C# 3.0

Mingsheng Hong
CS 215, Spring 2008

Review

- Reflection
- Conversion
  - Explicit and implicit conversions
  - User-defined conversions
- Exceptions

Exception Example

```
try {
  int x=5, y=0; x/=y;
} catch(DivideByZeroException except) {
  Console.WriteLine("Exception "+except.Message);
} catch(System.ArithmeticException except) {
  Console.WriteLine("Exception "+except.Message);
} catch(Exception except) {
  Console.WriteLine("Exception "+except.Message);
}
```

- Not a substitute for transfer of control!
  - `Customer c = obj as Customer;`
    `Customer c = (Customer) obj;`

Checked and Unchecked

- Two contexts for evaluating arithmetic
- unchecked
  - default context
  - overflows do not throw exceptions
  - can use `unchecked` operator to make explicit

```
double d = double.MaxValue;
unchecked { int i = (int)d; }
```

- checked
  - overflows throw `System.OverflowException`
  - use `checked` operator

Roadmap for Today's Lecture

- C# 3.0 language features
  - Implicitly typed variables
  - Automatic properties
  - Initializers
  - Anonymous types
  - Lambda expressions
  - Extension methods

C# Version 3

- High level points
  - less (finger) typing → shorter program → fewer bugs
  - better functional programming
  - LINQ: language-integrated query
- E.g. Retrieve items with more than 10 letters

```
string[] videoGames = {"Morrowind", "BioShock",
    "Half Life 2: Episode 1", "The Darkness"};
IEnumerable<string> subset = from g in videoGames
    where g.Length > 10 orderby g select g;
```

- Several features required to make this work
  - implicitly typed variables
  - extension methods

Implicitly Typed Local Vars

- Type of the variable *inferred* from expression
  - must include initializer

```
var i = 5;
var s = "Hello";
var d = 1.0;
var orders = new Dictionary<int,Order>();
```

  - works in for loops

```
var evenNumbers = new int[] { 2, 4, 6, 8 };
foreach (var item in evenNumbers) {
  Console.WriteLine("Item value: {0}", item);
}
```

  - can't be null. Why not?

Implicitly Typed Local Arrays

- Must have consistent types
  - `var a = new[] { 1, 10, 100, 1000 };`
  - or have implicit conversions
  - `Var b = new[] {1, "2", false}; //error`

Automatic Properties

```
class Car {
    private string carName = string.Empty;
    public string CarName {
          get { return carName; }
          set { carName = value; }
    }
}
```

- Automatic property syntax

```
class Car {
    public string CarName { get; set; }
}
```

Intializers

- Recall: named parameters in attributes
  - `[Help("http://…", Topic = "Programming")]`
- Initializers for writable public field or property
  -
```
Public class Point {
  public int X { get; set; }
  public int Y { get; set; }
}
```
  - `int p1 = new Point(); p1.X = 10; p2.Y = 20;`
    `var p2 = new Point { X = 10, Y = 20 };`
- Can be nested (eg. Rectangle with two public `Point` properties)

Collection Initialization

- Recall: initialize a standard array

```
int[] digits = new int[] {0, 1};
```

- Collection initializers
  - `List<int> digits = new List<int> {0, 1};`
- Applicable to `List<Rectangle>`

```
List<Rectangle> myListOfRects = new List<Rectangle> {
    new Rectangle {TopLeft = new Point {X=10, Y=10},
                   BottomRight = new Point {X=20, Y=20}},
    new Rectangle {TopLeft = new Point {X=2, Y=2},
                   BottomRight = new Point {X=10,Y=10}}
};
```

Anonymous Types

- `var x = new {P1 = 10, P2 = "name"};`
  - `x` is of anonymous type with two properties
  - type can't be referred to by name in program
- Structural type equivalence
  - two anonymous types can be compatible
- Can be nested

Lambda Expressions

- Generalized function syntax
  - λ x . x + 1
  - in C# 3.0, have `x => x + 1`
  - Syntax: (Input parameters) => {function body;}
- From anonymous method syntax:
  - `delegate(int x) { return x + 1;}`
- Example

```
List<int> list = new List<int> {0, 1, 2};
List<int> evenNumbers
   = list.FindAll(i => (i % 2) == 0);
```

Notes

- Can have implicitly typed variables
- Can have zero or more variables
- Can have expression or statement body
- Can be converted to a compatible delegate

```
delegate R Func<A,R>(A arg);
Func<int,int> f1 = x => x + 1;
Func<int,double> f2 = x => x + 1;
```

- If expression body, get expression trees

```
Expression<Func<int, int>> expTree
    = (x => x + 1);
```

Extension Methods

- Can add methods to existing classes
  - any methods (although look static)
  - new methods defined only in static classes
  - `this` modifier on the first parameter
- When import a namespace that has extensions, then added to classes
  - once imported, called as usual
- Local methods take precedence
  - first local for normal method, then extension

Examples

```
public static class Extensions {
    public static int ToInt32(this string s) {
      return Int32.Parse(s);
    }
    public static T[] Slice<T>(this T[] a,
                               int index, int count) {
      if (index < 0 || count < 0 || a.Length-index < count)
        throw new ArgumentException();
      T[] result = new T[count];
      Array.Copy(a, index, result, 0, count);
      return result;
    }
   static void Main(string[] args){
       int x = "123".ToInt32(); //same as ToInt32("123");
       string[] strs = {"ab", "cd", "ef"};
       foreach (var str in strs.Slice<string>(0, 2)) {
             Console.WriteLine(str);
       }
   }
}
```

Extending Interface Types

- Need to supply an implementation

```
interface IBasicMath {
  int Add(int x, int y);
}
class MyCalc : IBasicMath {
  public int Add(int x, int y) { return x + y;}
}
static class MathExtensions {
  public static int Subtract(this IBasicMath itf,
                             int x, int y) {
    return x - y;
  }
}
```

Type Inference & 3.0 Features

```
• //extension method defined in some static class
  public static IEnumerable<S> Select<T,S>(
                this IEnumerable<T> source,
                Func<T,S> selector) {
      foreach (T element in source)
            yield return selector(element);
  }
• var customers = new[] {
      new {Name = "Jack", ID = 8},
      new {Name = "Kate", ID = 15}};
  foreach (var n in customers.Select(c => c.Name)) {
      Console.WriteLine(x);
  }
```