Information Hiding

We have seen that one of the real advantages of an IDE like Microsoft Visual Studio is that it hides much of the complexity involved in Windows programming. To use even a very complicated control like a Button, you only need to know the characteristics of the Button that you can manipulate (its **fields**) and the commands the Button understands (its **methods** and the inputs it can respond to (its **events**). So, if you wish to change the width of a Button, you simply say *button1.width = 100*. There is no need for you to know, or care, how the system actually changes the width of the Button, redraws it and recalculates its target area for mouse clicks. Similarly, if you want the Button to do something when clicked, you invoke its *Click* handler. You don't need to understand the quite complex processes involved in monitoring for mouse clicks and communicating with the operating system. All this is hidden.

Technically, this is called ***Information Hiding***, and it is the primary philosophy of Object Oriented Programming – the programmer works on a "need to know" basis.

A system that is based on Information Hiding divides any control into two parts: its **interface**, and its **implementation**. The interface is that part of the control that people interact with – methods they can call, properties they can modify, and so forth. The implementation is how all these things actually work underneath. The user has access only to the interface.

We see Information Hiding in many areas. For example, when I drive my car, I am benefiting from the principle of Information Hiding. I only know that the accelerator makes the car go faster and the brake makes the car go slower. I have absolutely no idea how the engine works, or how it makes the wheels go around. For my car, the accelerator and brake are the interface, and the engine and transmission and so forth are the implementation.

One of the great benefits of Information Hiding is that as long as the interface remains the same, the implementation can change, and it doesn't impact the user. For example, if I get a new car and it has a rotary engine (implementation), it makes no difference to me, as long as the pedals (the interface) work the same. Similarly, if someone comes up with a much more efficient implementation of a button, it won't affect my programming, as long the methods, events and properties stay the same.

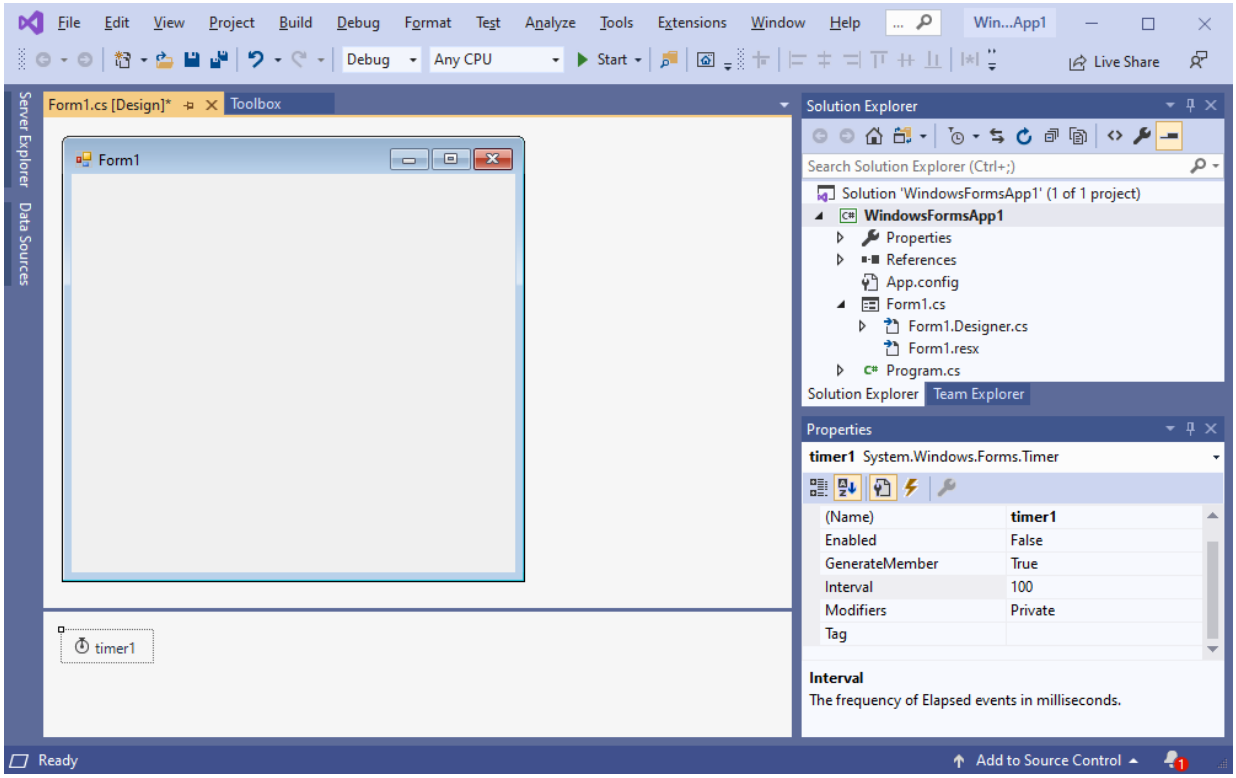
Today we will look at an extremely complex component, the Timer component, which sits in the background of our program and executes an event handler at some specified interval. The implementation of this behaviour is horrendously complex but we can just ignore the complexity that lies beneath. The Timer is easy to use because the interface is very simple; we need only know its methods, events and properties. The thousands of lines of code that are actually required to implement its behaviour need not concern us.

The *Timer* Component

The *Timer* is like a little clock running in the background of your program. You write an event handler for it, and each time the *Timer* 'ticks', the code in the event handler is executed.

The *Timer* component is on the *Components* tab of the ToolBox. The *Timer* component is similar to a control except it has no visual representation on the form, sometimes referred to as a non-visual control. That is, unlike a *Button* or a *TextBox*, a *Timer* does not appear on the screen at run-time. When you add a *Timer* to your application (design time), the *Timer* icon does not appear on the *Form* itself but in the section below the *Form*. The *Timer* exists at run-time, and you can write event handlers for it, it just doesn't appear on the user's screen.

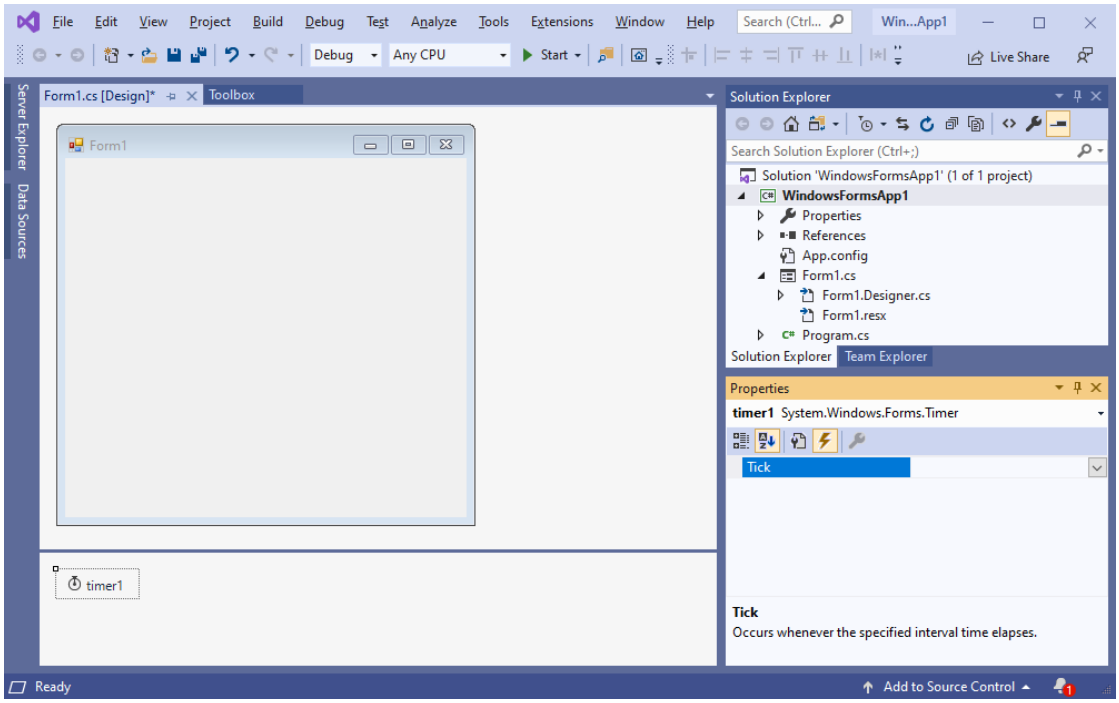
When you place a *Timer* component onto a form, you use the Properties window to see the *Timer's* properties and events.



Although the *Timer* component has an extremely complex **implementation** you can see that it has a very simple **interface.** It has only four properties:

(Name)	This is the name used to refer to the Timer component in your code. The default Name is *timer1*.
Enabled	A Boolean property, where the default is False. It determines whether the Timer is turned on or off, ie whether the Timer is ticking. When *timer1.Enabled = true*, the Timer ticks. When *timer1.Enabled = false*, the Timer does not tick. *timer1.Enabled* can also be modified at run-time to turn the Timer on and off.
GenerateMember	We will ignore this property for the moment.
Interval	Determines how frequently the Timer ticks when it is Enabled. The value of Interval is given in milliseconds. Thus, if you want the Timer to tick every second, set Interval to 1000. The default value is 100.
Modifiers	We will ignore this property for the moment.
Tag	A storage element for the convenience of the programmer. This may be useful in the assignment.

The *Timer* component has only one Event – *Tick* – which occurs when the *Timer* ticks. You write a *Tick* handler in the same way that you write *Click* handlers for buttons, by selecting *Tick* in the events tab of the *Properties* window, and filling in the code skeleton.



This is a good time to talk about the default handlers that you access by **double-clicking** a component on the Form at design time. For buttons, this is the *Click* handler. For forms, it is *Load*. For Timers it is *Tick.*

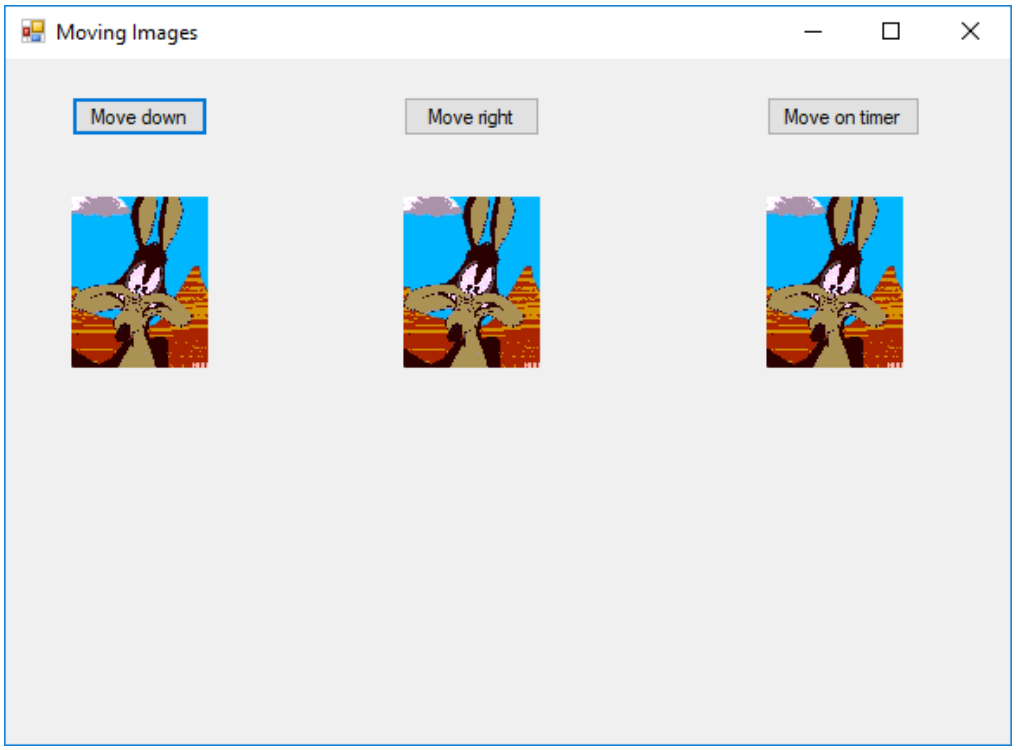
Whatever code you write in the *Tick* handler will be executed each time the *Timer* fires (i.e. every *Interval* milliseconds) when *timer1.Enabled* is true. *Timer* components are essentially like infinite for-loops. They simply keep executing their handler over and over until someone turns them off.

Task 1: The Timer Component

1. Start a new project.
2. Place a *TextBox* on the Form. Set its *Text* property to 0 (zero).
3. From the Components tab of the ToolBox, place a *Timer* component onto the *Form*.
4. Use the Events tab of the Properties to create a *Tick* handler for your Timer. This is the event that will be executed at each *Timer* tick.
5. Write your *Tick* event handler so that, at each tick, the value in the *TextBox* increases by 1. Remember that you need to convert from strings (what the *TextBox* holds) to Integers (which you can add 1 to) and back as required.
6. You can modify the *Interval* property at design-time via the Properties. Test the program with several different values for the *Interval* property. How would you modify the *Interval* at run-time?
7. Add a *Button* to your *Form*. Write the required code so that, when you click on this *Button*, the *Timer* stops.
8. Tricky problem: Modify the *Click* handler of the *Button* you created in step 7, so that it toggles the *Timer*. That is, if the *Timer* is not enabled, it enables it; if the *Timer* is enabled, it turns it off. This can be done with a single line of code. (Hint: implement the Boolean **not** operator.)

Task 2: Moving Images

1. Start a new project.
2. Place three *PictureBox* controls on the *Form*.
3. Place a *Button* directly above each *PictureBox*.



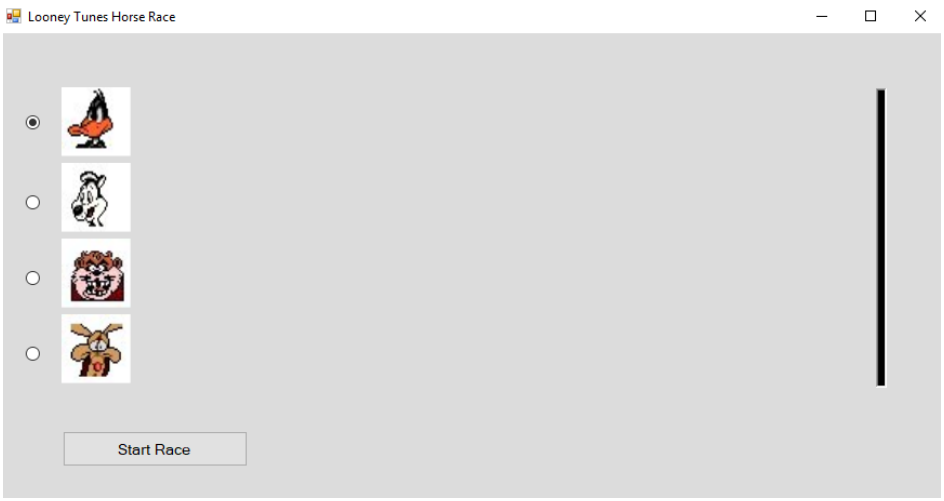
4. Write a *button1_Click* handler so that, when the *button1* is clicked, the first image moves 10 pixels down the page:

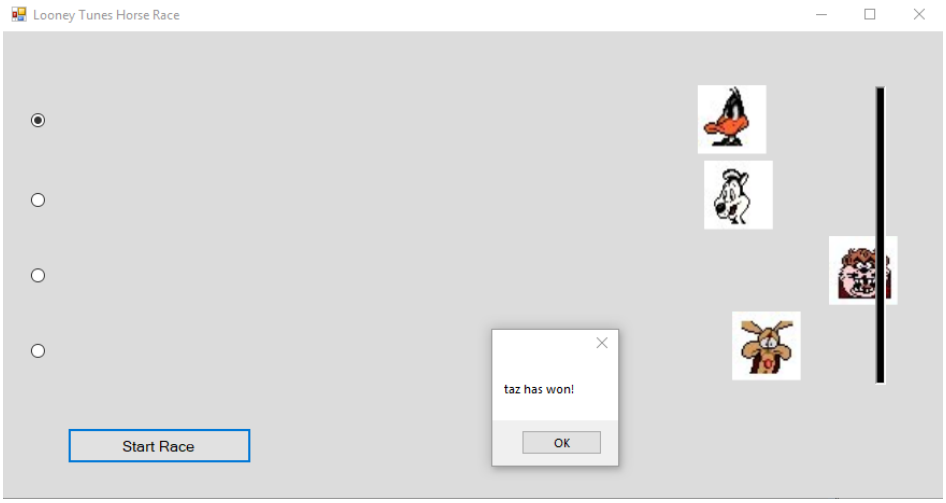
```
pictureBox1.Top = pictureBox1.Top + 10;
```

5. When you have checked that this code works, rewrite the code using constants and the += operator.
6. Write a *button2_Click* handler so that, when the *button2* is clicked, the second image moves 10 pixels across the page.
7. Write a *button3_Click* handler so that, when the *button3* is clicked, a timer is enabled and the third image moves down and across the page, from top right corner to bottom left of the screen.
8. What other properties could you change?

Task 3: Looney Tune Horse Race ******* CHECKPOINT 5 *******

1. In this task you are going to build a horse race game. The four "horses" are images loaded into *PictureBox* controls (use image files on Teams or source your own images). I have used a *Panel* control for the finish line. The winner is the first image to touch the finish line. The outcome of the race is displayed in a *Label* control. Screenshots of my implementation before and after a race are shown below:





2. **Set up the form** with four PictureBoxes, a *Start Race* Button, a Panel for the finish line and a Timer. The Radio Buttons are optional (see #5), and used for user to bet on a winner.

3. **Start by designing the classes.** You should implement two classes: Horse and Controller.

   The **Horse class** needs to know its name, its PictureBox which has been loaded with its image at design time, a reference to the Random object that is created in the Form, and an integer field for the left hand side of the finish line Panel.
   It needs a constant for the maximum speed of 10 pixels.
   It needs a constructor to initialise all its field values.
   It needs to know how to move itself (this should be calculated as a random value between 0 and its speed), how to reset its PictureBox back to the start position. It also needs to know and whether the right hand side of its PictureBox has passed the finish line.
   Will you need to create any properties for the fields?

The **Controller class** is the game engine. It controls the race, commanding each horse in turn to move itself and querying each horse to find out if it has reached the finish line.
It should hold an array of Horses, and a Boolean field that holds whether the race is over.
Its constructor should be used to create the four Horse instances and initialise the other fields with appropriate values.
It has a method to restart the race which tells each horse to reset itself to the starting position.
The Race method should tell each horse to move itself. It should ask if each horse has reached the finish line. If so, the race should be stopped and the winner announced.

A MessageBox.Show() is a useful method to provide feedback to the user:

```
MessageBox.Show(horses[i].Name + " has won!");
```

4. The **Form1 class** is the container for the controls and should be used only as the user interface.
   Create an array to hold the four PictureBoxes, a Controller object and a Random object.
   Initialise the Timer's enabled to false in the Form's constructor.
   The *StartRace* button will instruct the controller to restart the race and enable the timer.
   On each Timer Tick, the controller object should check whether the race is over (in which case it should disable the timer) or if the race is still on (then call its Race() method).

5. **Refining your code.** Go through your code checking that all literal values are replaced with constants. Check for repetitious code, is there a need for an array? What happens if two horses win the race?

6. Add a radio button beside each image's starting position. The user bets on one of the "horses" by clicking on the corresponding radio button. Inform the user (using *MessageBox.Show* or a Label) whether his choice won. To get really fancy, start the user with some money and allow him to state how much he bets. Keep track of his winnings. You can even offer the player odds on his bet.
