

主控台應用程式

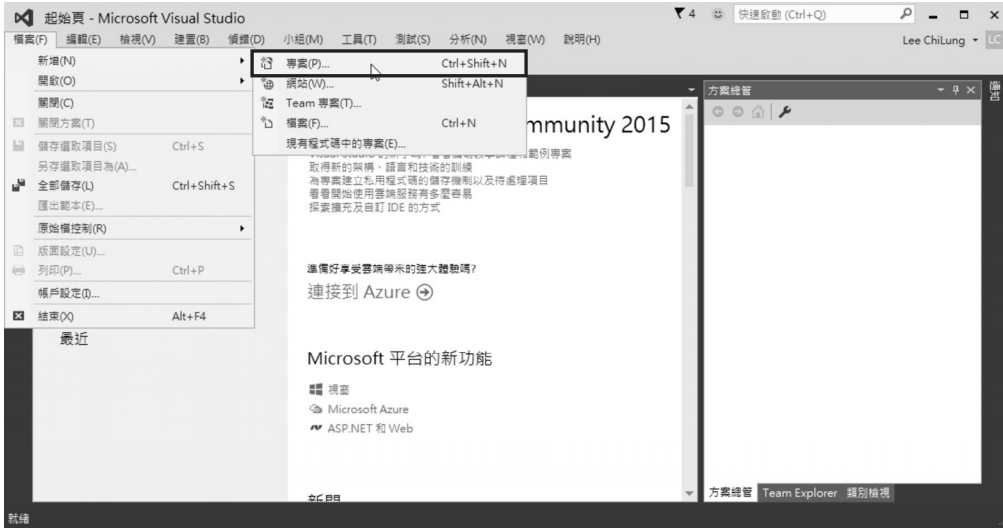
2-1 | 主控台

如果讀者使用過 MS-DOS 或是 Linux 作業系統的話，想必對於命令行（Command Line）並不陌生，主控台應用程式便是設計在類似命令行底下作業的模式，而主控台（Console）則是 System 命名空間（NameSpace）底下的類別之一，用來處理主控台應用程式裡輸出或輸入等各種動作，如果讀者對以上一些專有名稱並不熟悉也沒關係，其中重要的觀念在本書中都會加以說明！

相較於我們習慣的視窗圖型介面，主控台模式在顯示上是純粹的文字構成，或許讀者一開始會覺得不是很友善，但要記得「萬丈高樓平地起」，在我們把所學運用到視窗模式之前，觀念的部份最好還是先從基礎開始學習才比較扎實。本書接下來的單元小節，大部分都將以主控台模式先介紹基本觀念，再以視窗程式方式來應用實作，接下來我們就來看看如何新增一個主控台應用程式吧。

2-2 | 主控台工作環境簡介

執行「Visual Studio 2015」程式，進入 Visual Studio Community 2015 的開發環境，然後執行功能表的【檔案/新增/專案】選項。

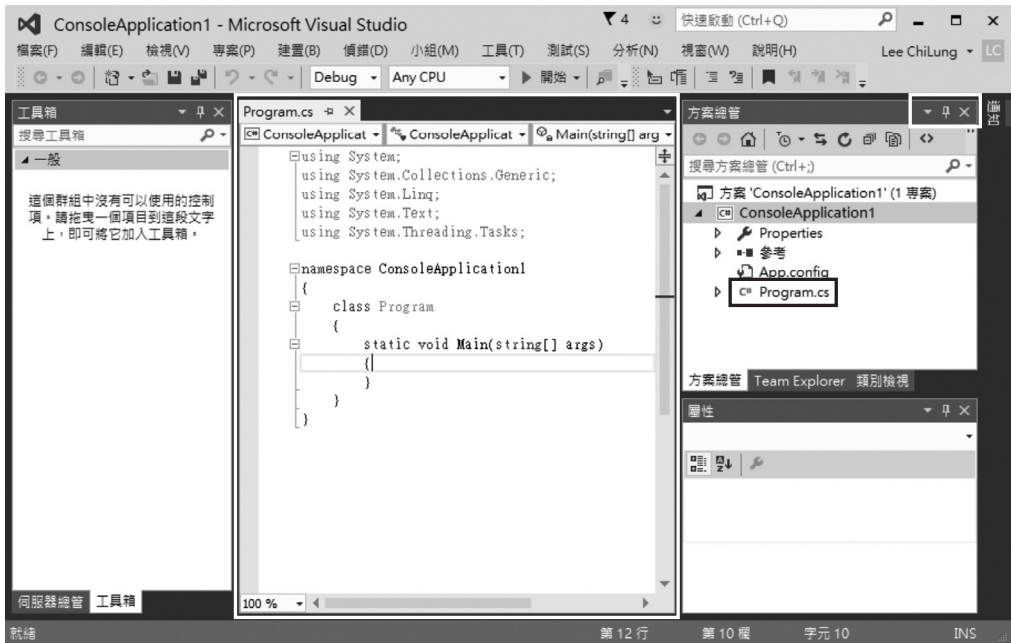


在跳出的「新增專案」視窗中，選擇「Visual C#」項目，再選擇「主控台應用程式」項目，接著取一個自己可以方便辨識的名稱。專案的名稱由使用者自行命名，此處預設的名稱為「ConsoleApplication1」，存檔的位置也是可以自由選擇，然後請務必勾選「為方案建立目錄」選項，Visual C# 便會為該專案建立個別的資料夾，將所有的檔案置於資料夾中，最後再按下「確定」鈕，即完成主控台應用程式的新增。

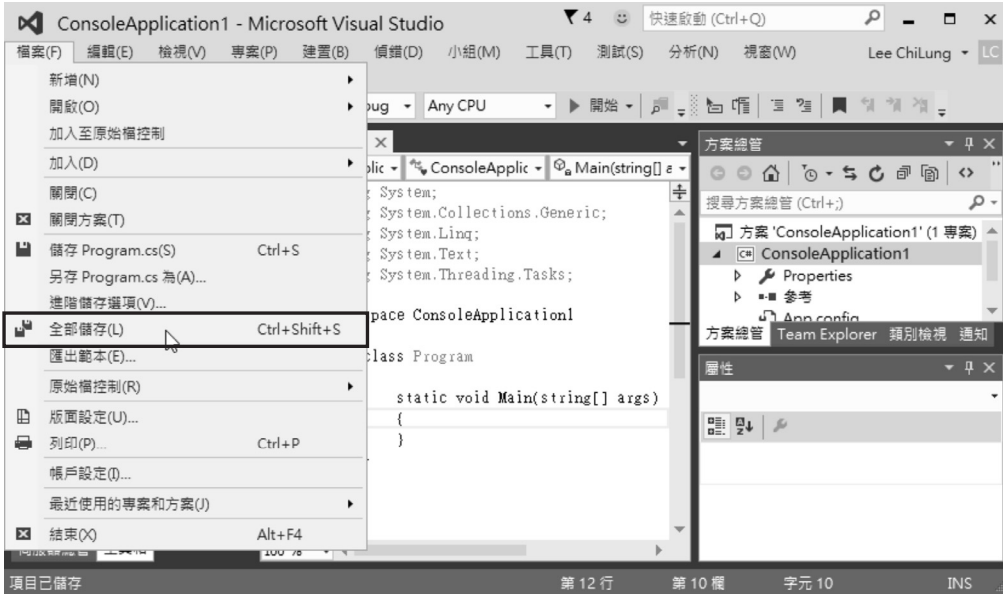


完成上述步驟後，畫面便會變成主控台應用程式的工作環境了。畫面中間有「程式碼編輯區」，是我們要編輯程式碼的地方，另外還有「方案總管」、「工具箱」、「屬性視窗」等各個區塊。每個區塊的右上角都有「視窗位置」、「隱藏/顯示」、「關閉」等 3 個按鈕。

「方案總管」、「工具箱」、「屬性視窗」這 3 個區塊都有其個別的功能，「方案總管」內會顯示這個專案底下包含的程式碼及資訊，例如：讀者可以看到圖中「程式碼編輯區」，顯示的便是方案總管下 Program.cs，這個 Visual C# 新專案預設產生的程式碼內容；而「工具箱」及「屬性視窗」，雖然在主控台應用程式下不常用到，但我們會在後續的單元中使用到。



在一切就緒之後，大家可能會迫不及待想要開始寫第一個程式，在寫程式之前，還是先練習將檔案存檔，養成先存檔的好習慣，以避免當電腦突然發生狀況時，所有心血付之一炬，請從檔案功能表中選擇「全部儲存」選項。



2-3 | 第一個主控台程式的編譯與執行

Visual C# 會自動幫我們新建立的主控台應用程式，設定好一些基本常用的東西，可以從方案總管中看到詳情，像是剛剛看到的 Program.cs 裡面的內容，便是一個它預先弄好基本架構的例子。

不過在讀者試著去看懂這段程式碼之前，如果您是個程式初學者，筆者想先從更簡單一點的例子開始講起，請看下面這段程式碼。

🔊 參考檔案：2.3-1.sln

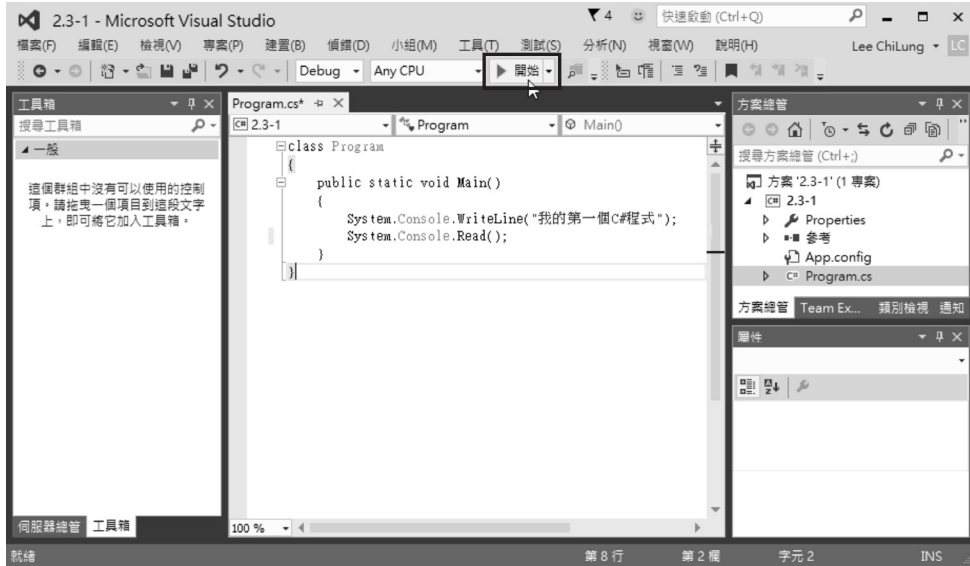
```
class Program
{
    public static void Main()
    {
        System.Console.WriteLine("我的第一個 C# 程式");
        System.Console.Read();
    }
}
```

跟預設的程式碼架構比起來，這段程式碼顯然更加簡單，筆者說明一下這段程式碼的意思。首先在 C# 語言中，所有的程式碼都要寫在一個類別（Class）或是結構（Structure）內，此例中我們的程式碼便是在 Program 這個類別當中，接下來 Main() 這個方法（Method），則是所有 C# 程式的進入點，Main 前面的幾個單詞的意義以及何謂「方法」，我們也會在後面介紹到，注意到這裡在類別與方法中的內容，都要寫在 { } 大括號之間。最後兩行用分號（;）結尾的程式碼敘述，則分別是使用 System 這個命名空間內 Console 類別中的 WriteLine 方法印出一行文字，以及用 Read 方法讀取使用者輸入的資料，此處加入 Read 方法是用來讓畫面停住，否則程式執行畫面會一閃即逝。

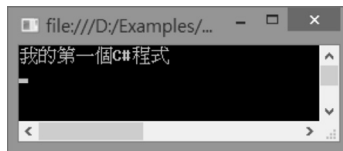
將以上程式碼鍵入程式碼編輯視窗（注意！在 C# 語言中大小寫是有差別的），在輸入的過程中，讀者應該會發現 Visual C# 會在您輸入的同時列出建議的程式指令清單，這項方便的功能微軟稱之為 IntelliSense，相信讀者在之後的使用中，會發現它非常方便。如下圖所示，在輸入「S」字母的時候，會出現 S 開頭的程式指令清單。



完成輸入之後，便可以開始編譯並執行這個程式了！只要按下工具列上三角形開始符號或是按鍵盤的「F5」鍵，Visual C# 便會開始編譯並執行程式！



如下圖所示，程式執行結果簡單的列出一行文字「我的第一個 C# 程式」，此時就完成了第一個 C# 程式！



接下來我們把程式碼稍微修改一下，來看看跟剛剛的程式有什麼不一樣。

參考檔案：2.3-12.sln

```
using System;

namespace MyLib
{
    class Program
    {
        public static void Main()
        {
            Console.WriteLine("我的第一個 C#程式");
            Console.Read();
        }
    }
}
```

6-1 | 迴圈結構 for

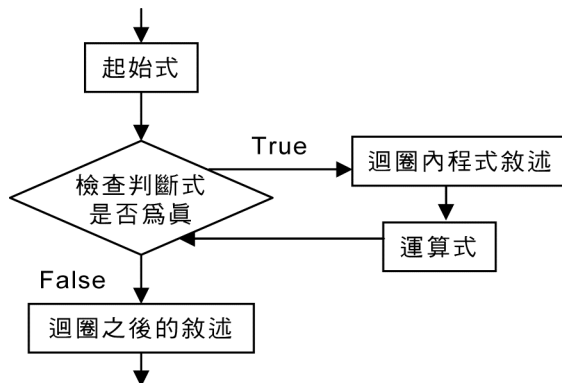
for 迴圈藉由「起始式（initial statement）」、「判斷式（conditional statement）」和「運算式（execution statement）」，來控制迴圈的執行與結束。

起始式和運算式，都可以是一行程式敘述。判斷式的用法則與 if 類似，for 迴圈會依據判斷式的是否滿足，來控制程式執行的次數。

for 迴圈語法如下：

```
for( initial statement; conditional statement; execution statement){
    program statements;
    ...
}
```

for 迴圈的流程圖表示法如下：



就最簡單的迴圈用途來說，for 迴圈中「起始式」、「判斷式」和「運算式」的用法，通常是如下表所述：

起始式	用來初始化一個或多個變數的值。
判斷式	運用此變數的真假值來判斷是否進入程式區塊。
運算式	對此變數做一些運算，例如：遞增或遞減運算。

參考下面的程式碼：

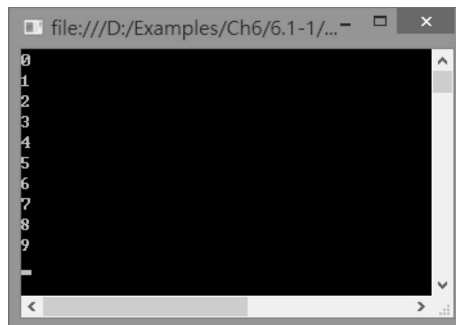
參考檔案：6.1-1.sln

```
using System;
class Program
{
    public static void Main()
    {
        int i;
        for(i = 0; i < 10; i++){
            Console.WriteLine(i);
        }
        Console.Read();
    }
}
```

這段簡單的程式執行結果，會在螢幕上依序印出 0~9 的數字。我們用這個例子來觀察這段程式碼的執行流程，並比照一下 for 迴圈中的「起始式」、「判斷式」和「運算式」與程式碼的對應關係。

當程式第一次執行至 for 迴圈時，起始式 `i=0` 會把變數 `i` 的值初始化為 0，接下來判斷式 `i<10` 的要求是變數 `i` 要小於 10，而這時的 `i`（值為 0）會符合要求，因此會進入迴圈中的程式區塊，執行 `WriteLine` 方法印出 `i` 的值，最後再執行運算式 `i++`，將變數 `i` 加上 1，`i` 的值變成 1，最後回到迴圈的開頭，此時程式不會再重新執行一次起始式的敘述，而是會直接比對判斷式，決定是否再次進入迴圈中的程式區塊，如此一直執行下去，直到某次判斷時，變數 `i` 不符合判斷式 `i<10` 為止，程式就會離開整個迴圈區塊。

在 for 迴圈中，起始式、判斷式和運算式都可以為空白，但是中間分號仍然要寫出來，否則會造成語法錯誤。以下為這段程式碼的執行結果：



再參考下面這個計算 $1+2+\cdots+10$ 級數（從 1 加到 10）的範例程式碼：

🔊 參考檔案：6.1-2.sln

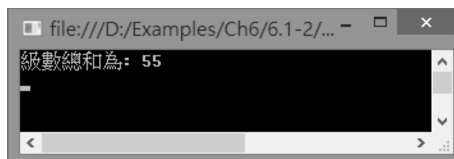
```
using System;

class Program
{
    public static void Main()
    {
        int i, sum = 0;
        for(i = 0; i <= 10; i++)
            sum += i;

        Console.WriteLine("級數總和為: " + sum);
        Console.Read();
    }
}
```

注意到這個例子中，`for` 迴圈的內容沒有加上大括號，理由和前一單元 `if` 判斷結構的原因一樣，在這類結構中，當我們只有一行程式敘述時，大括號通常是可以省略的。

這個迴圈的「起始式」、「判斷式」和「運算式」與前一段程式碼大致相同，唯一的差異是判斷式 `i <= 10` 的判斷範圍多了一個等號，而我們在這個例子中使用了另一個變數 `sum` 來儲存總和，所以在每次迴圈中，變數 `i` 的值都會被加總進變數 `sum` 當中，而運算式 `i++` 告訴我們變數 `i` 每次增加 1，所以這段程式碼相當於從 $0+1+\dots$ 一直加到 10（最後一次符合判斷式）為止。最後會在螢幕上列出計算結果，「級數總和為: 55」，如下圖所示：





程式範例：階乘計算主控台程式

學習重點：用 for 迴圈來計算階乘／參考檔案：6.1-3.sln

🔊 程式設計目標

撰寫一個程式，可以讓使用者自行輸入 1 個數字，程式會計算這個數字的階乘結果，例如：使用者輸入 5，則程式會計算出 $1 \times 2 \times 3 \times 4 \times 5 = 120$ ，並印出結果 120。

🔊 程式碼撰寫

```
01 using System;
02
03 class Program
04 {
05     public static void Main()
06     {
07         Console.Write("請輸入欲計算階乘的數字: ");
08         int number;
09         number = int.Parse(Console.ReadLine());
10
11         double sum = 1;
12         for (; number > 1; --number)
13             sum *= number;
14
15         Console.WriteLine("階乘結果為: " + sum);
16         Console.Read();
17     }
18 }
```

🔊 程式碼解說

因為階乘的數字是要讓使用者指定的，所以程式一開始會先印出提示輸入的文字並宣告一整數變數 `number`，用來儲存使用者輸入的階乘數字，最後用 `ReadLine` 方法，讀進使用者輸入的值並儲存到剛剛宣告的變數 `number` 即可。

```
Console.Write("請輸入欲計算階乘的數字: ");
int number;
number = int.Parse(Console.ReadLine());
```

接著用 `double` 型態宣告變數 `sum`，因為階乘的結果常常會很大，所以在這裡使用 `double` 型態會比較適合（數字超過支援的範圍依然會溢位，但一般情況應該已比整數型態夠用了！）。注意由於後面的迴圈中要進行連續乘法，所以 `sum` 的起始值應該要設為 1，而非前面級數範例的 0，否則在階乘中會導致階乘結果也變成 0。請注意這裡迴圈中「起始式」、「判斷式」和「運算式」的設定方式與之前稍有不同，我們直接使用剛剛使用者輸入的階乘數遞減來控制迴圈，故起始式的部份不需要再另外設定起始值，運算式 `--number` 就是遞減的部份，而迴圈的執行條件 `number > 1`，則設定在判斷式中。如此設定，`number` 變數就會依次遞減直到 1 為止，而迴圈執行時，則會每次把當時 `number` 的值與 `sum` 相乘，最後就可以得到階乘的計算結果。

```
double sum = 1;
for (; number > 1; --number)
    sum *= number;
```

程式最後只要將儲存在 `sum` 變數當中的結果，用 `WriteLine` 方法印出來即可。

```
Console.WriteLine("階乘結果為: " + sum);
```

🔊 執行結果



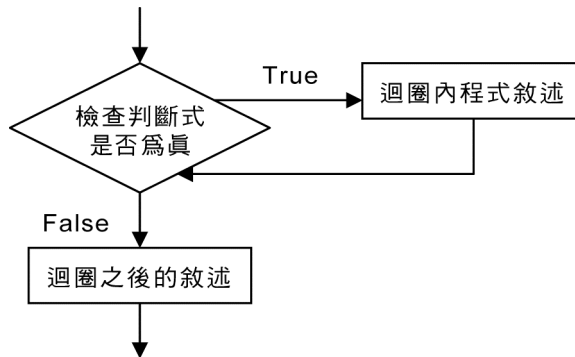
6-2 | 迴圈結構 while

`while` 迴圈的結構與 `for` 迴圈相似，但是比 `for` 迴圈更簡單，`while` 迴圈並沒有「起始式」與「運算式」的區塊，只是單純的藉由「判斷式」的真假，來決定是否繼續執行迴圈內的程式敘述。

`while` 迴圈的語法如下：

```
while( conditional statement){  
    program statements;  
    ...  
}
```

流程圖的表示法如下：



由於 **while** 迴圈並沒有起始式與運算式的區塊，當需要起始式與運算式的計算時，就需要在迴圈的前面或裡面加入一些程式敘述。若要將 **for** 迴圈的觀念，用 **while** 迴圈表示出來，可以將 **for** 迴圈的起始式放置在 **while** 迴圈區塊之前；運算式放置在 **while** 迴圈區塊結束前的最後一個程式敘述。參考下面的比較範例：

在本單元一開始介紹 **for** 迴圈時，我們用一個印出 0~9 的小程式，示範了在 **for** 迴圈當中「起始式」、「判斷式」和「運算式」的設定方式，而在這個簡單的例子當中，起始式是 **i=0**、判斷式為 **i<10**、運算式則是 **i++**，程式碼修改如下：

```
int i;  
for(i = 0; i < 10; i++){  
    Console.WriteLine(i);  
}
```

而同樣的程式應用，如果用 **while** 迴圈改寫，起始式和運算式的位置就得改到適當的位置，因為 **while** 迴圈的設定，只重視判斷迴圈是否繼續的「判斷式」，故我們若有設定起始式或運算式的需求，就得自己填在適當的位置，實際上 **for** 迴圈能做的事，**while** 迴圈也都能做到，反之亦然。這個 **while** 迴圈範例就與上面 **for** 迴圈的範例之執行結果完全相同。

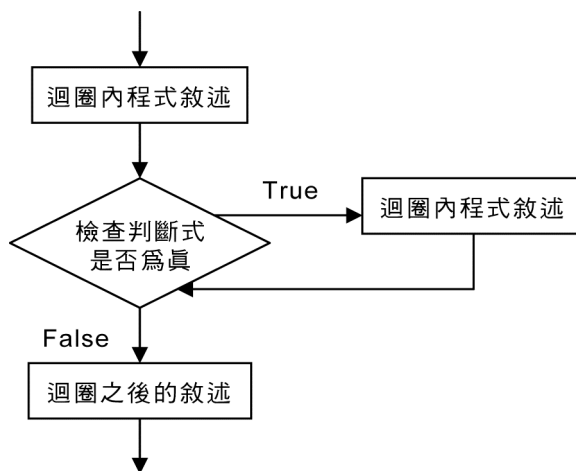
```
int i;  
  
i=0; //起始式  
while(i < 10){  
    Console.WriteLine(i);  
    i++; //運算式  
}
```

除了 while 迴圈本身以外，while 迴圈還有一個常見的變形「do…while 迴圈」。do…while 迴圈和 while 迴圈的主要差別在於迴圈內的程式敘述，不管如何都會先執行一次，執行一次後再根據判斷式的成立與否，決定是否再繼續進入迴圈以執行程式碼。

do…while 迴圈的使用語法如下：

```
do{  
    program statements;  
    ...  
} while(conditional statement);
```

do…while 迴圈的流程圖表示法如下：



筆者這邊要再說明一次，其實 do…while 迴圈與 while 迴圈最大的不同之處，是在 do…while 迴圈內的程式敘述至少會被執行一次，而接下來的動作就與 while 迴圈相同了。我們參考下面的程式碼範例：

15-4 | 專題應用 4：龜兔賽跑預測遊戲視窗程式



專題應用 4：龜兔賽跑預測遊戲視窗程式

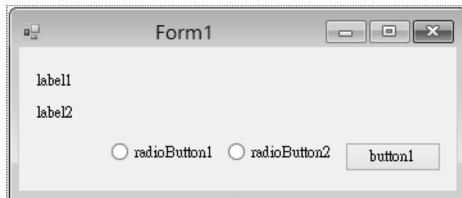
學習重點：視窗程式的類別綜合練習／參考檔案：15.4-1.sln

🔊 程式設計目標

龜兔賽跑是一則著名的寓言故事，我們要建立一個視窗程式，設定 2 個標籤用來代表故事的兩個主角，根據兩者的特色設定兩個不同的類別，其中烏龜是勤奮的跑者，平均移動速度慢但持續前進，而兔子平均移動速度快，但每移動一步就會隨機休息一段時間，先從畫面左方移到右方的跑者為勝利，使用者可以猜測哪一方會獲得勝利。

🔊 表單配置

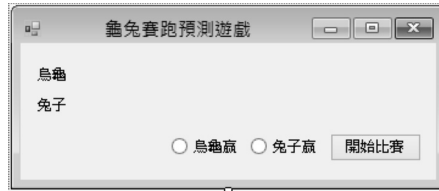
新增 2 個 Label、2 個 RadioButton 及 1 個 Button，大約配置如下：



然後依照以下表格更改各物件屬性值：

物件	Name 屬性	Text 屬性
label1	turtleLabel	烏龜
label2	rabbitLabel	兔子
radioButton1	turtleButton	烏龜贏
radioButton2	rabbitButton	兔子贏
button1	startButton	開始比賽
timer1	*可以不用變更	*無此項屬性
Form1	*不用變更	龜兔賽跑預測遊戲

更改後的表單配置大約如下：



🔊 程式碼撰寫

► [AnimalRunner.cs]

```
01 using System;
02
03 namespace RunnerSpace
04 {
05     class AnimalRunner
06     {
07         protected int base_speed;
08         protected int extra_speed;
09         protected Random rnd;
10
11         public virtual int step(){
12             return base_speed + rnd.Next(0, extra_speed);
13         }
14     }
15 }
```

► [Turtle.cs]

```
01 using System;
02
03 namespace RunnerSpace
04 {
05     class Turtle : AnimalRunner
06     {
07         public Turtle(int bs, int es) {
08             this.base_speed = bs;
09             this.extra_speed = es;
10             this.rnd = new Random();
11         }
12     }
13 }
```

► [Rabbit.cs]

```
01 using System;
02
03 namespace RunnerSpace
04 {
05     class Rabbit : AnimalRunner
06     {
07         private int break_time;
08
09         public Rabbit(int bs, int es){
10             this.base_speed = bs;
11             this.extra_speed = es;
12             this.rnd = new Random();
13             this.break_time = 0;
14         }
15
16         public override int step(){
17             if (this.break_time == 0){
18                 this.break_time = rnd.Next(1, 5);
19                 return base_speed + rnd.Next(0, extra_speed);
20             }
21             else {
22                 --this.break_time;
23                 return 0;
24             }
25         }
26     }
27 }
```

► [Program.cs]

```
01 private Turtle turtle;
02 private Rabbit rabbit;
03
04 private void Form1_Load(object sender, EventArgs e)
05 {
06     reset();
07 }
08
09 private void reset() {
10     this.turtle = new Turtle(4, 3);
11     this.rabbit = new Rabbit(15, 5);

```



```

12     turtleLabel.Left = 12;
13     rabbitLabel.Left = 12;
14     startButton.Enabled = true;
15     timer1.Stop();
16 }
17
18 private void timer1_Tick(object sender, EventArgs e)
19 {
20     turtleLabel.Left += this.turtle.step();
21     rabbitLabel.Left += this.rabbit.step();
22     if (turtleLabel.Left > 320) {
23         reset();
24         if (turtleButton.Checked == true)
25             MessageBox.Show("烏龜贏了! 恭喜您猜對了!");
26         else
27             MessageBox.Show("烏龜贏了! 但您猜錯了");
28     }
29     else if (rabbitLabel.Left > 320) {
30         reset();
31         if (rabbitButton.Checked == true)
32             MessageBox.Show("兔子贏了! 恭喜您猜對了!");
33         else
34             MessageBox.Show("兔子贏了! 但您猜錯了");
35     }
36 }
37
38 private void startButton_Click(object sender, EventArgs e)
39 {
40     startButton.Enabled = false;
41     timer1.Start();
42 }

```

🔊 程式碼解說

我們將所有的賽跑角色，都定義在 `RunnerSpace` 命名空間當中，先定義最基礎的 `AnimalRunner` 類別，並設定三個 `protected` 屬性，包含基礎速度 `base_speed`、額外速度 `extra_speed` 以及亂數物件 `rnd`。

```

namespace RunnerSpace
{
    class AnimalRunner
    {

```

```
protected int base_speed;  
protected int extra_speed;  
protected Random rnd;
```

以 `base_speed` 和 `extra_speed` 為基礎，我們定義 `step` 方法，會回傳該跑者移動一次的單位距離，其中 `base_speed` 是保證最小值，而亂數物件則會在 0 到 `extra_speed` 之間隨機產生一個數字，將 `base_speed` 與產生的隨機數字加起來後回傳即可。此方法在宣告時我們加上了 `virtual` 保留字，以留待之後的類別覆載。

```
public virtual int step(){  
    return base_speed + rnd.Next(0, extra_speed);  
}  
}  
}
```

`Turtle` 類別直接繼承 `AnimalRunner` 類別，由於烏龜的移動特色較單純，故繼承之後直接多載一個多引數建構子，方便我們之後建立實際物件就可以了。

```
namespace RunnerSpace  
{  
    class Turtle : AnimalRunner  
    {  
        public Turtle(int bs, int es) {  
            this.base_speed = bs;  
            this.extra_speed = es;  
            this.rnd = new Random();  
        }  
    }  
}
```

`Rabbit` 類別一開始也一樣繼承 `AnimalRunner` 類別，但配合兔子會隨機休息的特色，我們加上了一個屬性 `break_time`，用來記錄剩餘休息時間，多載的建構子中也額外加上該屬性的初始化數據。

```
namespace RunnerSpace  
{  
    class Rabbit : AnimalRunner  
    {  
        private int break_time;
```

```

public Rabbit(int bs, int es){
    this.base_speed = bs;
    this.extra_speed = es;
    this.rnd = new Random();
    this.break_time = 0;
}

```

接著要覆載從 **AnimalRunner** 類別繼承下來的 **step** 方法，加上 **override** 保留字開始重新定義 **step** 方法。當剩餘休息時間為 0 時，代表要移動了，程式就會重新產生 1~5 單位的休息時間並正常回傳移動一次的距離，若剩餘休息時間不為 0，代表正在休息，則將休息時間減少 1 個單位，並回傳 0 表示不移動。

```

public override int step(){
    if (this.break_time == 0){
        this.break_time = rnd.Next(1, 5);
        return base_speed + rnd.Next(0, extra_speed);
    }
    else {
        --this.break_time;
        return 0;
    }
}
}

```

主程式的部份設定兩個類別層級變數，分別是代表烏龜的 **turtle** 和代表兔子的 **rabbit**。設定 **reset** 方法用來初始化這兩個賽跑者的變數，並將對應的標籤位置移到初始位置、停止計時器計時並允許使用者按鈕開始比賽。

```

private Turtle turtle;
private Rabbit rabbit;

private void Form1_Load(object sender, EventArgs e)
{
    reset();
}

```

```
private void reset() {  
    this.turtle = new Turtle(4, 3);  
    this.rabbit = new Rabbit(15, 5);  
    turtleLabel.Left = 12;  
    rabbitLabel.Left = 12;  
    startButton.Enabled = true;  
    timer1.Stop();  
}
```

我們將計時器 `timer1` 的間隔屬性設定為 `500(ms)` 左右，而計時器的 `Tick` 事件做如下設定：每次 `Tick` 事件發生，代表時間過了一單位，故程式會呼叫兩個跑者的 `step` 方法，並將對應標籤依據回傳的數字往右移動。

```
private void timer1_Tick(object sender, EventArgs e)  
{  
    turtleLabel.Left += this.turtle.step();  
    rabbitLabel.Left += this.rabbit.step();  
}
```

移動完之後就先檢查烏龜標籤距離畫面左方是否已經超過 `320` 距離單位了，此一是依照設定的視窗大小而定，若是的話就重新初始化遊戲，並用一組 `if...else...` 判斷使用者的猜測是否正確並秀出相應的對話方塊訊息；反之若是兔子標籤已超過距離畫面左方 `320` 單位的話亦然。

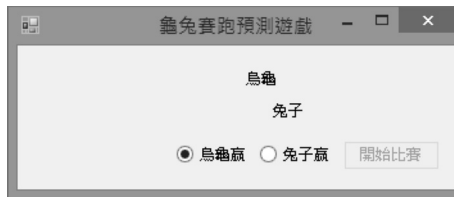
```
If (turtleLabel.Left > 320) {  
    reset();  
    if (turtleButton.Checked == true)  
        MessageBox.Show("烏龜贏了! 恭喜您猜對了!");  
    else  
        MessageBox.Show("烏龜贏了! 但您猜錯了");  
}  
else if (rabbitLabel.Left > 320) {  
    reset();  
    if (rabbitButton.Checked == true)  
        MessageBox.Show("兔子贏了! 恭喜您猜對了!");  
    else  
        MessageBox.Show("兔子贏了! 但您猜錯了");  
}  
}
```

主程式最後只要將「開始比賽」按鈕連結到計時器 `timer1` 開始計時，以及暫時讓按鈕本身不能被點擊即可。

```
private void startButton_Click(object sender, EventArgs e)
{
    startButton.Enabled = false;
    timer1.Start();
}
```

執行結果

猜測「烏龜贏」，然後按下「開始比賽」按鈕。



最後結果為「兔子贏」，程式會告知使用者「您猜錯了」。

