Announcement

- Assignment 1 will be released tonight
  - Due next Friday
  - Submit through CMS
  - Write a small C# program
    - Input: A matrix and an operator (I/D/T)
      - I: Increase all elements in matrix by 1
      - D: Decrease all elements in matrix by 1
      - T: Return the transpose of this matrix
Visual C# 2008 Express walkthrough
.Net framework, CLR, CTS
- There’s no concept of primitive type in C#
  - `int` is a value type
  - `string` is a reference type
C# language guide
Outline

- C# Type
  - Value types
  - Reference types
  - Boxing and unboxing
- Basic C# Features: Arrays
Common Type System

Type

Value types
- Built-in value types
- User-defined value types
- Enumerations

Reference types
- Pointer types
- Interface types
- Self-describing types
  - Arrays
  - Class types
    - User-Defined Classes
    - Boxed Value Types
    - Delegates
Value Types

- Built-in value types
- User-defined value types
- Enumerations
## Built-in Value Types

### Integer Types

<table>
<thead>
<tr>
<th>Type Specifier</th>
<th>Bits</th>
<th>Range</th>
<th>Data Suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>sbyte</td>
<td>8</td>
<td>– 2^7 through 2^7-1</td>
<td></td>
</tr>
<tr>
<td>byte</td>
<td>8</td>
<td>0 through 2^8-1</td>
<td></td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>-2^15 through 2^15-1</td>
<td></td>
</tr>
<tr>
<td>ushort</td>
<td>16</td>
<td>0 through 2^16-1</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>-2^31 through 2^31-1</td>
<td></td>
</tr>
<tr>
<td>uint</td>
<td>32</td>
<td>0 through 2^32-1</td>
<td>U, u</td>
</tr>
<tr>
<td>long</td>
<td>64</td>
<td>-2^63 through 2^63-1</td>
<td>L, l</td>
</tr>
<tr>
<td>ulong</td>
<td>64</td>
<td>0 through 2^64-1</td>
<td>UL, ul</td>
</tr>
</tbody>
</table>
## Built-in Value Types

### Floating-Point Data Types

<table>
<thead>
<tr>
<th>Type Specifier</th>
<th>Bits</th>
<th>Range</th>
<th>Data Suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>32</td>
<td>(\pm 1.5 \times 10^{-45}) to (\pm 3.4 \times 10^{38})</td>
<td>F, f</td>
</tr>
<tr>
<td>double</td>
<td>64</td>
<td>(\pm 5.0 \times 10^{-324}) to (\pm 1.7 \times 10^{308})</td>
<td>D, d</td>
</tr>
</tbody>
</table>

### Declaration
- float tax = 0.5f;
- double tax = 0.5; / double tax = 0.5d;
Built-in Value Types

- **Boolean Types**

<table>
<thead>
<tr>
<th>Type Specifier</th>
<th>Bits</th>
<th>Range</th>
<th>Data Suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>8</td>
<td>true, false</td>
<td></td>
</tr>
</tbody>
</table>

- Traditionally, value types are stored outside dynamic memory allocation range
  - Stored in *stack* rather than *heap*
Structs
- Contains a collection of fields, methods and properties
- Non-extensible (sealed)
- Suitable for representing lightweight objects
  - Struct is less expensive in terms of memory
  - e.g. Point
User-defined Value Types

- **Definition**
  ```csharp
  struct Point
  {
    public int x;
    public int y;
    public Point(int x, int y)
    {
      this.x = x;
      this.y = y;
    }
  }
  ```

- **Instantiation**
  ```csharp
  Point a = new Point(10, 10);
  a.x = 20;
  ```
More on Structs

- System.Int32, System.Byte, System.Double, etc. are all structs
- Different from Java
Definition

```java
enum ClassDay {
    Monday,
    Wednesday,
    Friday
}
```

Instantiation

```java
ClassDay today = ClassDay.Friday;
```
Reference Types

- Reference types are always dynamically allocated
- Variables based on reference types (called objects) store *references* to the actual data
- The following keywords are used to declare a reference type
  - class, interface, delegate
- Example
  - String type: string
Comparison of Types

- **Value**
  - Value type variables directly contain values
  - Reference type variables contain a *pointer* to real object

- **Inheritance**
  - Value types inherit from `System.ValueType`
    - Treated specially by runtime; no subclassing
  - Reference types inherit from `System.Object`

- **Assignment**
  - Assigning one value type to another copies the contained value
  - Assigning one reference type object to another duplicates the reference but not the actual value!
Refer to a memory location
  - very much like pointers in other languages like C/C++

Can be set to null

TypeA a = new TypeA();
TypeA b = a;
Memory Layout: Value Types

- Contain the actual value, not the location
- Inherit from System.ValueType
  - treated specially by the runtime: no subclassing
- Copies of value types make a *real* copy

```csharp
int a = 123;
int b = a;
```
Value types variables are not objects
  - This gives performance gain in most cases
  - But value variables can become objects on demand
  - Called “boxing”; reverse is called “unboxing”

```c
int a = 123;
object o1 = a;
object o2 = o1;
int b = (int) o2;
```
This is important for parameter passing
C# Variables

- **Definite assignment**
  - int i; // i is a local variable
  - Console.WriteLine(i); //error CS0165: Use of unassigned local variable 'i'

- **Default values**
  - only for instance variables, static variables, and array elements
    - string s;     // s == null
    - double d;     // d == 0.0
C# Arrays

- Simple definition
  - int[] array = new int[30];

- “Jagged” Arrays
  - int[][] array = new int[2][];
    array[0] = new int[100];
    array[1] = new int[5];
  - The “outer” array with two elements will be stored consecutively.
  - However, the inner arrays (array[0] & array[1]) will not be stored consecutively in heap.
    - Recall that arrays are reference-type objects
  - Can have arbitrary dimensions
C# Arrays

- Multidimensional Arrays
  - Stored sequentially
  - Visually look like a rectangle (or a cube or a hypercube depending on the number of dimensions)

- Example
  - `int[,] array = new int[9, 9];
    array[3,8] = 100;`