Coding for Game Development

Section Four

XNA Game Programming

Object Orientation in C# [Part One]

Common Numeric Types in XNA

- **int** 32 bit integer
  default for whole number values
- **long** 64 bit integer
  rarely used in XNA
- **float** 32 bit floating point value
  frequently used in XNA
- **double** 64 bit floating point value
  default for numbers with decimal point
- **decimal** 128 bit floating point value
  rarely needed in XNA

Numeric Literals

- int: 12 15 34
- long: 1234245543345334L
  Lower case L ("l") not recommended!
- float: 12.3F 12.3f
- double: 12.3 .123 12e4 1.23e-4 not 123.
- decimal: 12.34M 12.34m (remember "*M*oney")
- Hex int: 0x123ABD456 0x123abc456
- Hex long: 0x1234ABCD5678EFABL
  0x1234abcd5678efabL

float, **double** and **decimal**

- In normal C# programming, it is common practice to use **double** for numeric values requiring a decimal point and **decimal** for monetary values.
- This is reflected in the literal numeric constants:
  - **double** literal: 27.96
  - **float** literal: 27.96F
  - **decimal** literal: 27.96M
- When using **double**, there's no need to add any special characters at the end of each literal.

Floating point numbers in XNA

- In games programming, it is traditional to use **float** instead of **double**.
- Being smaller and faster, **float** gave a performance increase which was useful for earlier computers, but not so important today.
- The XNA library still requires **float** for most floating point values and method parameters.
- This gives the designer a tricky choice:
  - Use **float** throughout and add F to all literals.
  - Use **double** and cast to float when calling libraries.
- Neither option is obviously better than the other!

Other Common Types

- **bool** true or false value
  note spelling: *not* boolean
- **string** variable length string of characters
- **char** single Unicode character (16 bits)

String Literals

- There are two forms of string literal:
  - Normal literals (can contain "\"escape codes):
    "abddec" "Bob\'s book" "\n" "me@my.com"
    "c:\\folder\\file.com" "embedded \"quotes\""
  - Verbatim literals (with leading @):
    (useful for Windows file names with embedded \)
    @"abddec" @"Bob's book" @"me@my.com"
    @"c:\folder\file.com" @"embedded ""quotes"""
  - Strings can be joined using the + operator:
    - @"c:\folder\" + "file.com" + "\n"

Other Types

- **sbyte** 8 bit signed integer
- **byte** 8 bit unsigned integer (not ubyte!)
- **short** 16 bit signed integer
- **ushort** 16 bit unsigned integer
- **uint** 32 bit unsigned integer
- **long** 64 bit signed integer
- **ulong** 64 bit unsigned integer

These types are *not* for general use – they should only be used when specifically required by the code

Strongly Typed Language

- C# is a *strongly typed* language - meaning that we can't easily mix different types in an expression and let the compiler sort it out:
  - Most of the time, mixing the wrong types together is an indication that we've got our program design wrong.
  - Even if we do try to let the compiler sort it out, it usually gets it wrong and puts a bug in our code.
- When we need to be able to convert values from one type to another, we use the **Convert** class. For example:

```
string s = "-12.45";
double d = Convert.ToDouble(s);
```


Convert Class Methods

- Conversion methods include:
  - ToInt32(v) Converts to an int value
  - ToBoolean(v) Converts to boolean.
  - ToByte(v) Converts to an 8-bit unsigned integer.
  - ToChar(v) Converts to a Unicode character.
  - ToDecimal(v) Converts to decimal.
  - ToDouble(v) Converts to double.
  - ToSingle(v) Converts to float.
- The value to be converted can be either a value of some other type, or a string containing the value.

Type Casting

- Convert is very useful, but because it requires method calls it can be somewhat slow to execute.
- A faster alternative is to use a type cast in those places where the compiler allows it.
- To use a type cast we place the required type name in parentheses before the value.
- For example, if we have a **double** value but the method parameter requires a **float**, we can use a type cast like this:

(float) doubleValue

Class Skeleton

- A C# class takes this form:

```
using System;
using ......;
namespace projectName
{
    class ClassName
    {
        ... fields ....
        ... methods ....
    }
}
```

- The *namespace* is optional but is put in by the IDE.
- Typically, classes are marked **public**, **internal** or perhaps not marked at all (which defaults to **internal**)

Fields

- A class usually contains *fields*.
- These are data items which are:
  - Shared between methods
  - Persist between method calls.
- A field declaration is placed inside the class but *not* inside any method.
- For beginners, I recommend that you put all your fields near the top of the class file.
- When an object is created, a separate set of fields is created for each object.

Example Class with Fields

- Here is an example of a class which has five fields and no methods.
- Note there is a C# convention that private field names begin with an underscore character:

```
class MotorCar
{
    string _make;
    string _model;
    int _numberOfWheels;
    int _gear;
    bool _lightsOn;
}
```

- Note: To save space on the slide, I've omitted the *using* and *namespace* lines.

Creating Objects

- When we have a class, we can create several objects from it:

```
MotorCar bobsCar = new MotorCar();
MotorCar jillsCar = new MotorCar();
```

- Each object has its own separate set of fields, and each field has its own independent value.

bobsCar

```
_make:  "Skoda"
_model: "Fabia"
_numberOfWheels: 4
_gear: 3
_lightsOn: false
```

jillsCar

```
_make:  "Ford"
_model: "Ka"
_numberOfWheels: 4
_gear: 0
_lightsOn: true
```


Adding Some Methods

- Let's add a couple of methods to the class:

(lines omitted to save space)

```
class MotorCar
{
    : : : : :
    int _gear;

    public void GearUp() {
      _gear++;
    }
    public void GearDown() {
      _gear--;
    }
}
```

- Each method modifies the **_gear** field. The change is permanent, but only the current object is affected.

Calling the Methods

- We call methods using this form:

***objectName.MethodName*();**

- Applying this to our MotorCar objects:

```
bobsCar.GearDown();

jillsCar.GearUp();
```

bobsCar

```
_make:  "Skoda"
_model: "Fabia"
_numberOfWheels: 4
_gear: 3 — 2
_lightsOn: false
```

jillsCar

```
_make:  "Ford"
_model: "Ka"
_numberOfWheels: 4
_gear: 0 — 1
_lightsOn: true
```


Constructors

- For each class we can define a constructor method which is executed when the object is created:

no type or void!

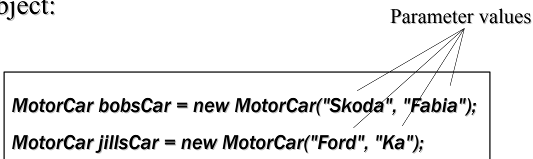
```
class MotorCar
{
    string _make;
    string _model;
    int _numberOfWheels;
    int _gear;
    bool _lightsOn;

    public   MotorCar(string make, string model)
    {
      _make = make;
      _model = model;
      _numberOfWheels = 4;
      _gear = 0;
      _lightsOn = false;
    }
}
```

parameters

Creating Objects using a Constructor

- A constructor is used to initialise the object to known and safe values.
- When a class has a constructor, we must pass the appropriate values into the constructor when we create the object:



- Note: any method can be defined with parameters, not just constructors.