# 第 12 章　打磚塊遊戲

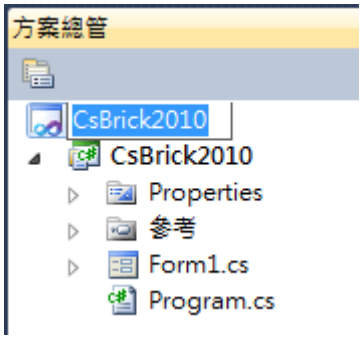
**簡介：**

打磚塊遊戲的主要功能包括：一個碰到障礙會反彈的球，一個可以移動的球拍，以及許多被球碰撞後會消失的磚塊物件。此專案中將介紹如何在程式執行階段，用程式動態的產生大量的磚塊物件，同時設計檢查物件碰撞以及反彈行為的程式。

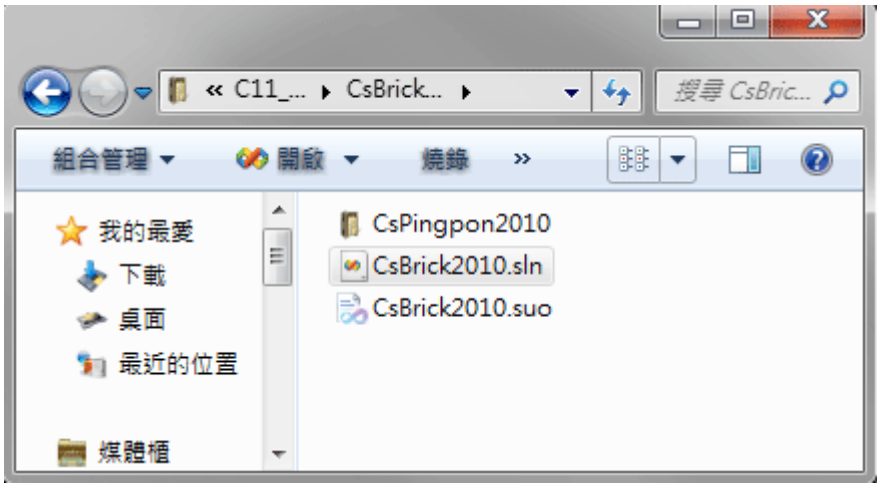
## 12-1 複製乒乓球單元內容

[複製與修改專案名稱]

此專案前半部完全繼承上一單元乒乓球遊戲之功能，你可以將此兩個單元當作一個完整的專案從頭作一次；或者將前一個專案複製之後，依據本單元說明繼續作下去，但是專案或方案名稱會與前一單元重複，如果希望修改為不同名稱請特別注意：應該開啟專案之後在方案總管視窗之內修改，畫面如下：




雖然專案外層的目錄名稱可以在檔案總管中直接重新命名！但是如上例中 CsBrick2010 這樣的名稱切忌在作業系統的檔案總管中修改，原因是在方案愈專案內不是只有一個檔案，很多不同檔案之間是有連結關係的！經過方案總管修改時軟體會設法使得資料連結繼續保持，就是修改其他相關檔案的內容；在檔案總管修改時只會針對一個檔名修改，對於檔案總管來說，專案內的檔案之間有何關係？他是不會處理的！這會造成專案內部資料不同步，很難處理。在方案總管修改名稱後再看看專案目錄變成這樣：




還是有點遺憾，原來專案的下層目錄名稱 CsPingpon2010 還是沒有改到！如果堅持要改完整就使用記事本強制開啟 CsBrick.sln 檔案，將裡面的路徑名稱與 CsPingpon2010 同步改成新的 CsBrick2010 專案即可。

[跨專案的物件複製]

在此或許有人想問：如果兩個專案一起開啟，物件可不可以互相拷貝呢？甚至可不可以將整張作好的表單複製到另一專案貼上呢？基本上如工具箱物件是可以的！譬如 A 專案製作好的一個精美按鍵，可以直接複製到另一同版本專案，但是它的事件程式不會跟過去！至於如表單這種複雜的物件，還包含好幾個實質檔案，就必須使用較複雜的匯入機制了，其實就是使用功能表的→專案→加入現有項目功能，在此暫不深入討論。目前你直接取得的乒乓球程式執行畫面如下，接著就是要在上面佈置很多磚塊，讓球去一一擊破了！


**12-2、磚塊的製作：**

[動態產生控制項的概念]

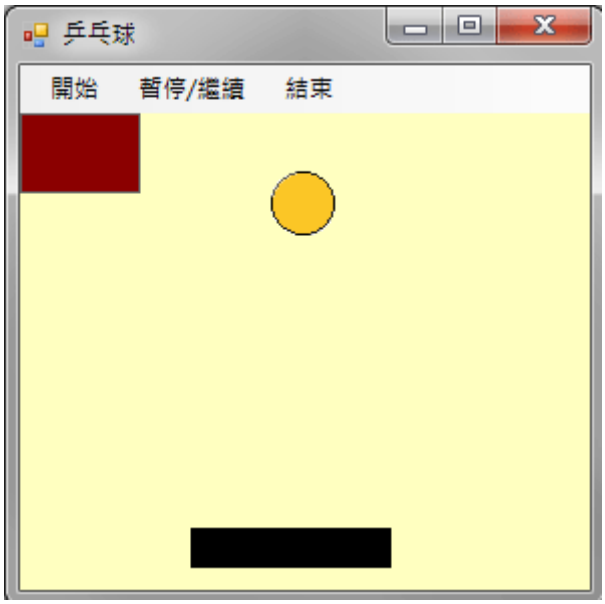
通常這類遊戲的磚塊(目標)數目都很多，如果要在設計階段直接擺好上百個磚塊，還必須排列整齊是非常耗時費事的作法。還好表單上的物件，或稱控制項(Control)如同程式內的變數，也可以在程式進行中用程式碼的宣告方式產生，這種作法我們稱之為「**動態產生控制項**」。只是不同於一般變數宣告，新控制項的產生必須加入「new」關鍵字。其差異在於如果你只是宣告一個物件型別，譬如 Label Q，程式會知道 Q 是一個 Label 類型的物件，實際呼叫使用它時卻會碰到「**並未將物件參考設定為物件的執行個體**」的錯誤訊息！這表示該物件只有專案登錄的名稱，並未真的在記憶體中被建立為實體(instance)，因此它是無法被實際操作的！如同很多父母親替尚未出生的孩子先取好名字，但實際上世間還沒有真的這個人，你也不能為這個「名字」報戶口取得權益一樣！

[製作磚塊物件的副程式]

通常我們會用到動態產生控制項的原因，一是數量很大，一是不確定物件的數量與位置。所以設計程式時必須保持彈性，最好將產生物件的程式碼變成一個會回傳(Return)物件的自訂副程式，我們在打地鼠單元就做過類似的副程式，可以找到並回傳指定名稱的已存在 Label 物件，此地則是必須回傳全新的 Label 物件加入到表單。此副程式實際的操作程序是將我們希望製作的控制項屬性以參數形式傳遞到副程式內，副程式猶如一個工廠或模具，依訂單要求製作出實體物件之後回傳給主程式使用。我們先建立如下的製造磚塊的副程式，並在 Form_Load 事件中試用一下，在表單的左上角座標(0,0)處產生一個磚塊，程式碼與執行畫面如下：

```
private Label Brick(int x, int y)
    Label Q = new Label();
    Q.Width = 60;
    Q.Height = 40;
    Q.BackColor = Color.DarkRed;
    Q.BorderStyle = BorderStyle.FixedSingle;
    Q.Left = x;
    Q.Top = this.menuStrip1.Height + y;
    return Q;
}

private void Form1_Load(object sender, EventArgs e)
{
    this.Controls.Add(Brick(0, 0));
}
```




上述程式碼首先建立一個名為 Brick 的副程式，標題中的 Label 表示回傳值是一個 Label 物件，也就是我們需要的磚塊。當然我們也可以使用 PictureBox 之類的其他物件作為磚塊，但是考慮到記憶體的使用量，應該盡量使用功能少，記憶體資源消耗也少的輕量級物件，如 Label 就是比 PictureBox 好的選擇。

副程式先用關鍵字 new 宣告並建置一個稱為 Q 的 Label 物件，接著連續宣告它的多個屬性，這些屬性中除了位置(x, y)由外部以參數傳入之外，其他都是副程式預設的，所以產品外觀都一樣(寬 60 點、高 40 點、深紅色、有邊線)，只有位置由主程式決定，因為本表單有主功能表(menuStrip1)，所以磚塊的 Y 座標必須加上它的高度(Height)。

[產生的新物件必須加入表單]

最後的 return Q 就是將物件回傳給主程式的意思。在主程式使用它的方式是：this.Controls.Add(Brick(0, 0)); 此語法是將 Brick 產出的 Q 物件加入(Add)表單(this)的控制項集合(Controls)之內，這樣才能在表單上顯示出來，效果如同我們在設計階段將工具箱物件佈置到表單上。在此是將新磚塊設在 x=0, y=0 處就是畫面的左上角。

[用迴圈製作磚牆]

接下來我們要建置遊戲需要的所有磚塊，目標是在視窗的上方做出 10X10 共一百個磚塊的磚牆。這可以在 Form_Load 事件中用兩層的巢狀迴圈控制 x 與 y 座標，再重複呼叫 Brick 副程式即可，程式碼如下：

```
private void Form1_Load(object sender, EventArgs e)
{
    for (int i = 0; i < 10; i++)
    {
        for (int j = 0; j < 10; j++)
        {
            this.Controls.Add(Brick(i  * 60, j * 40));
        }
    }
}
```

上述迴圈的內層改變 y 座標(j)，所以會在固定的 y 位置產生一排垂直排列的磚塊，外層迴圈改變 x 座標，在不同的 x 位置再產生另一排磚塊，掌握好磚塊的寬(60)與高(40)就會變成整齊砌好的磚牆。

[調整視窗符合磚牆的範圍]

但是因為我們事前並未刻意調整視窗大小以符合這面磚牆，所以上述程式產生的磚牆可能會超出視窗範圍，或只佔據視窗的左上角；球與球拍也可能身陷在磚塊之中，使畫面變得不合理。在設計階段要先計算精準這些動態產生物件的關係位置不是不行，但是挺麻煩的！因此可以使用動態調整視窗及既有物件的技巧，以程式計算方式調整畫面。首先我們要使視窗的可編輯區域切齊磚牆的寬度，程式碼如下：

```
this.Width = 600 + this.Width - this.ClientSize.Width;
```

其中的 this 代表視窗，this.Width 代表視窗外框寬度，this.ClientSize.Width 代表的是視窗內部可編輯區的寬度，兩者相減就是視窗左右邊框的寬度總和。我們的目標是讓視窗的可編輯區切齊磚牆的寬度，因為每塊磚寬 60 乘以 10 等於 600，磚牆總寬度為 600 點，this.ClientSize.Width 也應該等於 600 點寬，可惜這個值(this.ClientSize.Width)是唯讀的，無法直接設定！所以我們只能改變視窗外框的寬度(this.Width)來間接調整它，就是使得 this.Width 等於磚牆寬度加上邊框的寬度。同理，高度的處理也類似，但是必須加上可以容納球與球拍的高度空間(在此設定 200 點)。程式碼如下：

```
this.Height = 400 + this.Height - this.ClientSize.Height;
```

[動態調整球與球拍位置]

接下來是球與球拍的位置調整程式：

```
B.Top = 40 * 10 + 30;
P.Top = this.ClientSize.Height  - 30;
B.Left = (this.ClientSize.Width  - B.Width) / 2;
P.Left = (this.ClientSize.Width  - P.Width) / 2;
```

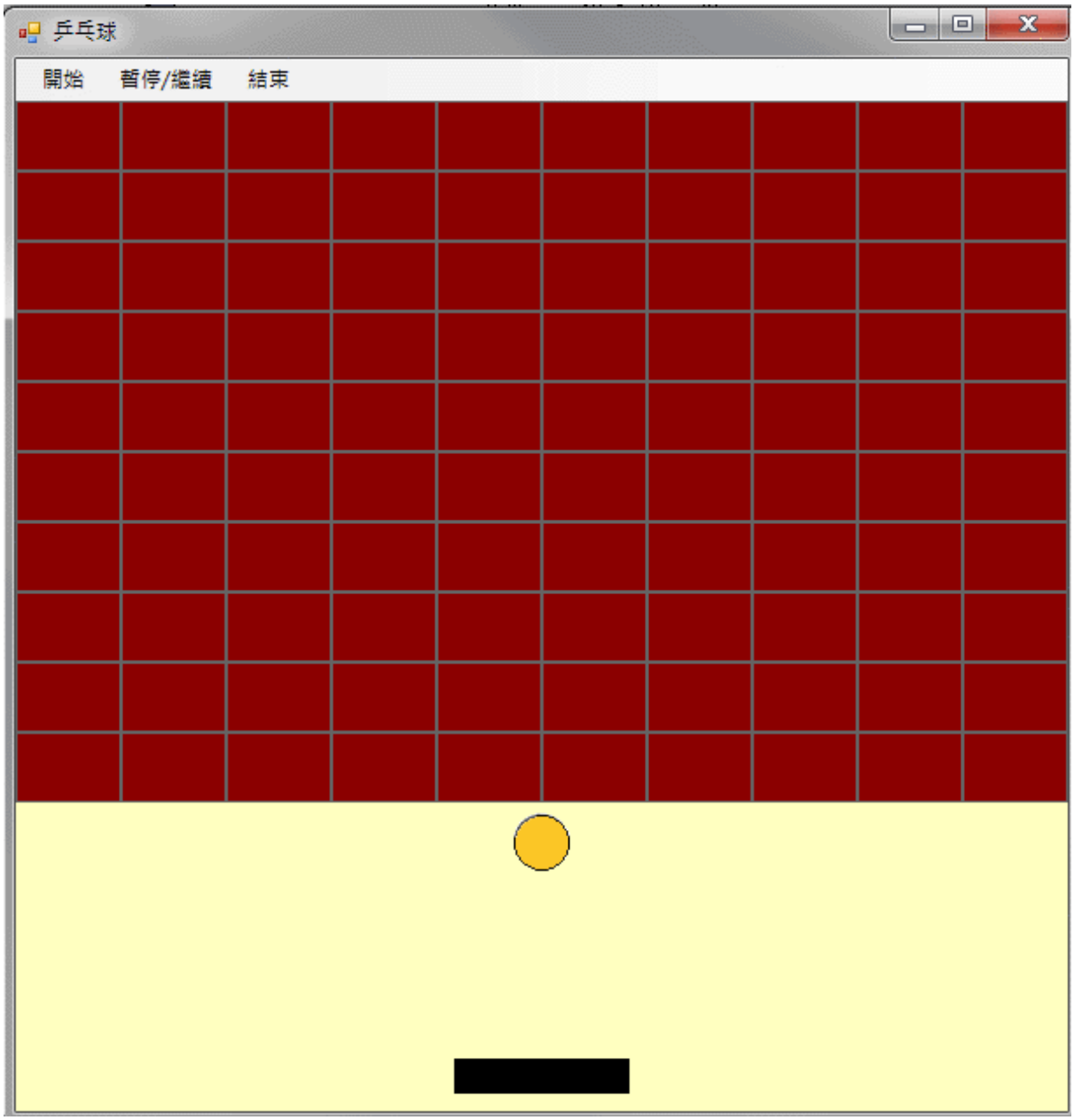
球(B)的高度是磚牆高度(40*10)加上 30，較接近磚牆的位置；拍子(P)的高度是可編輯區高度減 30，較接近視窗的底部。兩者水平位置都給它置中，計算式是可編輯區的寬度減去物件寬度除以 2！完整的 Form_Load 程式碼與執行程式畫面如下：

```
private void Form1_Load(object sender, EventArgs e)
{
    for (int i = 0; i < 10; i++)
    {
        for (int j = 0; j < 10; j++)
        {
            this.Controls.Add(Brick(i  * 60, j * 40));
        }
    }
    this.Width = 600 + this.Width - this.ClientSize.Width;
    this.Height = 400 + this.Height - this.ClientSize.Height  + 200;
    B.Top = 40 * 10 + 30;
    P.Top = this.ClientSize.Height  - 30;
    B.Left = (this.ClientSize.Width  - B.Width) / 2;
    P.Left = (this.ClientSize.Width  - P.Width) / 2;
}
```

因為我們的程式繼承自上一單元，在原程式開始功能表會有重置球的位置的程式，當時是這樣寫的：

```
B.Left = this.ClientSize.Width  / 2;
B.Top = this.ClientSize.Height  / 3;
```

現在也必須以上面有關 B 位置的四行程式取代了！執行程式畫面如下，10×10 的磚牆已經直接建好了！




**12-3、磚塊碰撞效果**

[碰撞偵測的概念]

到此為止程式執行時球會在磚牆上劃過，但是不會對磚塊有任何作用或反應，我們的目標是讓球與磚塊碰撞時會產生合理的反彈，同時磚塊也要消失！以寫程式的角度來說，如何檢查出球與磚塊是否產生碰撞？又是碰到了磚塊的上、下、左、右哪一邊？最為困難，也就是所謂的「碰撞偵測」程式。

在此我們的數學邏輯是隨時記住球的前一個位置，包括它的上下左右邊界，比較球的目前位置與前一位置，如果兩者跨越了磚塊的某一個邊界，就是撞到這個邊了！據此判斷磚塊被碰撞與否，以及球應該如何反彈。具體做法是先在公用變數區域宣告 Btop,

Bbottom, Bleft 與 Bright 四個變數，在 timer1_Tick 事件中球未移動前先記住球(B)的 Top, Bottom, Left 與 Right 四個屬性值。timer1_Tick 事件副程式的前半段程式碼變成如下：

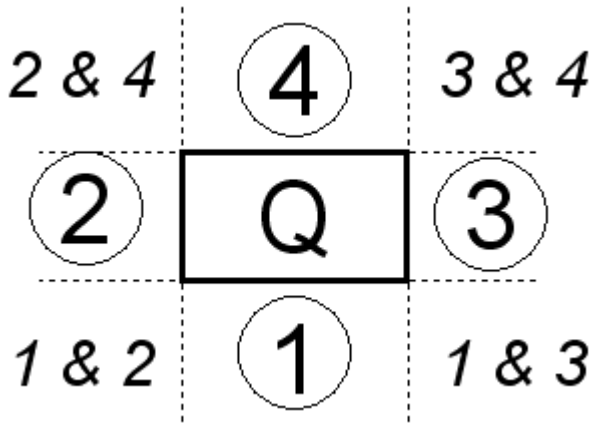
```
int Bleft, Bright, Btop, Bbottom;
private void timer1_Tick(object sender, EventArgs e)
{
    Bleft = B.Left;
    Bright = B.Right;
    Btop = B.Top;
    Bbottom = B.Bottom;
    B.Left += Vx;
    B.Top += Vy;
    ------
```

[碰撞偵測副程式的實作]

這只是記住了球的兩個連續位置的上下左右邊界，檢查碰撞時需要一一以此數據與每一個磚塊的每一個邊界做比較。這個過程相當繁複，所以我們先將此問題簡化成「一個」磚塊與「一個」球的碰撞問題。方法是建立一個碰撞偵測副程式，在此命名為 chkHit。此副程式可以用參數傳入(等於指定)一個磚塊(Label Q)，讓它與本遊戲中唯一的一顆球(B)作碰撞與否的檢查。如果有碰撞就立即讓磚塊消失，球也必須依其與磚塊相撞的角度產生合理的反彈。譬如：碰到磚塊左邊就該往左彈，碰到右邊就該往右彈等等。以程式的角度來說就是 Vx 或 Vy 之一要改變其正負號，確定有碰撞時副程式回傳一個 true 值，若無碰撞發生則回傳 false。chkHit 副程式的設計如下：

```
private Boolean chkHit(Label Q)
{
    if (B.Top > Q.Bottom) { return false; } //球在磚塊的下方
    if (B.Right < Q.Left) { return false; } //球在磚塊左方
    if (B.Left > Q.Right) { return false; } //球在磚塊右方
    if (B.Bottom < Q.Top) { return false; } //球在磚塊的上方
    Q.Visible = false;
    if (B.Top <= Q.Bottom && Btop > Q.Bottom) //撞到磚塊底部
    {
        Vy = Math.Abs(Vy);
        return true;
    }
    if (B.Right >= Q.Left && Bright < Q.Left) //撞到磚塊左邊
    {
        Vx = -Math.Abs(Vx);
        return true;
    }
    if (B.Left <= Q.Right && Bright > Q.Right) //撞到磚塊右邊
    {
        Vx = Math.Abs(Vx);
        return true;
    }
    if (B.Bottom >= Q.Top && Bbottom < Q.Top) //撞到磚塊頂部
    {
        Vy = -Math.Abs(Vy);
        return true;
    }
    return false;
}
```

副程式的第一行 if (B.Top > Q.Bottom) { return false; }檢查球目前是否在磚塊的下邊界之外，條件是球(B)的頂部(Top)大於磚塊(Q)的底部(Bottom)，這表示兩物件確定沒有碰撞的狀況之一，球位於如下示意圖的標示①區，副程式至此結束，傳回一個 false。依此類推，接下來的三行程式分別檢查球是否在示意圖的②③④區，請注意此四區都是半平面，且互有重疊(如標示 1&2，2&3…等區塊)，最終涵蓋了磚塊之外的所有區域，所以如果四個檢查都通過之後，球『必然』與磚塊有接觸，也就是有部分重疊，那就是『碰撞』了！此時一定要發生的事是磚塊必須消失(Q.Visible = false)。




接著看看球應該如何反彈？讓我們解析第一種狀況，就是球碰到磚塊的底部：

```
if (B.Top <= Q.Bottom && Btop > Q.Bottom) //撞到磚塊底部
{
    Vy = Math.Abs(Vy);
    return true;
}
```

判斷式是如果目前的球頂部(B.Top)與磚塊下邊緣有重疊(B.Top >= Q.Bottom)，但是之前一個位置時的球並沒有(Bright < Q.Left)，這表示球剛剛跨越過這個邊界，就是撞到此邊界了，正常的反彈行為是 Vy 變成正值(往下，Vy = Math.Abs(Vy))，同時不必再看其他狀況了，因為合理的反彈應該只有一個方向，所以結束副程式回傳一個 true(撞到磚塊底部了！)。同樣的，接下來的三個 if 判斷式決定球是否撞到左(Q.Left)、右(Q.Right)或上(Q.Top)邊緣，只要撞到其中一個，就立即結束副程式，不必再繼續往下檢查了！

[碰撞偵測的優先次序]

您是否注意到我們偏愛先從下方開始檢查碰撞？原因是磚牆位於上方，球最可能撞到的就是下緣，如果確認撞到下緣，其他邊緣就不必檢查了！這除了可以減少程式執行時間之外，因為我們是數位遊戲，球事實上是跳躍式的移動，確實有可能同時符合兩個面都有碰撞的條件，這有一點違反物理定律，此時最好優先讓最可能的碰撞條件先被檢出，執行反應程式，而忽略其他機率較低的碰撞情況，以免低可能性的碰撞反而排擠掉較合理的碰撞行為。

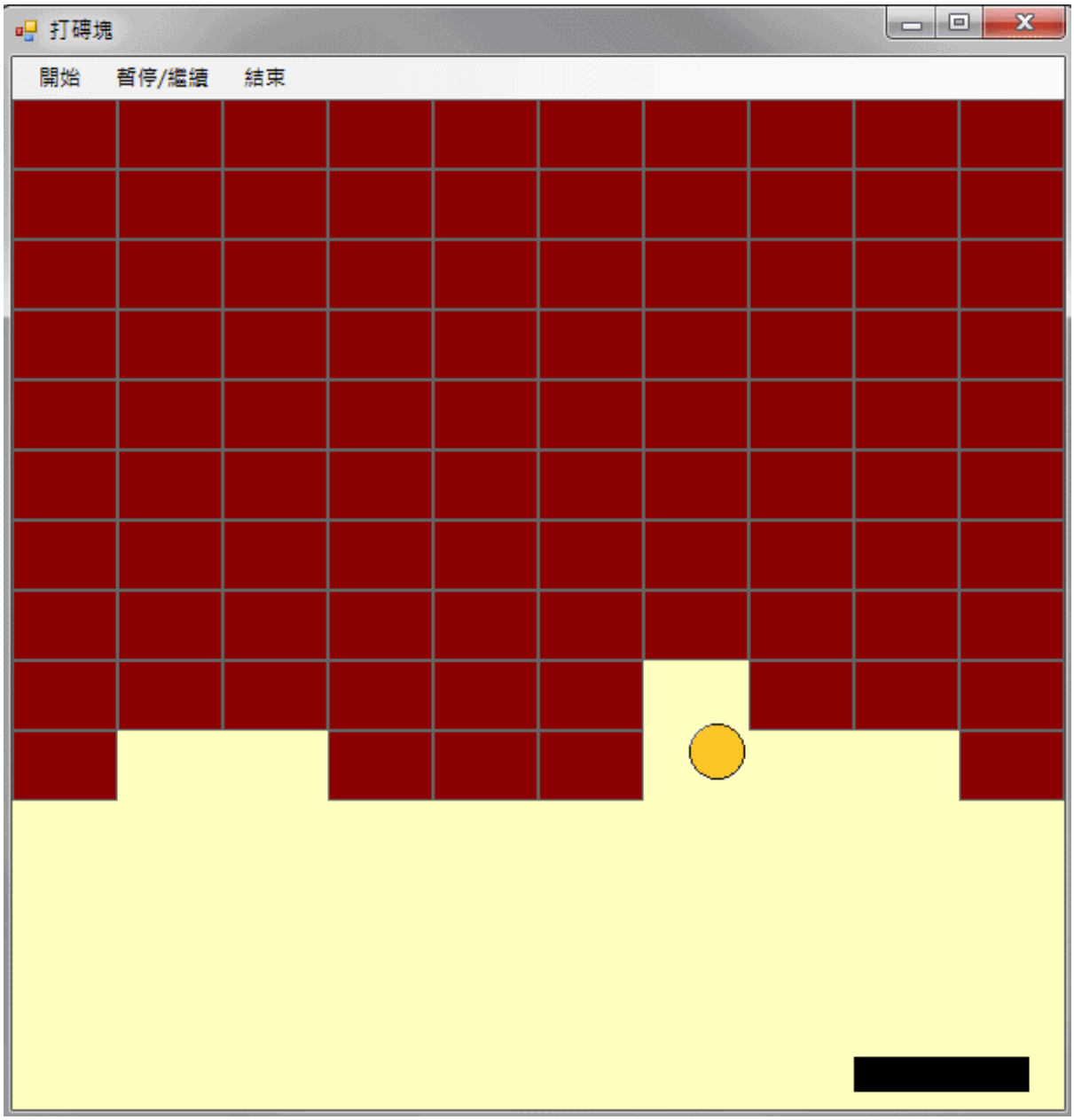
[碰撞偵測的執行時機]

至此，其實我們都還無法測試 chkHit 是否正常運作，必須在 timer1_Tick 事件之末加上執行所有磚塊碰撞檢查的程式碼：

```
foreach (var c in this.Controls)
{
    if (c is Label)
    {
        Label Q = (Label)c;
        if (Q.Visible)
        {
            if (chkHit(Q)) { break; }
        }
    }
}
```

這是一個特殊的迴圈，foreach (var c in this.Controls)的意義是依序呼叫每一個表單上的控制項(資料型態未定稱 var)，並暫定其名稱為 c 的意思。接下來用判斷式 if (c is Label)先判斷此控制項的型態是否為標籤(Label)？同時還必須是還看得見(Visible)的標籤(if (Q.Visible))。在此專案中所有標籤都是磚塊，看得見表示還未被擊破，必須進一步檢查碰撞與否，否則就不必作碰撞檢查了！接下來的 chkHit(Q)就是將 Q 物件丟去檢查，完成後會回傳 true 或 false 如果是 true 就跳出整個副程式(break)，不再檢查其他磚塊，其意義是球的每一步移動只能擊破一個磚塊，這樣在視覺上比較合理。

最後必須強調，因為程式畢竟還是跳躍式的移動，與類比的世界中連續運動的物理現象終究會有差異，不合理的情況還是可能會出現，要減少這種問題合理的方向是將 Timer 的 Interval 屬性減少(執行速度變快)，一次移動的距離(Vx 與 Vy)也減少，代價則是電腦資源使用較多。最後看看可能的遊戲過程畫面吧！




**12-4、進階挑戰**

一、如何加入計分機制？

提示：宣告公用變數紀錄積分，於磚塊擊破時加分。

二、如何建立不同外觀的磚塊，且區隔計分？

提示：修改 Brick 副程式的內容，並將不同的預期計分寫入 Tag 屬性。

三、如何加入音效？

提示：參考打地鼠單元加入資源檔，並在碰撞程式碼之後加入音效播放程式。

**課後閱讀**

**程式執行階段的物件建置與刪除**

[動態產生物件的程式處理]

本單元最有趣的新技術應該是使用程式碼製造大量磚塊的程式，比較專業的術語叫作「動態產生物件」。因為我們已經習慣在設計階段自工具箱拉出物件，而在程式碼中宣告數值或文字變數，這讓我們有個刻板印象，以為這是截然不同的兩件事情，變數與物件也是完全不同的東西。但事實上所有的物件也都可以如一般變數一樣直接用程式碼宣告產生，只是這些「物件」通常比文數字資料結構大而複雜，要正確的讓它們成為表單上可以正常運作的一份子，步驟會多出幾步。

第一個差別是文數字等等變數我們稱為實質型別(Value Type)，使用如 int i; 這樣的語法宣告之後這個變數就是實際可以使用的「東西」，你可以定義它等於 3 還可以將它與其他數字相加等等。也就是電腦已經配置了記憶體位置給這個變數 i，你讓 i=3 的時候記憶體的這個位置的值就變成 3 了！

[物件名稱與實體的宣告]

但是對於非實質型別的東西，譬如只用 Label A 這樣的宣告，只是告訴電腦它的資料類型，就是專案在記事本上記載「名稱 A 的東西是個 Label」，但事實上電腦並不會因此在記憶體中給它一個實際的位置，當你以為 A 是個實體物件繼續操作時不是出現錯誤訊息，就是物件根本看不到！因為它實際上並不存在(只有一個空名牌)！關鍵是要使用 new 來建置物件「實體」，因此動態產生物件的第一關是必須用 Label A = new Label(); 這種語法宣告新物件，因為建置新物件也是一個『動作』所以程式末端 Label 後面的小括號是必須的！非實質型別的東西稱為參考型別(Reference Type)，意思是物件名稱與實體儲存的位置不同，有可能只有空名牌沒有實體，但是實質型別有名字時就一定有實體了！

[新物件必須宣告加入表單]

要產生新物件的操作接下來是用程式碼去定義原本我們習慣用屬性視窗定義的一些屬性，包括物件的大小、顏色與位置等等！最後還有一個容易出錯的地方是，即使你做好了屬性完整的物件實體，它仍然深藏在記憶體中，我們必須將它「加入表單」才會變成可見的物件。程式碼像這樣：this.Controls.Add(A);！

[動態刪除物件]

或許想像力豐富的同學會問：可以用程式碼新增物件，那麼可不可以刪除物件呢？答案當然是可以的！只要用『物件 • Dispose();』這個方法就可以了！Dispose 原意是處理或丟棄的意思，甚至你在設計階段手動加入的物件都可以在程式執行中殺掉！你可以試試在表單上隨便加入一個 pictureBox1，再到 Form_Load 事件中加入一行程式：

pictureBox1.Dispose();結果程式一開始就會看不到這個物件，因為它已經被程式碼殺掉了！

[刪除或隱藏物件的差異]

也因此，如果本單元的程式不要將磚塊隱藏(Q.Visible=false)而是直接刪除這個磚塊(Q.Dispose())其實也是可以的！重玩時只要再執行一次建立磚牆的程式即可。對於玩家來說視覺效果完全一樣！唯一要避免的情況是打掉磚塊時隱藏它，之後又用 new 建立新的磚塊，於是乎記憶體裡面隱形的磚塊越來越多！雖然看不見，但是一樣會消耗記憶體，最後程式就越跑越慢甚至當掉了！

[設計階段的建置物件程式碼]

在此我們可以學到一個很重要的概念，就是**不要將設計階段的物件設計認為是不可更動的狀態**。各位可以去看 Form1.Designer.cs 裡面的程式碼，其實與我們動態產生物件的程式碼幾乎完全一樣！下面就是本單元球物件的建置程式碼：

```
//
// B
//
this.B.Image = ((System.Drawing.Image)(resources.GetObject("B.Image")));
this.B.Location = new System.Drawing.Point(207, 148);
this.B.Name = "B";
this.B.Size = new System.Drawing.Size(32, 32);
this.B.SizeMode = System.Windows.Forms.PictureBoxSizeMode.AutoSize;
this.B.TabIndex = 0;
this.B.TabStop = false;
```

是不是覺得很面熟呢？原來我們新增磚塊物件的副程式就是模仿它寫的！

因此，程式的物件狀態始終就是一個可以動態改變的環境，所謂的設計階段只是給予物件們一個初始的狀態而已！程式執行之後沒有東西不能再度改變，學到此單元你應該會發現程式設計的空間與可能性忽然又變大了！設計階段的圖形化物件設計環境雖然減少了很多程式碼的寫作，但是對於初學者很容易變成一個限制，以為物件一經設計就沒辦法增減了！這個單元就是要打破這個迷思，圖形化設計與程式碼設計一直都是相通的！