First Assignment
- Input specification: you can decide
  - As command-line arguments, or ask user to type during run-time
  - As a single string, a one dimensional array, or a multidimensional array

Review
- OO features
  - Accessibility
  - Virtual and override
  - Class members
    - Property
    - Indexer
    - Operator
- Function parameters: ref

Function Parameters: ref
- ref parameters
  - reference to a variable
  - can change the variable passed in
- Caller must use the ref key word as well!
- void F(ref int x) {
  x = 1;
}
  int x = 0;
  F(ref x); //what’s the value of x?

Function Parameters: ref
- Note
  - Without ref, reference types are passed by value
  - But can change underlying object
- class A {
  public int value;
  public A(int val) { value = val; }
}
  void F(A a) {
    a = new A(1);
    a.value = 1;
  }
  A a = new A(0);
  F(a); //what is a.value?
**Function Parameters: out**

- **out parameters**
  - value provided by callee (required)
- void F(out int x) {
  - x = 1;
}
- int x; //no need to initialize
- F(out x); //what's the value of x?
- **Why?**
  - Used for a function to return multiple values

**Function Parameters: params**

- 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());
  - 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:
    - IDictionaryEnumerator iDictEnum = h.GetEnumerator();
    - while(iDictEnum.MoveNext()) {
      - object val = iDictEnum.Value;
      - object key = iDictEnum.Key;
      - // do something with the key/value pair
    }

**Iterators: Implementation**

- Can implement own iterator (inner) class
  - must implement IEnumerable:
    - public IEnumerator 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 iterator class
  - eg. ar is a private array field in your collection class
    - public IEnumerator GetEnumerator() {
      - for(int i = 0; i < ar.Length; i++) {
        - yield return ar[i];
      }
    }
  - also have yield break
  - ends the iteration
Nullable Types

- Old software engineering problem:
  - what is the default "unassigned" int value
    - -1? 0? 9999? 0xFFFF?
  - problem is that any of these may be meaningful
  - Contrast: reference types can have null value
- C# 2.0 adds nullable types
  - value types that can be assigned a null value
  - given a value type, eg. for int, use int?
    - int? a = 5;
    - int b = 10; int? c = b;

HasValue and Value properties

- int? x = 5; //how about int? x = null
  - If (x.HasValue) Console.WriteLine(x.Value);
- Conversion between types still works
  - as long as there already was a conversion
  - E.g. converting double? to int?
- Null coalescing operator ??
  - a ?? b is a if a is non-null, and b otherwise
  - E.g. int? a = null;
    - int b = a ?? -1; //b is ?

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

Generics - Motivation

- Consider a general-purpose stack in C# 1.0
  - public class Stack {
    - object[] items;
    - int count;
    - public void Push(object item) { ... }
    - public object Pop() { ... }
  }
  - Stack stack = new Stack();
    - stack.Push(new Customer());
    - Customer c = (Customer)stack.Pop();
    - stack.Push(3); //type error
    - int x = stack.Pop(); //type error

Generics - Motivation

- Solution: specify a data type for Stack elts
  - public class Stack<T> {
    - T[] items;
    - int count;
    - public void Push(T item) { ... }
    - public T Pop() { ... }
  }
  - Stack<Customer> stack = new Stack();
    - stack.Push(new Customer());
    - Customer c = stack.Pop();
    - stack.Push(3); //type error
    - int x = stack.Pop(); //type error

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?
- What’s wrong here?

Where Clauses

- Need a type-safe way
  ```java
  public class Stack<T> where T : new() {
    public T PopEmpty() { return new T(); }
  }
  ```
- new constraints
  - guarantees public parameterless constructor

Interface 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

Type Constraints

- class StructWithURL<T,U> where S: struct, T where T: U
- Can require
  - class/struct = reference/value type
  - another class parameter
  - type compatible with this parameter
- Think of drawing subtype relations (graphs)

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

Better Iterators with Generics

- Implement IEnumerable<T>
  ```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> GetEnumerator()`
  - `IEnumerator IEnumerable.GetEnumerator()`

  ```csharp
  return GetEnumerator();
  ```

- Really should be generated by compiler

- `IEnumerator<T>` inherits from `IEnumerator`

**Notes on Iterators**

- Can implement more than one iterator
  - ```csharp
      public IEnumerable<T> BottomToTop { get {
          for (int i = 0; i < count; i++) {
            yield return items[i];
          }
      } }
      ```
  - ```csharp
      public IEnumerable<T> TopToBottom { get {
        return this;
      } }
      ```
  - ```csharp
      foreach (int i in stack.BottomToTop) ...
      foreach (int i in stack.TopToBottom) ...
      ```