C# Slides adapted from 4 week course at Cornell by Tom Roeder
A First Program

using System;

namespace Test {
    int a = 137
    class Hello {
        public static void Main(string[] args) {
            Console.WriteLine("Hello {0}", a);
        }
    }
}

Using brings in a namespace, which is an abstract container of symbols

Console.WriteLine is used to send formatted output to the screen. A format is of the form {index [,alignment][:formatting]}


Inheritance

class A {
    protected int a;
    public virtual void print() {
        Console.WriteLine("a = " + a);
    }
}

class B : A {
    public override void print() {
        Console.WriteLine("a’s value is " + (a + 42));
    }
}
using System;

class A {
    public void F() { Console.WriteLine("A.F"); }
    public virtual void G() { Console.WriteLine("A.G"); }
}

class B: A {
    new public void F() { Console.WriteLine("B.F"); }
    public override void G() { Console.WriteLine("B.G"); }
}
class Test {
    static void Main() {
        B b = new B();
        A a = b;
        a.F(); b.F(); a.G(); b.G();
    }
}
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
Common types

- Everything in C# inherits from `object`
  - Complaint: too slow
  - Java reasoning: no need to waste space

- Integer types:
  - Signed: sbyte, int, short, long
  - Unsigned: byte, uint, ushort, ulong

- Floating point: float, double
Common types

- string type: string
  - can index like char array
  - has method Split
- e.g.,
  - string s = “Hello”;
    char third = s[2];
    string[] split = s.Split(third);
Common types

- Default values
  - only for instance variables, static variables, and array elts
  - eg.
    - double x; // x == 0
    - string f; // f.equals("")
    - A a; // a == null

- what is the difference between double and class A?
  - reference types vs. value types
  - two families of types in C#

<table>
<thead>
<tr>
<th>Value type</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>false</td>
</tr>
<tr>
<td>byte</td>
<td>0</td>
</tr>
<tr>
<td>char</td>
<td>'o'</td>
</tr>
<tr>
<td>decimal</td>
<td>0.0M</td>
</tr>
<tr>
<td>double</td>
<td>0.0D</td>
</tr>
<tr>
<td>enum</td>
<td>The value produced by the expression (E)0, where E is the enum identifier.</td>
</tr>
<tr>
<td>float</td>
<td>0.0F</td>
</tr>
<tr>
<td>int</td>
<td>0</td>
</tr>
<tr>
<td>long</td>
<td>0L</td>
</tr>
<tr>
<td>sbyte</td>
<td>0</td>
</tr>
<tr>
<td>short</td>
<td>0</td>
</tr>
<tr>
<td>struct</td>
<td>The value produced by setting all value-type fields to their default values and all reference-type fields to null.</td>
</tr>
<tr>
<td>uint</td>
<td>0</td>
</tr>
<tr>
<td>ulong</td>
<td>0</td>
</tr>
<tr>
<td>ushort</td>
<td>0</td>
</tr>
</tbody>
</table>
Reference Types

- Normal objects (as in Java)
  - inherit from object
  - refer to a memory location
  - can be set to null
  - very much like pointers in other languages

```java
{  
  A a = new A();
  A b = a;
}
```

Diagram:
- `memory`
- `a`
- `b`
- `var of class A`
Value Types

- Contain the actual value, not the location
- Inherit from System.ValueType
  - treated specially by the runtime: no subclassing
  - not objects in normal case
  - but can become objects on demand

```csharp
{  
    int a = 137;
    int b = a;
}
```
Boxing and Unboxing

- Value types not objects
  - performance gain in common case
  - sometimes need to become objects
  - called “boxing”. Reverse is “unboxing”

```java
{  
    int a = 137;
    object o1 = a;
    object o2 = o1;
    int b = (int)o2;
}
```

Unboxing (explicit), if o2 is null or not an `int`, an `InvalidCastException` is thrown
Differences between types

- **Copy semantics:**
  - `Polynomial a = new Polynomial();
    Polynomial b = a;
    b.Coefficient[0] = 10;
    Console.WriteLine(a.Coefficient[0]);`  
    Output: 10
  - `int a = 1;
    int b = a;
    b = 10;
    Console.WriteLine(a);`  
    Output: 1

- **Copies of value types make a real copy**
  - important for parameter passing, too
  - boxing still copies
Value vs. Reference

Value
- Intrinsic types and structs (vector2d...)
- Passed by value (copied)
- Stored on the stack (unless part of a reference)

Reference
- Classes and interfaces, and “boxed” value types
- Passed by reference (implicit pointer)
- Variables sit on the stack, but hold a pointer to an address on the heap; real object lives on heap
Common Value Types

- All integer and floating point types
- Strings
- Anything that wouldn’t be an object in Java
- Structs
  - user-defined value types
  - can contain arbitrary data
  - non-extensible (sealed subclasses)
  - examples: Point, TwoDPoint, inheritance
Reference Types

- All are classes that are subtypes of object
  - single inheritance in class hierarchy
  - implement arbitrarily many interfaces
    - same idea for interfaces as in Java: access patterns
    - note interface naming: IAmAnInterface
  - can be abstract
    - class must be marked as abstract, but no member need be abstract
- May contain non-method non-data members
Arrays

- Can have standard C arrays
  - int[] array = new int[30];
  - int[][] array = new int[2][];
    array[0] = new int[100];
    array[1] = new int[1];
  - int[][] arr = new int[][] {
      new int[] {10, 11, 12},
      new int[] {13, 14, 15, 16, 17}
    };
- Called “jagged” arrays
- stored in random parts of the heap
- stored in row major order
- Can have arbitrary dimensions
- Recall that an array is an object
C# Arrays

- Multidimensional
  - stored sequentially
  - not specified what order
    - for instance: what is the order for foreach?
  - JIT computes the offset code
  - int[,] array = new int[10,30];
    array[3,7] = 137;

- saves computation for some applications
- can have arbitrary dimensions
C# Arrays - Multidimensional

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"}}};
C# Arrays

- can implement arbitrary storage order with a neat property trick:
  - indexers:
    ```csharp
    public int this[int a, int b] {
        get {
            // do calculation to find true location of (a,b)
            return mat[f(a, b), g(a, b)];
        }
    }
    ```
  - Allows “indexing” of an object
    - what sort of object might you want to index?
Properties

• Recall normal access patterns
  • `protected int x;
    public intGetX();
    public void SetX(int newVal);
  
• elevated into the language:
  public int X {
    get {
      return x;
    }
    set {
      x = value;
    }
Properties

- Can have three types of property
  - read-write, read-only, write-only
  - note: also have readonly modifier
- Why properties?
  - can be interface members
    ```csharp
    public int ID { get; }
    ```
  - clean up naming schemes
  - Abstracts many common patterns
    - static and dynamic properties of code; tunable knobs
    - note: in Java, used for function pointers
Indexers

- Allow bracket notation on any object
  ```csharp
  public string this[int a, double b] { ... }
  ```
- Used, eg. in hashtables
  - `val = h[key]`
  - simplifies notation
- Related to C++ operator[ ] overloading
- Special property
Function parameters

- **ref parameters**
  - reference to a variable
  - can change the variable passed in

- **out parameters**
  - value provided by callee

- **Note:** reference types are passed by value
  - so can change underlying object
Reference parameters

- **ref** must be used in both the call and declaration
  ```java
  public void Changer(ref int v)
  int myv;
  Changer(ref int myv)
  ```

- **ref** must be used in both the call and declaration
  ```java
  public void Changer(out int v)
  int myv;
  Changer(out int myv)
  ```

**Error:** myv not initialized

OK not to be initialized, however, must be assigned before Changer returns.
Function parameters

- For variable number of parameters
  - public void f(int x, params char[] ar);
  - call f(1), f(1, ‘s’), f(1, ‘s’, ‘f’), f(1, “sf”.ToCharArray());
  - explicit array
- where is this used?
  - example from C: printf

- Can use object[] to get arbitrary parameters
  - why would we want to avoid this?
  - will box value types
Iterators

- Common code pattern: walk a data structure
  - want to abstract to a GetNext() walk
  - iterator returns next element in walk
  - can be done explicitly:
    ```csharp
    IDictionaryEnumerator iDictEnum = h.GetEnumerator();
    while(iDictEnum.MoveNext()) {
        object val = iDictEnum.Value;
        object key = iDictEnum.Key;
        // do something with the key/value pair
    }
    ```
Iterators

- **C# way**
  ```csharp
  foreach (object key in h.Keys) {
      object val = h[key];
      // do something with the key/value pair
  }
  ```

- Can do even better with generics (C# 2.0)
  - can know the type of the key
  - then no need to cast

- now in Java (1.5) too
  ```java
  for (Object o : collection) { ... }
  ```
Iterators

- Can implement own iterable class
  - must implement IEnumerable:
    public IEnumerable GetEnumerator() { ... }
  - IEnumerator: MoveNext(), Current, Reset()
- old way (C# 1.1)
  - implement a state machine in an inner class
  - keeps track of where and returns next
  - tedious and error prone
C# 2.0 Iterators

- **Major change:** `yield return`
  - compiler builds the inner class
  - eg.
    ```csharp
    public IEnumerator GetEnumerator() {
        for(int i = 0; i < ar.Length; i++) {
            yield return ar[i];
        }
    }
    ```
- Also have `yield break`
- limited form of co-routines
Comparators

- Sort method on many containers
  - provides efficient sorting
  - needs to be able to compare to objects

Solution: IComparer

```csharp
public class ArrivalComparer : IComparer {
    public ArrivalComparer() {}
    public int Compare(object x, object y) {
        return ((Process)x).Arrival.CompareTo(((Process)y).Arrival);
    }
}
```

- Can then call
  ```csharp
  sortedList.Sort(new ArrivalComparer());
  ```
From last time

- out parameters
  - difference is that the callee is required to assign it before returning
  - not the case with ref parameters
  - caller is not required to set an out parameter before passing it in
Constructs for Small Data

- **enum**
  - like in C: give names to a family of values
  - eg. `Color c = Red`
  - can define `enum Color { Red, Orange, Blue };`
  - as in C, can give values to each
  - implicit conversion to integers as needed
  - enums are value types
Nullable Types

- Old software engineering problem:
  - what is the default “unassigned” int value
    - -1? 0? some random value?
  - problem is that any of these may be meaningful
    - e.g., int fd = socket(...), int t = tempInKelvin()?

- C# 2.0 adds nullable types
  - given a value type, eg. int, use int? a
  - now can be set to null
Nullable Types

- Now null can function as the default for all
  - compiler sets up boxing/unboxing as needed
  - box contains a slot to note that it is null
- Just like implementation with a flag
  - done in the compiler: type-checked
  - less tedious and error-prone
- Conversion between types still works
  - as long as there already was a conversion
Partial Types

- Another software engineering concern
  - how to separate generated and written code?
  - often need to be in same class
  - eg. from Visual Studio’s wizards
- C# 2.0 solution: allow multiple files
  - `public partial class A { ... }
  - each file uses partial
  - compiler joins the class specs
Notes

- 2.0 keywords can be used as identifiers
  - partial, where, yield
  - (good, since those are useful variable names)
  - compiler distinguishes on context
- in fact, can make any keyword a regular ident
  - two ways:
    - Unicode characters: int \u0066\u006f\u0072 = 137
    - @ symbol: int @for = 137
Notes

- **static constructors**
  - add keyword static to constructor
  - will be called when first instance is constructed
  - useful for class-specific data
    - eg. sequence numbers, connections to services

- **explicit interface member instantiations**
  - expose a method only when explicitly cast to iface
  - write interface name before method name
  - eg. public void ICollector.Collect() { ... }
Generics - Motivation

- Consider hashtable usage in C#
  - just like the Java way
  - each object has an associated HashCode
  - hash tables use this code to place and search obj
    - IDictionaryEnumerator ide = h.GetEnumerator();
      while(ide.MoveNext()) {
        String VIN = (String)ide.Key;
        Car car = (Car)ide.Value;
        Console.WriteLine("Drive off in car {0}", VIN);
        car.Drive();
      }
Generics – Motivation

- Unfortunate that hash doesn’t know:
  - key is string
  - value is Car
- Could easily add wrong type (get exception)
- But: don’t want to code a new hash each time
  - prefer general implementations
- Thus need a meta-variable for the type(s)
  - like templates in C++, but well-typed
Generics

- Write public class Stack<T> { ... }
  - T is the type variable
  - will be instantiated when declared
  - Stack<int> intStack = new Stack<int>();
- Push some type failures to compile time
  - goal of software engineering
- Can have arbitrarily many type parameters
  - Dictionary<TKey, TValue>
Constraints on Generics

- What if we want to write
  ```java
  public class Stack<T> {
    public T PopEmpty() {
      return new T();
    }
  }
  ```
  Will this work?
  - In C++, yes.

- What’s wrong here?
  - What is the type T, determined at run time
    - Does it have a public parameterless constructor?
Where clauses

- Need a type-safe way
  ```csharp
  public class Stack<T> where T : new() {
    public T PopEmpty() {
      return new T();
    }
  }
  ```

- New constraints
  - guarantees public parameterless constructor
  - no more parameters currently allowed
  - workaround?
Further Constraints

- Suppose have interface
  ```java
  public interface iface { public void Ping(); }
  ```
- Want to assume that T implements iface
  ```java
  public class Stack<T> where T : iface {
    public void PingTop() {
      top.Ping();
    }
  }
  ```
- No need to cast
  - compiler uses type information to decide
  - eg. IClonable
Bare Type Constraints

- class StructWithClass<S,T,U>
  where S: struct, T
  where T: U
{
  ...
}

- Can require
  - class/struct = reference/value type
  - another parameter
    - type compatible with this parameter
- Think of drawing subtype relations (graphs)
Open and closed types

- Type is open if
  - it is an unresolved type parameter
  - contains an unresolved type parameter
- Closed if not open
- eg.
  - `T x = new T();`
  - `Stack<int> s;`
  - `C<double, U> c;`
Accessibility of types

- Suppose C is public and A is private
  - what is the accessibility of C<A>?  
    - Cannot be constructed as public
    - must be private

- In general?
  - take the intersection of all accessibility
  - public, protected, internal, private, protected internal
Errors

- class A { ... }
- class B { ... }
- class Incompat<S,T>
  where S: A, T
  where T: B
{
    ...
}
Errors

- class StructWithClass<S,T,U>
  where S: struct, T
  where T: U
  where U: A
  {
    ...
  }
Errors

interface I<T>
{
    void F();
}
class X<U,V>: I<U>, I<V>
{
    void I<U>.F() {...}
    void I<V>.F() {...}
}
Better iterators with Generics

- Implement `IEnumerable<T>`
  - then implement
    
    ```csharp
    public IEnumerator<T> GetEnumerator() {
        // eg. implementation for a Set
        foreach (T key in elements.keys) {
            yield return key;
        }
    }
    ```
Quirk

- Need to implement two methods:
  - `IEnumerator<T> GetEnumerable()`
  - `IEnumerator IEnumerable.GetEnumerable()`

```csharp
IEnumerator GetEnumerator()
{
    return GetEnumerator();
}
```
- Really should be generated by compiler

- `IEnumerator<T> inherits from IEnumerator`
Notes

- Null coalescing operator `??`
  - `a ?? b` is `a` if `a` is non-null and `b` otherwise
- `internal` modifier
  - Can only be accessed in this namespace
  - `internal class C { ... }`

Parse this

- `F(G<A,B>(7))`
- `rule: ( ) ] > : ; , . ?`
Collection Pattern for a class C

- Contains a public GetEnumerator that returns a class/struct/interface type (call it E)
- E contains a public method with signature MoveNext() and return type bool
- E contains a public property Current that permits reading the current value
- type of Current is called the element type
- Then foreach will work
Can implement more than one iterator

```csharp
public IEnumerable<T> BottomToTop {
    get {
        for (int i = 0; i < count; i++) {
            yield return items[i];
        }
    }
}

public IEnumerable<T> TopToBottom {
    get {
        return this;
    }
}
```