Yin Lou
CS 2026, Spring 2010

Advanced C# Types

Review

- OO features
  - Accessibility
  - Virtual and Override
  - Class members
    - Properties
    - Indexers
    - Operators
- Function parameters
  - ref, out

Outline

- Function params
  - params keyword
- Iterators
- Advanced C# types
  - Nullable types
  - Partial types
  - Generics

params keyword

- Used in methods where the number of arguments is variable
- Only one params keyword can be used in a method
- No parameters defined after the params parameter

Syntax Example

- C#

```
public int SumGrades (params int[] grades)
{
        int sum = 0;
        for (int i=0; i<grades.length; i++)
                sum += grades[i];
        return sum;
}
```

- Java

```
public int sumGrades (int ... grades) {
        int sum = 0;
        for (int i=0; i<grades.length; i++)
                sum += grades[i];
        return sum;
}
```

Iterators

- Common programming pattern
- Allows you to walk through a collection of elements in a data structure
- Example

```
foreach (string name in names)
{
        Console.WriteLine(name);
}
```

- Similar to Java

```
for (String name : names) {
        System.out.println(name);
}
```

Creating your own Iterator

- Implement a method GetEnumerator() that returns an IEnumerator
  - IEnumerator: MoveNext(), Current, Reset()
- Use yield return to return current element
- Use yield break to conditionally stop the iteration

Custom Iterator Example

```
public class Foo : IEnumerable
{
        int[] myArray;
        public IEnumerator GetEnumerator()
        {
                for (int i=0; i<myArray.length; i++)
                {
                        yield return myArray[i];
                }
        }
}
```

- Also have yield break
  - ends the iteration

Nullable Types

- Built-in value-types, like int, have default values
- References are assigned null by default
  - An int can not be assigned null value
- Null values are useful to test whether a variable has been assigned to or not

Nullable Types

- C# 2.0 added nullable types
  - Value-types that can accept null
  - Example: int? a = null;
- The HasValue and Value properties
  - if (x.HasValue) { Console.WriteLine(x.Value); }
- The ?? operator
  - a ?? b evaluate to a if a is non-null, and to b otherwise
  - Similar to a != null ? a : b

Partial Types

- It is better to break up large classes into multiple files
  - Increase readability
  - Separate generate and hand-written code
- As of C# 2.0 , partial classes allow splitting code into multiple files
  - public partial class Foo { … }
  - Each file must use partial
  - Compiler joins all the classes

Generics

- Write public class Stack<T> { .. }
  - T is the type variable (a class name)
  - Will be instantiated when declared
  - Stack<int> intStack = new Stack<int>();
- Can have multiple types
  - public class Dictionary<TKey, TValue>
- Push some type failures to compile time
  - Reduce potential bugs
- Similar to templates in C++ and similar to generics in Java

Constraints on Generics

- What if we want to write

```
public class Stack<T>
{
        public T PopEmpty()
        {
                return new T();
        }
}
```

- Will this work?

New Constraints

- guarantees public default constructor

```
public class Stack<T>  where T : new()
{
        public T PopEmpty()
        {
                return new T();
        }
}
```

Interface Constraints

- Suppose have interface
  - public interface IFace { public void Ping(); }
- Want to assume that T implements IFace

```
public class Stack<T> where T : IFace
{
        public void PingTop()
        {
                top.Ping();
        }
}
```

- No need to cast
  - compiler uses type information to decide

Type Constraints

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

```
class StructWithClass<S> where S: struct  {…}
```

Better Iterators with Generics

- Implement IEnumerable<T>

```
public IEnumerator<T> GetEnumerator()
{
        // eg. implementation for a Set
        foreach (T key in elements.keys)
        {
                yield return key;
        }
}
```