
  - Directly contain data
  - Cannot be null

- **Reference types**
  - Contain references to objects
  - May be null

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

Unified Type System

- **Value types**
  - Primitives
    ```
    int i; float x;
    ```
  - Enums
    ```
    enum State { Off, On }
    ```
  - Structs
    ```
    struct Point {int x,y;}
    ```
  - Unsigned ints are included for writing systems code (not in Java)

- **Reference types**
  - Root
    ```
    object
    ```
  - String
    ```
    string
    ```
  - Classes
    ```
    class Foo: Bar, IFoo {...}
    ```
  - Interfaces
    ```
    interface IFoo: IBar {...}
    ```
  - Arrays
    ```
    string[] a = new string[10];
    ```
  - Delegates
    ```
    delegate void Empty();
    ```
# Types

## Unified Type System

<table>
<thead>
<tr>
<th></th>
<th>Value (Struct)</th>
<th>Reference (Class)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable holds</td>
<td>Actual value</td>
<td>Memory location</td>
</tr>
<tr>
<td>Allocated on</td>
<td>Stack, member</td>
<td>Heap</td>
</tr>
<tr>
<td>Nullability</td>
<td>Always has value</td>
<td>May be null</td>
</tr>
<tr>
<td>Default value</td>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>Aliasing (in a scope)</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Assignment means</td>
<td>Copy data</td>
<td>Copy reference</td>
</tr>
</tbody>
</table>
Types
Unified Type System

- Benefits of value types
  - No heap allocation, less GC pressure
  - More efficient use of memory
  - Less reference indirection
  - Unified type system
    » No primitive/object dichotomy
    » THIS IS A BIG DEAL!!!!
**Types**

**Conversions**

- **Implicit conversions**
  - Occur automatically
  - Guaranteed to succeed
  - No information (precision) loss

- **Explicit conversions**
  - Require a cast
  - May not succeed
  - Information (precision) might be lost

- **Both implicit and explicit conversions can be user-defined**

```java
int x = 123456;
long y = x;       // implicit
short z = (short)x;  // explicit

double d = 1.2345678901234;
float f = (float)d;  // explicit
long l = (long)d;   // explicit
```
Types
Unified Type System

- Everything is an object
  - All types ultimately inherit from object
  - Any piece of data can be stored, transported, and manipulated with no extra work
Types
Unified Type System

- Polymorphism
  - The ability to perform an operation on an object without knowing the precise type of the object
  - Notice use of primitive types too!! (Java added in 1.5)

```csharp
void Poly(object o) {
    Console.WriteLine(o.ToString());
}
```

Poly(42);
Poly("abcd");
Poly(12.345678901234m);
Poly(new Point(23,45));
Types
Unified Type System

● Question: How can we treat value and reference types polymorphically?
  – How does an int (value type) get converted into an object (reference type)?

● Answer: Boxing!
  – Boxing – take something and wrap it up and put it into a Box!!! (objectification 😊)
  – A key innovation of C#
  – Only value types get boxed
  – Reference types do not get boxed
Types
Unified Type System

- Boxing
  - Copies a value type into a reference type (object)
  - Each value type has corresponding “hidden” reference type
  - Note that a reference-type copy is made of the value type
    » Value types are never aliased
  - Value type is converted implicitly to object, a reference type
    » Essentially an “up cast”
    » So you can do things like toString on an int
Types
Unified Type System

- **Unboxing**
  - Inverse operation of boxing
  - Copies the value out of the box
    » Copies from reference type to value type
  - Requires an explicit conversion
    » May not succeed (like all explicit conversions)
    » Essentially a “down cast”
Types
Unified Type System

- Boxing and unboxing

```csharp
int i = 123;
object o = i;
int j = (int)o;
```

![Diagram showing boxing and unboxing](image)
**Types**

*Unified Type System*

- **Benefits of boxing**
  - Enables polymorphism across all types
  - Collection classes work with all types
    » Oh, yeah, this is HUGELY cool!!!
  - Eliminates need for wrapper classes

- **Disadvantages of boxing**
  - Some performance cost doing the conversion
Classes

- The class concept is pretty much the same as for Java and C++ and C#
- Some syntactic differences
  - Like how you specify inheritance
  - C++ permits multiple inheritance
  - C# has destructors (Java doesn’t)
- Other differences we will see as we go along
**Class Access Control**

- **public** – visible to all
- **protected** – visible only from derived classes
- **private** – visible only within the given class (default, where in Java default is internal)
- **internal** – visible only within the same assembly
- **protected internal** – visible only to the current assembly or types derived from the containing class
- **sealed** - can’t be inherited (like final in Java) – structs are implicitly sealed
- **const** – like C++ (compile time)
- **read-only** – like const, but runtime, set in constructor
Class Inheritance

- **Use**: to indicate inheritance (like extends in Java)
  - Just like C++

- **Constructors can invoke base constructor by mentioning name as in C++**
  - `public child (int x, int y) : base(x, y) ...`

- **Casting up and down like C++**

- **Use virtual keyword to indicate virtual functions**

- **Abstract class concept**
  - Using abstract keyword
Interfaces in Java 1.4

```java
interface IA {
    void g ();
}

interface IB extends IA {
    void f ();
}

interface IC extends IA {
    void f ();
}

class X implements IB, IC {
    void g () { System.out.println ("g"); }
    void f () { System.out.println ("f"); }
}

ERROR – This will not compile in JAVA 1.4 and before
```
Interfaces in Java 1.5

interface IA {
    void g ();
}

interface IB extends IA {
    void f ();
}

interface IC extends IA {
    void f ();
}

class X implements IB, IC {
    void g () { System.out.println ("g"); }  
    void f () { System.out.println ("f"); }  
}  

This will compile in JAVA 1.5! Notice, only one ‘f()’!
Interfaces in C#

```csharp
class Test {
    public static void Main () {
        X x = new X ();
        ((IA)x).g();
        ((IC)x).f();
        ((IB)x).f();
    }
}

interface IA {
    void g ();
}

interface IB : IA {
    void f ();
}

interface IC : IA {
    void f ();
}

class X : IB, IC {
    void IA.g () { Console.WriteLine ("IA.g"); }
    void IC.f () { Console.WriteLine ("IC.f"); }
    void IB.f () { Console.WriteLine ("IB.f"); }
}
```
**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; ... }
```

```c
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
# Classes and Structs - Differences

<table>
<thead>
<tr>
<th>Class</th>
<th>Struct</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Reference type</strong></td>
<td><strong>Value type</strong></td>
</tr>
<tr>
<td><strong>Can inherit from any non-sealed reference type</strong></td>
<td><strong>No inheritance (inherits only from System.ValueType)</strong></td>
</tr>
<tr>
<td><strong>Can have a destructor</strong></td>
<td><strong>No destructor</strong></td>
</tr>
<tr>
<td><strong>Can have user-defined parameterless constructor</strong></td>
<td><strong>No user-defined parameterless constructor</strong></td>
</tr>
</tbody>
</table>
**C# Structs vs. C++ Structs**

- Very different from C++ struct

<table>
<thead>
<tr>
<th>C++ Struct</th>
<th>C# Struct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Same as C++ class, but all members are public</td>
<td>User-defined value type</td>
</tr>
<tr>
<td>Can be allocated on the heap, on the stack or as a member (can be used as value or reference)</td>
<td>Always allocated on the stack or as a member</td>
</tr>
<tr>
<td>Members are always public</td>
<td>Members can be public, internal or private</td>
</tr>
</tbody>
</table>
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();
**Struct Example**

```csharp
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
```

These are properties which we’ll describe later.
**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
```
using System;
public class ShowTypes {
    public static void Main(string[] args) {
        CheckType (5);
        CheckType (10f);
        CheckType ("Hello");
    }
    private static void CheckType (object obj) {
        if (obj is int) {
            Console.WriteLine("Integer parameter");
        } else if (obj is float) {
            Console.WriteLine("Float parameter");
        } else if (obj is string) {
            Console.WriteLine("String parameter");
        }
    }
}
Statements
Overview

- High C++ fidelity
- `if`, `while`, `do` require bool condition
  - no if (x=1) error
- `goto` can’t jump into blocks
- `switch` statement
  - No fall-through is allowed you must use break
- `foreach` statement
  - Go through arrays or collections
- Expression statements must do work

```cpp
void Foo() {
    i == 1;   // error
}
```
Variables

- Variables must be assigned a value before they can be used
  - Explicitly or automatically
  - Called definite assignment
- Automatic assignment occurs for static fields, class instance fields and array elements

```csharp
void Foo() {
    string s;
    Console.WriteLine(s); // Error
}
```
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
    }
}
```
Program Structure
Main Method

- Execution begins at the static `Main()` method
- Can have only one method with one of the following signatures in an assembly
  - `static void Main()`
  - `static int Main()`
  - `static void Main(string[] args)`
  - `static int Main(string[] args)`
- `args[0] = param1, args[1] = param2` (in C++, `args[0]` is program.exe name)
Passing Arguments

- **Java**
  - Primitives by value; Objects by ref
- **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
Examples

- Timer
- Drawing
- Class with properties