7SENG001 Enterprise Application Development

Week 5

(with in lecture tutorials)

Programming C# Part 2

Programming C# Part 3

Introduction to C#

- Lecture 5 – Introduction to C# Part 2

Outline

- Literals
- Operators
- Assignment and Equality
- String operations, named operations, and formatting
- Casting and data conversion
- structs and enumerators
- Methods and in, out, and ref
- Checked, unchecked and using ‘as’

Literals

- **Integer** constants can be written as decimal or hexadecimal
  - 123
  - 0x45AF
  - 077 (is a decimal)
- Floating point and double
  - 1.0, 2E5,
  - 1.6 e-19
  - 132. is not valid

Literals

- **Character literals** are Unicode characters surrounded by single quotes
  - ‘X’, ‘u\20AC’ (€ symbol)
  - \n (newline)
  - \\ (backslash)
  - Can be implicitly converted to int, long etc
- String literals are characters enclosed in double quotes
  - “C:\\new_source\\my_data.txt”
  - And the verbatim we have seen previously @“C:\new_source\my_data.txt”

Revise

Operators

Operator category	Operators
Arithmetic	+ - * / %
Logical (boolean and bitwise)	& | ^ ! ~ && || true false
String concatenation	+
Increment, decrement	++ --
Shift	<< >>
Relational	== != < > <= >=
Assignment	= += -= *= /= %= &= |= ^= <<= >>=
Member access	.
Indexing	[]
Cast	()
Conditional	?:
Delegate concatenation and removal	+ -
Object creation	new
Type information	as is sizeof typeof
Overflow exception control	checked unchecked
Indirection and Address	* -> [] &

Work like other languages (note integer division truncates 11/4 gives 2)

Assignment

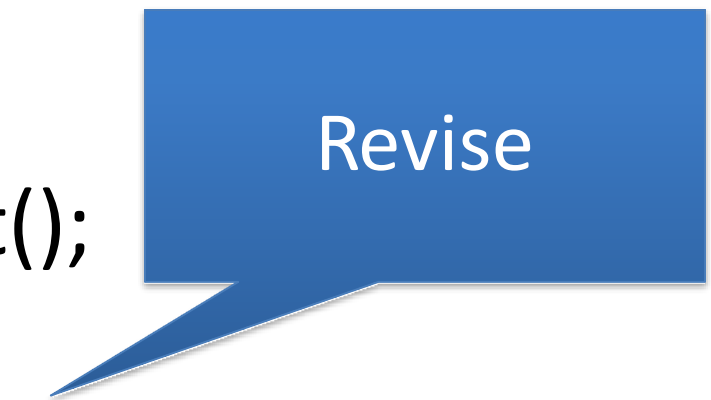
- int i = 5;
- int j;

j = i; //copying the value

Account a1 = new Account();

Account a2;

a2 = a1; //copying reference



Equality Operator

- int i = 5;
- int j = 5;
- if(i == j) //true if values are the same
- GameTile t1 = new GameTile();
- GameTile t2 = new GameTile();
- if (t1 == t2) //likely to be false as comparing the **reference** (not values inside)
- bool areEqual = System.Object.ReferenceEquals(t1, t2);
- **Must always be sure how equivalence is defined for your types**

Mixed type operations

- C# allows implicit mixed-type operations
  - Implicit conversions **from smaller to a larger is acceptable**
  - Floating point types are considered larger than integer types
- When conversion happens
  - The **value** of the smaller is converted to the bigger
  - The result is of the bigger type
  - Example:

```
int i = 5; long n = 4; double x;
x = n * i;      //i temporarily converted to long
                //long result converted to double
```

String operators

- Can be concatenated with other strings by using the + operator
- string output = “The value is: “ + value;
- String can be checked for equality of value by using the == and != operator
  - if (answer == “yes”)

String formatting

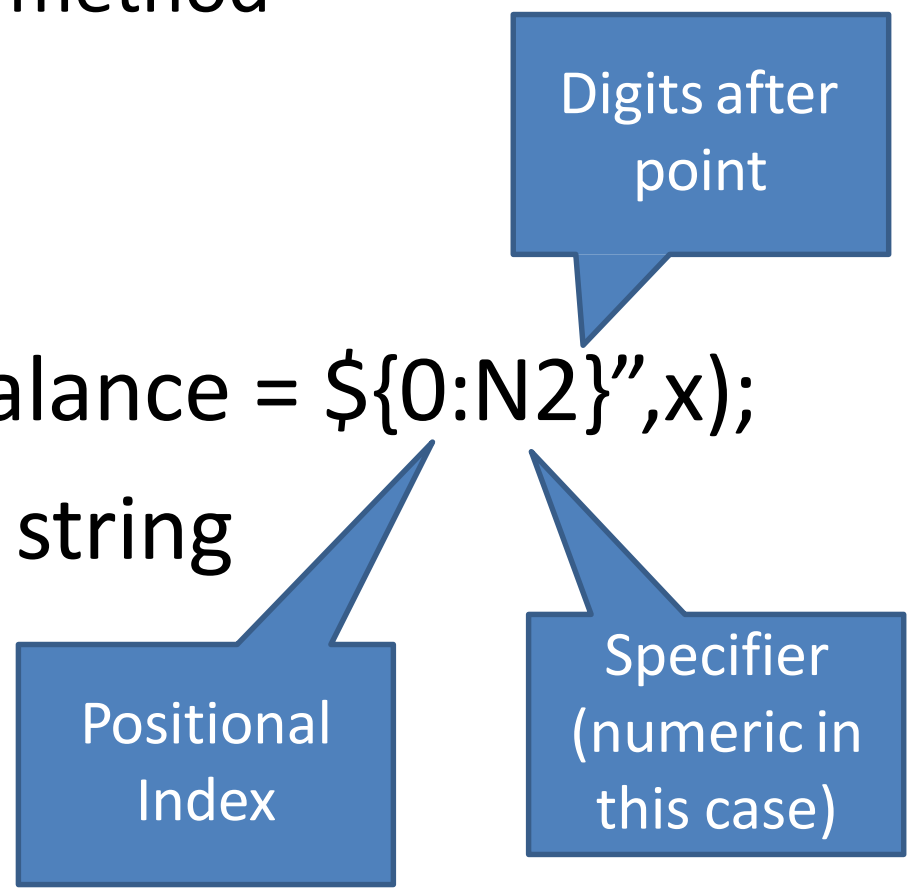
Note can be used in GUI formatted string too

- The string class has methods for formatting
  - Which is also used by the **System.Console.WriteLine** method
- Example

```
double x = 9924.456;
string fs = string.Format("Balance = ${0:N2}",x);
```

Will produce the formatted string

"Balance = $9,924.57"



String Formatting

- Format Specifiers
- C – currency
- D – decimal
- E – exponential
- F – fixed point
- G – general : accuracy is total number of digits
- N – numeric: similar to F but with separation in the thousands

Output Formatting

- Field width and justification can also be specified

string fs = string.Format(“{0,-5}”, data);

  - A left justified field that is five characters wide

- Width and formating combined and used directly in output

Console.WriteLine(“>{0,12:F3}<“, 127.23456);

> _ _ _127.132<

Format Specifier	Description	Examples	Output
C or c	Currency	Console.Write("{0:C}", 2.5); Console.Write("{0:C}", -2.5);	$2.50 ($2.50)
D or d	Decimal	Console.Write("{0:D5}", 25);	00025
E or e	Scientific	Console.Write("{0:E}", 250000);	2.500000E+005
F or f	Fixed-point	Console.Write("{0:F2}", 25); Console.Write("{0:F0}", 25);	25.00 25
G or g	General	Console.Write("{0:G}", 2.5);	2.5
N or n	Number	Console.Write("{0:N}", 2500000);	2,500,000.00
X or x	Hexadecimal	Console.Write("{0:X}", 250); Console.Write("{0:X}", 0xffff);	FA FFFF

Casting

Revise

- bool b; char c; int i; long n; double x;

c = 97; //implicit conversion in later versions

c = (char)97; //cast - c becomes Unicode ‘a’

i = (int)5.8; //cast – i gets value 5

b = i; // error – no conversion from int to bool

b = (bool)i; // error – no conversion from int to bool

n = i; //ok – implicit conversion, small to big

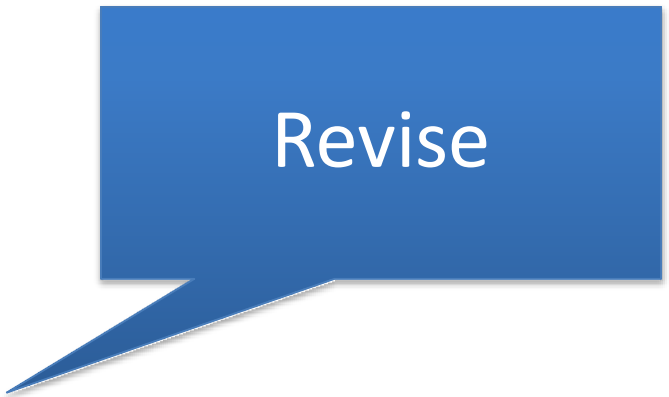
Be careful when casting, may indicate some is wrong with the design

Data Conversion

- All data types have the ability to parse a string into their type

String input = “23454”;

int i = int.Parse(input);



- Parses a string into an integer
- Also TryParse method

Value and reference variables

- Value
  - Data is stored and accessed directly
  - On the stack
  - **Structures, primive types (int, float, double etc.)**
- Reference
  - Data is stored and accessed indirectly
  - On the heap
  - **Classes, arrays …**

Structs

- For example used to represent a point on the screen (note that the Point object exists in C#)

```
struct  Point
{
   int x;
   int y;
}
```

Structs

- No inheritance
- Value variables - passed by value
  - Class based object are passed by reference
- Cannot be null
- Cannot have a finalize method (cleanup)
- When to use stucts vs classes??

Enums

```
public class MyClass
 {
 enum Computer { Compaq, Dell, HP, IBM }
public static int Main(string[] args)
    {
    Computer MyComputers = new Computer();
    MyComputers = Computer.Compaq;
    Console.WriteLine("${MyComputers}",
    MyComputers);  Console.Read(); // Wait for
    Return key  return 0;
    }
}
```

What are the advantages of enum

out and ref parameters

The **out** keyword causes arguments to be passed by reference. This is similar to the **ref** keyword except that it requires that the variable be initialized before being passed. To use an **out** parameter, both the method definition and the calling method must explicitly use the **out** keyword. For example:

```
public class MyClass
 {
    public static void setToFive(out int i) {
          i = 5;
     }
…
int i = 0;
setToFive(out i);
```

To use a **ref** parameter, both the method definition and the calling method must explicitly use the **ref** keyword

```
    public static void addOne(ref int i) {
          ++i;
     }
    e.g. addOne(ref i);
```

Normal call

```
    public static void multiplyByTen(int i) {
          i = i * 10;
    }
```

out ref

```
public static void inOutRef()
 {
        int i;
 setToFive(out i); // Out variables can be passed un-
    initialised
// i initialised by call to setToFive() AddOne(ref i);
// Pass by reference – changes externally visible
 mangle(i); // Pass by value – changes not seen externally
 Console.WriteLine("i set to {0}", i);
 }
```

out vs ref

- **ref** tells the compiler that the object is initialized before entering the function
- while **out** tells the compiler that the object will be initialized inside the function.

Examples - out

```
static void passByOut(out int c, out float d)
        {
       //here we are forced by the compiler
to initialise the parameters
       //we will just reset the variables to
zero
            c = 0;
            d = 0.0f;

        }
```

ref

```
static void passByRef(ref int a, ref float b)
        {
    //we are not forced to initialise

                a =  a + 1;
                b =  b * b;

        }
```

How we call out and ref

```
int a = 1;
float b = 3.5f;
Console.WriteLine("a before pass = " + a + " b before pass = " + b);
passByRef(ref a, ref b);
Console.WriteLine("a after pass = " + a + " b after pass = " + b);
Console.ReadLine();
int c = 25;
float d = 6.5f;
Console.WriteLine("c before pass by out = " + c + " d before pass by out = " + d);
passByOut(out c, out d);
Console.WriteLine("c after pass by out = " + c + " d after pass by out = " + d);
Console.ReadLine();
Console.WriteLine("Pass by 'in' to multiply 3*4.0 for example: " +
    passByInToMultiply(3, 4.0f));
Console.ReadLine();
```

output

```
a before pass = 1 b before pass = 3.5
a after pass = 2 b after pass = 12.25

c before pass by out = 25 d before pass by out = 6.5
c after pass by out = 0 d after pass by out = 0

Pass by 'in' to multiply 3*4.0 for example: 12
```

But a note of caution

- Document side effects
- Using out or ref parameters requires experience with pointers, understanding how value types and reference types differ, and handling methods with **multiple return values**.

  Framework architects designing for a general audience should not expect users to master working with out and ref parameters

- However, you might find this occasionally useful for return multiple parameters (C, C++ like)

Expression evaluation

```
public static int Main(string[] args) {
 byte b = 255;
unchecked // Ignore overflows
 {
 b += 10; // Overflow byte
 }
 Console.WriteLine("{0}", b);
 return 0; }
```

Check for overflow

```
public static int Main(string[] args) {
 byte b = 255;
checked // Check for overflow
 {
 b += 10; // Overflow byte
 }
 Console.WriteLine("{0}", b);
 return 0; }
```

Using as – the exception proof cast

```
public class TypeConvert {
    public static string ChangeToString(object obj) {
        /*if obj isn't based on a string the usual way to cast is
         string s = (string) obj; and this would cause an exception * */
        string s = obj as string;  //use this instead
        if (s != null) {
        // The conversion worked
        return s;
    }
}
```

Introduction to C#

- Lecture 5 – Introduction to C# Part 3

Outline

- static vs dynamic typing
- Expression-bodied methods
- Delegates
- Lambdas
- Events

var – implicit type

- `var` says let the compiler figure out the type.
- `dynamic` says let the runtime figure out the type.

```
dynamic x = "hello";  // Static type is dynamic, runtime type is string
var y = "hello";      // Static type is string, runtime type is string
//int i = x;            // Runtime error (cannot convert string to int)
//int j = y;          // Both static types are string, so this is OK.
x = 1;      // Works! Is this dangerous?
y = 1;      // Compile time error, so safer.
```

- Another example
  - `var vowels = new[] {'a','e','i','o','u'};   // Compiler infers char[]`
- What about readability?

Expression-bodied methods

- A method that comprises a single expression, such as

int Foo (int x) { return x * 2; }

can be written more tersely as an *expression-bodied method*.

- A fat arrow replaces the braces and return keyword

int Foo (int x) **=> x * 2;**

- Expression-bodied functions can also have a void return type:

void Foo (int x) **=> Console.WriteLine (x);**

Delegates - declaration

- Placeholder for functions: reference to a method.
  Used for implementing events and callback methods
- A delegate **type** must be declared before use:

keyword return delegate type parameters

```
delegate int  MyDel(int x);
```

- Declares form of the methods referenced by delegate
- Declare variables

delegate type variable

```
MyDel delVar;
MyDel dVar;
```

Creating a delegate

- Create delegate object and save reference of method:

```
delVar = new MyDel(myInstObj.MyM1 ); //instance
 dVar = new MyDel(SClass.OtherM2 ); //static
```

- Or use short hand specifier

```
delVar = myInstObj.MyM1;
 dVar = SClass.OtherM2;
```

- Declare on one line

```
MyDel dVar = SClass.OtherM2;
```

Delegates - invocation

- Invoke a delegate like you would a normal function

```
MyDel dVar = SClass.OtherM2;
dVar(32);
```

- Delegates have an invocation list of methods that can be added and removed

```
MyDel delC = delA + delB;
delD += delC; delD -= delA;
```

- Create new delegate with the composed methods in its invocation list
- Calling delC(44) is the same as calling delA(44) and then delB(44)

```
// This delegate can point to any method,
// taking two integers and returning an integer.
public delegate int BinaryOp(int x, int y);
// This class contains methods BinaryOp will
// point to.
public class SimpleMath
{
    public static int Add(int x, int y)
    { return x + y; }
    public static int Subtract(int x, int y)
    { return x - y; }
}
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Simple Delegate Example *****\n");
        // Create a BinaryOp delegate object that
        // "points to" SimpleMath.Add().
        BinaryOp b = new BinaryOp(SimpleMath.Add);
        // Invoke Add() method indirectly using delegate object.
        Console.WriteLine("10 + 10 is {0}", b(10, 10));
        Console.ReadLine();
    }
}
```

Lambda expressions

- C# 3.0 introduced lambda expressions (a simpler syntax for anonymous methods)
- Lambda expressions are inline function definitions
- Remember a function (method) can be assigned to a delegate matching its signature and return type

```
public static int Add20(int x)
{
    return x + 20;
}
```

```
// Anonymous method or function
MyDel del = delegate (int x) { return x + 20; };
// Lambda expressions
MyDel le1 = (int x) => { return x + 20; };
MyDel le2 = (x) => { return x + 20; };
MyDel le3 = x => { return x + 20; };
MyDel le4 = x => x + 20;
```

Events

- Are a simpler form of delegate, can only add, remove, and invoke event handlers
- .NET provides predefined delegate type to use with events:

```
public delegate void EventHandler(object sender, EventArgs e);
```

- An event has methods registered with it and invokes those methods when it is invoked
- Enable asynchronous communication (no waiting)
- Can be switched on and off as needed (enhanced performance)
- Events can be chained, more than one handler can execute for a given event