SIT232 - OBJECT ORIENTED DEVELOPMENT

Session 2. Control structures, classes, and data types

Outline

- Session 02. Control structures, classes, and data types
  - Objectives
  - The Structure Theorem
  - Defining Classes
  - Value Types and Reference Types
  - Composite Formatting
  - Logic Errors and Debugging

SESSION 2. CONTROL STRUCTURES, CLASSES, AND DATA TYPES

Objectives

- At the end of this session you should:
  - Be familiar with the ***selection and repetition control structures*** provided by C# and be able to apply them in your programs;
  - Be able to ***define classes consisting of instance variables, methods, and constructors*** in C#;
  - Be able to ***instantiate objects*** from your own classes and build C# applications using them; and
  - Be able to ***produce complex formatted output*** and diagnose and correct logic errors in C#.

The Structure Theorem

- Complex programming problems must first be decomposed if we are to solve them
  - The object-oriented paradigm allows us to ***decompose a problem into several objects***
  - However there is *still a need for algorithms*

- The structure theorem tells us that any algorithm can be expressed using three elements
  - **Sequence**: an ordering of steps
  - **Selection**: making a decision
  - **Repetition**: looping

The Structure Theorem

- C# provides us with two selection structures:
  - **if/if-else**
  - **switch/case**

  and four loop structures:
  - **while**
  - **do-while**
  - **for**
  - **foreach**(considered in a future week)

The Structure Theorem

- Problem: The dangling-else...

```
if(condition)
    if(condition)
        true_statement;
    else
        false_statement;
```

```
if(condition)
    if(condition)
        true_statement;
else
    false_statement;
```

The compiler treats these as the same
Which is the correct version?

The Structure Theorem

- Problem: The dangling-else...

```
if(condition)
{
    if(condition)
        true_statement;
    else
        false_statement;
}
```

```
if(condition)
{
    if(condition)
        true_statement;
}
else
    false_statement;
```

Solve using braces

The Structure Theorem

- The while loop:

```
while (condition)   (i)
    statement;      (ii)
```

```
while (condition)   (i)
{
    statement;      (ii)
    ...
}
```

- Operation :
  i. Evaluate the condition, if false then leave the while loop;
  ii. Perform the statements in the loop (the loop body); and
  iii. Return to step (i);

The Structure Theorem

- The do-while loop:

```
do
    statement;      (i)
while(condition);   (ii)
```

```
do
{
    statement;      (i)
    ...
}while(condition);  (ii)
```

- Operation:
  i. Perform the statements in the loop (the loop body); and
  ii. Evaluate the condition, if false then leave the while loop;
  iii. Return to step (i);

Defining Classes

- Recall:
  - Object-oriented applications consist of a ***collection of cooperating objects***
  - An ***object is an instance of a class***, the ***class acts as a template for the object/s***
  - A ***class definition consists*** of
    - **Variables**
    - **Properties**
    - **Methods**
  - These represent the ***attributes and operations of an object's interface***

Defining Classes

- Class definition syntax:

```
[access_modifier ] class class_name
{
    [access_modifier ]class_member
    ...
}
```

- *access_modifier:*
  - `public`: accessible to all code
  - `private`: accessible only within the same class
  - `internal`: accessible only within the same assembly
  - `protected`
  - `protected internal`

Member default

Class default

} Examined with Inheritance

Defining Classes

- Creating objects:

```
class_name variable_name;
string line; // default constructor for empty string
variable_name = new class_name([parameters]);
line = new string('*', 50);
```

- or

```
class_name variable_name = new class_name([parameters]);
string line = new string('*', 50); //custom constructor

class_name variable_name = object;
string line = "**********************************";
string _50starline = new string(line);// copy constructor
```

Defining Classes

- **Creating operations – methods**:

```
[access_modifier ] return_type method_name ([parameter[, ...]])
{
    method_body
}
```

- **Calling** from another *method* in the same class:

```
method_name ([parameter[, ...]]);
OR
this.method_name ([parameter[, ...]]);
```

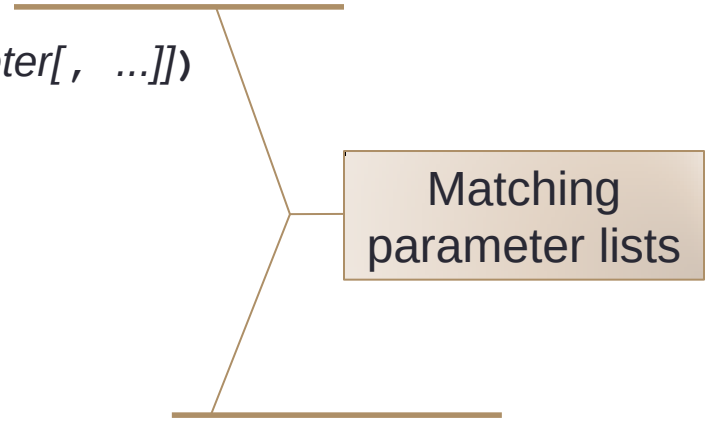
- Calling from another class using an object reference:

```
object_name.method_name ([parameter[, ...]]);
Console.WriteLine("{0} {1}", "Helen", "Smith");
```

Defining Classes

- ***Constructor methods are invoked immediately and automatically when an object is created***
  - Used for initialising an object
- Syntax:

```
[ access_modifier ] class class_name
{
    [ access_modifier ] class_name ([parameter[, ...]])
    {
            method_body
    }
    ...
}
```



- Called when creating an object:

```
class_name variable_name = new class_name ([parameter[, ...]]);
```

Defining Classes

- Creating attributes – instance variables:
  - Variables that are part of an object (instance) rather than part of class
- Syntax:

```
[access_modifier ] type name1[ = value][, name2[ = value][, ...]];
```

- Guideline:
  - Declare **`private`** and prefix name with underscore (_)
    - Needed to preserve **encapsulation**
    - Provide an interface with accessor and mutator methods or properties

Defining Classes

- Accessor and mutator methods provide a public interface to (**private**) attributes
  - Preserves **encapsulation**
  - Defines the ***abstraction/interface***
- **Accessor** method
  - Prefixed with 'get'
  - Used for **reading** data
- **Mutator** method
  - Prefixed with 'set'
  - Used for **storing/modifying** data

Defining Classes

```
private string _GivenName;

public string GetGivenName()
{
    return _GivenName;
}

public void SetGivenName(string value)
{
    _GivenName = value;
}
```

Example: Accessor and Mutator Method

Defining Classes

- Properties are offered by many modern object-oriented languages
  - Provide a more intuitive interface, e.g.,

```
sales.Count = sales.Count + 1;
```

  - instead of

```
sales.SetCount(sales.GetCount() + 1);
```

- Have optional get and set blocks:
  - A read/write property – defines both get and set blocks;
  - A read-only property – defines only the get block; and
  - A write-only property – defines only the set block.

Defining Classes

```
private string _GivenName;
public string GivenName
{
    get
    {
        return _GivenName;
    }
    set
    {
        _GivenName = value;
    }
}
```

Example: Property

Defining Classes

- The C# programming language also supports ***auto-implemented properties***, e.g.,

```
public string GivenName { get; set; }
```

Example: Auto-implemented Property

  - The *compiler automatically creates a hidden variable to store the data for the property*
  - *Can be changed to a manually implemented property later*

- Have optional get and set blocks:
  - A read/write property – indicate both get; and set;
  - A read-only property – get; private set;
  - A write-only property – private get; set;

Value Types and Reference Types

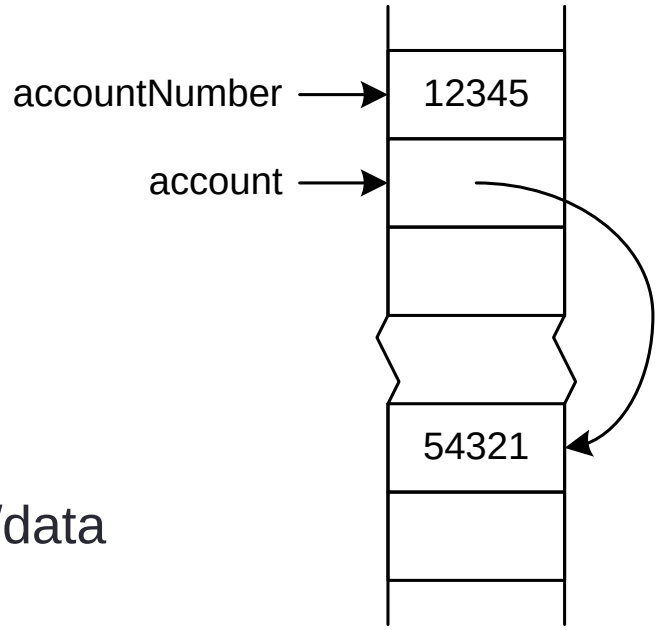
- **Value types** and ***reference types*** are important concept to grasp
- Value types

```
int accountNumber = 12345;
```

  - Store data directly
  - Simple types and user-defined structs
- Reference types

```
Account account = new Account(54321);
```

  - Store a memory reference to the object/data
  - Types defined using classes



Composite Formatting

- Until now we have only used simple format items
- Format item syntax:

  {*index* [,*field_width*][:*format*]}

  - *index* refers to parameters after the format string, starting from zero, i.e., {0}, {1}, ...
  - *field_width* represents the minimum field width
    - If the data is wider, the field is extended
  - *format* is passed to the relevant data type to indicate how it should be formatted (written)
- Examples:

```
Console.WriteLine("{0,-10}\t{1}({2})\t{3}, {4}", student.ID,
                  student.Mark, student.Grade, student.FamilyName, student.GivenName);
Console.WriteLine("The price is {0:c}", price);
```

Format

- C, c: for currency format
  Example 2000000.456m.ToString("C")
  Output: $2,000,000.46

- D, d for integer
  Example: 45678.ToString("D8")
  Output: 00045678

- E, e for E-Notation, default 6 decimal digits
  Example: 345678900000.ToString("E3)
  Output: 3.456E+011
  Example: 345678900000.ToString("e")
  Output: 3.456789e011

- F, f : precision specified that indicates the number of decimal digits
  Example: 3.7667892.ToString("F3")
  Output: 3.767
  // l4d.cs file

Format

- G, g : max number of digits
  Example: 65432.98765.ToString("G")
  Output: 65432.98765
  Example: 65432.98765.ToString("G7")
  Output: 65432.99
  Example: 65432.98765.ToString("G4")
  Output: 6.543E4

- N, n : embed comma and precision specifier sets the number of decimal digits
  Example 1000000.123m.ToString("N2")
  Output: 1,000,000.12

- X, x: Hex number
  Example: 950.ToString("x")
  Output: 3b6
  Example: 950.ToString("X6")
  Output: 0003B6

- *Ref: Fig4.17 Deitel pp.154*

Format

Console.WriteLine("Distance : {0,10} miles", 100) ;
Print output: Distance : 100 miles (7 leading spaces)

Console.WriteLine("Distance : {0,-10} miles", 100) ;
Print output: Distance : 100 miles (7 spaces after 100)

Console.WriteLine("Distance: {0, -12:E2} Mass: {1, -12:N} ", 100000000, 5000000);
Print output: Distance: 1.00E+008 Mass: 5,000,000.00

Console.WriteLine("Number {0} is black, number {1} is red");
Print output: Number {0} is black, number {1} is red

Notes
Console.WriteLine("Value: {0}", 12345.6789.ToString("N2"));
Console.WriteLine("Value: {0,2:n}", 12345.6789);
// l4e.cs
Ref: http://msdn2.microsoft.com/en-us/library/yf46atch.aspx

Format - Escape Sequence

\'	Single quote
\"	Double quote
\\	Backslash
\0	Null
\a	Alert
\b	Backspace
\f	Form feed
\n	new line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab

\\l4f.cs

```
string clockPath = "D:\\MyFiles\\Temp\\Assignment1.cs";
string clockPath = @ "D:\MyFiles\Temp\Assignment1.cs";
```

- *Ref: Fig3.16 Deitel pp.92-93*

String Format

- String.Format("{0:00000}", 15); // "00015"
- String.Format("{0:00000}", -15); // "-00015"
- String.Format("{0,5}", 15); // "   15"
- String.Format("{0,-5}", 15); // "15   "
- String.Format("{0,5:000}", 15); // " 015"
- String.Format("{0,-5:000}", 15); // "015 "
- String.Format("{0:#;minus #}", 15); // "15"
- String.Format("{0:#;minus #}", -15); // "minus 15"
- String.Format("{0:#;minus #;zero}", 0); // "zero"

// it similar to format in Excel in l4g.cs

String Format

```
String.Format("{0:+### ### ### ###}", 447900123456); // "+447 900 123 456"
String.Format("{0:##-####-####}", 8958712551);        // "89-5871-2551"
```

- **String Format for Double** // l4h.cs

```
// just two decimal places
String.Format("{0:0.00}", 123.4567);   // "123.46"
String.Format("{0:0.00}", 123.4);      // "123.40“
String.Format("{0:0.00}", 123.0);      // "123.00"
// max. two decimal places
String.Format("{0:0.##}", 123.4567);   // "123.46"
String.Format("{0:0.##}", 123.4);      // "123.4"
String.Format("{0:0.##}", 123.0);      // "123"
// at least two digits before decimal point
String.Format("{0:00.0}", 123.4567);   // "123.5"
String.Format("{0:00.0}", 23.4567);    // "23.5"
String.Format("{0:00.0}", 3.4567);     // "03.5"
String.Format("{0:00.0}", -3.4567);    // "-03.5"
```

String Format

- String.Format("{0:0,0.0}", 12345.67); // "12,345.7"
  String.Format("{0:0,0}", 12345.67); // "12,346"
- String.Format("{0:0.0}", 0.0); // "0.0"
  String.Format("{0:0.#}", 0.0); // "0"
  String.Format("{0:#.0}", 0.0); // ".0"
  String.Format("{0:#.#}", 0.0); // ""
- **Align numbers with spaces**

```
String.Format("{0,10:0.0}", 123.4567);  // "     123.5"
String.Format("{0,-10:0.0}", 123.4567); // "123.5     "
String.Format("{0,10:0.0}", -123.4567); // "    -123.5"
String.Format("{0,-10:0.0}", -123.4567); // "-123.5    "
```

String Format

```
String.Format("{0:0.00;minus 0.00;zero}", 123.4567);        // "123.46"
String.Format("{0:0.00;minus 0.00;zero}", -123.4567);       // "minus 123.46"
String.Format("{0:0.00;minus 0.00;zero}", 0.0);             // "zero"

String.Format("{0:my number is 0.0}", 12.3);                // "my number is 12.3"
String.Format("{0:0aaa.bbb0}", 12.3);                       // "12aaa.bbb3“
```

- **String Format for Int**

```
String.Format("{0:00000}", 15);                             // "00015"
String.Format("{0:00000}", -15);                            // "-00015“
```

- **Align number to the right or left**

```
String.Format("{0,5}", 15);                                 // " 15"
 String.Format("{0,5}", 15);                                // "15 "
String.Format("{0,5:000}", 15);                             // " 015"
String.Format("{0,-5:000}", 15);                            // "015 “
```

//l4j.cs

DateTime Format

- // create date time 2008-03-09 16:05:07.123
- DateTime dt = new DateTime(2008, 3, 9, 16, 5, 7, 123);

```
String.Format("{0:y yy yyy yyyy}", dt);  // "8 08 008 2008" year
String.Format("{0:M MM MMM MMMM}", dt);      // "3 03 Mar March" month
String.Format("{0:d dd ddd dddd}", dt);              // "9 09 Sun Sunday" day
 String.Format("{0:h hh H HH}", dt);                 // "4 04 16 16" hour 12/24
String.Format("{0:m mm}", dt);                       // "5 05" minute
String.Format("{0:s ss}", dt);                       // "7 07" second
String.Format("{0:f ff fff ffff}", dt);              // "1 12 123 1230" sec.fraction
String.Format("{0:F FF FFF FFFF}", dt);              // "1 12 123 123" without zeroes
String.Format("{0:t tt}", dt);                       // "P PM" A.M. or P.M.
String.Format("{0:z zz zzz}", dt);                   // "-6 -06 -06:00" time zone
```

//l4k.cs

DateTime Format

- // date separator in german culture is "." (so "/" changes to ".")

  String.Format("{0:d/M/yyyy HH:mm:ss}", dt); // "9/3/2008 16:05:07" - english (en-US)
  String.Format("{0:d/M/yyyy HH:mm:ss}", dt); // "9.3.2008 16:05:07" - german (de-DE)

- // month/day numbers without/with leading zeroes

  String.Format("{0:M/d/yyyy}", dt); // "3/9/2008"

  String.Format("{0:MM/dd/yyyy}", dt); // "03/09/2008" // day/month names
  String.Format("{0:ddd, MMM d, yyyy}", dt); // "Sun, Mar 9, 2008"

  String.Format("{0:dddd, MMMM d, yyyy}", dt); // "Sunday, March 9, 2008"

  String.Format("{0:MM/dd/yy}", dt); // "03/09/08"

  String.Format("{0:MM/dd/yyyy}", dt); // "03/09/2008"

DateTime Format

Specifier	DateTimeFormatInfo	property Pattern value (for en-US culture)
t	ShortTimePattern	h:mm tt
d	ShortDatePattern	M/d/yyyy
T	LongTimePattern	h:mm:ss tt
D	LongDatePattern	dddd, MMMM dd, yyyy
f	(combination of D and t)	dddd, MMMM dd, yyyy h:mm tt
F	FullDateTimePattern	dddd, MMMM dd, yyyy h:mm:ss tt
g	(combination of d and t)	M/d/yyyy h:mm tt
G	(combination of d and T)	M/d/yyyy h:mm:ss tt
m, M	MonthDayPattern	MMMM dd
y, Y	YearMonthPattern	MMMM, yyyy
r, R	RFC1123Pattern	ddd, dd MMM yyyy HH':'mm':'ss 'GMT' (*)
s	SortableDateTimePattern	yyyy'-'MM'-'dd'T'HH':'mm':'ss (*)
u	UniversalSortableDateTimePattern	yyyy'-'MM'-'dd HH':'mm':'ss'Z' (*)

(*) = culture independent

DateTime Format

- String.Format("{0:t}", dt); // "4:05 PM" ShortTime
- String.Format("{0:d}", dt); // "3/9/2008" ShortDate
- String.Format("{0:T}", dt); // "4:05:07 PM" LongTime
- String.Format("{0:D}", dt); // "Sunday, March 09, 2008" LongDate
- String.Format("{0:f}", dt); // "Sunday, March 09, 2008 4:05 PM" LongDate+ShortTime

- String.Format("{0:F}", dt); // "Sunday, March 09, 2008 4:05:07 PM" FullDateTime
  String.Format("{0:g}", dt); // "3/9/2008 4:05 PM" ShortDate+ShortTime
  String.Format("{0:G}", dt); // "3/9/2008 4:05:07 PM" ShortDate+LongTime
  String.Format("{0:m}", dt); // "March 09" MonthDay

- String.Format("{0:y}", dt); // "March, 2008" YearMonth
- String.Format("{0:r}", dt); // "Sun, 09 Mar 2008 16:05:07 GMT" RFC1123
  String.Format("{0:s}", dt); // "2008-03-09T16:05:07" SortableDateTime
  String.Format("{0:u}", dt); // "2008-03-09 16:05:07Z" UniversalSortableDateTime

// l4l.cs

Text Alignment

```
Console.WriteLine("-----------------------------");
Console.WriteLine("First Name | Last Name  | Age");
Console.WriteLine("-----------------------------");
Console.WriteLine(String.Format("{0,-10} | {1,-10} | {2,5}", "Bill", "Gates", 51));
Console.WriteLine(String.Format("{0,-10} | {1,-10} | {2,5}", "Edna", "Parker", 114));
Console.WriteLine(String.Format("{0,-10} | {1,-10} | {2,5}", "Johnny", "Depp", 44));
Console.WriteLine("-----------------------------");
```

- Output string: // l4m.cs

```
-------------------------------------
First Name | Last Name   | Age
-------------------------------------
Bill       | Gates       |   51
Edna       | Parker      | 114
Johnny     | Depp        |   44
-------------------------------------
```

Indent String with Spaces

```
public static string Indent(int count)
 {
    return "".PadLeft(count);
}
Console.WriteLine(Indent(0) + "List");
Console.WriteLine(Indent(3) + "Item 1");
Console.WriteLine(Indent(6) + "Item 1.1");
Console.WriteLine(Indent(6) + "Item 1.2");
Console.WriteLine(Indent(3) + "Item 2");
Console.WriteLine(Indent(6) + "Item 2.1");

// l4n.cs
```

Output

```
List
   Item 1
      Item 1.1
      Item 1.2
   Item 2
      Item 2.1
```

Logic Errors and Debugging

- A logic ***error*** occurs when the program
  - Compiles correctly
  - Runs successfully
  - Produces incorrect output
- You will often ***experience logic errors***
  - Many are simple, or become simple as you *gain experience in software development*
  - Others require you to *develop a technique to diagnosing and correcting the logic error*
- **Debuggers** are the primary tool used for finding and diagnosing logic errors

Summary

- Session 02. Control structures, classes, and data types
  - Objectives
  - The Structure Theorem
  - Defining Classes
  - Value Types and Reference Types
  - Composite Formatting
  - Logic Errors and Debugging

Summary

- Training Videos:
  - VS: Adding and Renaming Classes
  - C#: Defining Class Members
  - C#: Constructor Basics
  - C#: Value Types and Reference Types
  - VS: Working with Logic Errors
  - T&T: Creating a Project using Multiple Provided Files