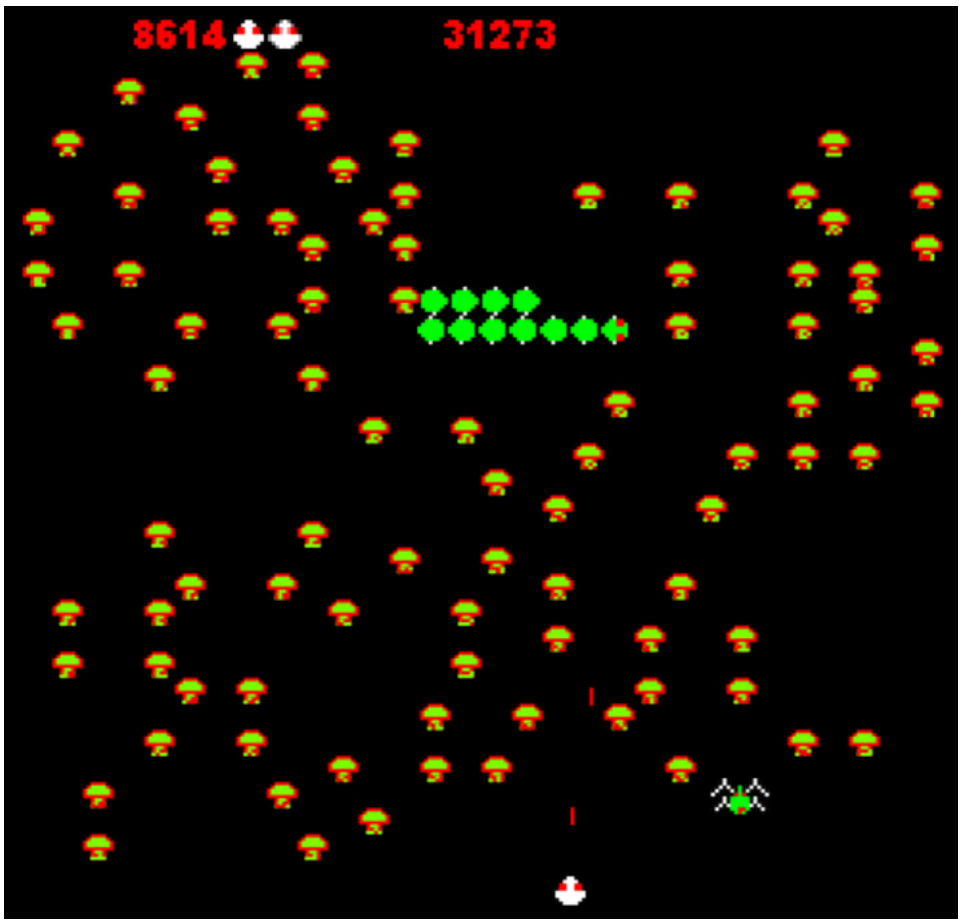
Práctica - Videojuego en modo consola

Como práctica-resumen de estos temas de Programación Orientada a Objetos vamos a desarrollar un proyecto de un videojuego en modo consola, siguiendo algunas de las pautas que se han visto en los apuntes para el proyecto Console Invaders que os hemos propuesto intentar hacer.

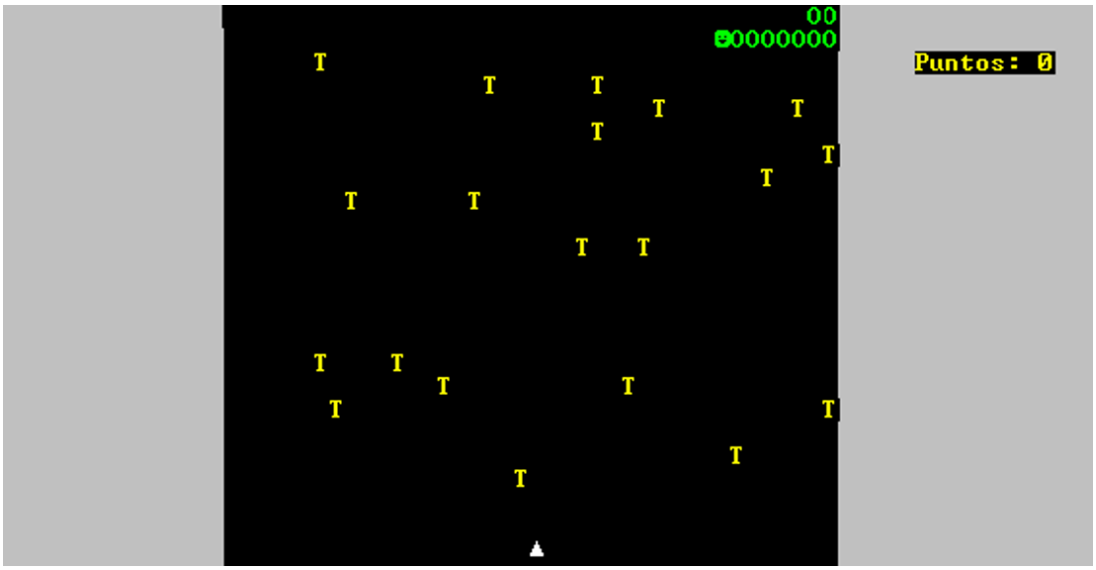
En este caso, vamos a desarrollar una versión alternativa del videojuego **Centipede**, un clásico de los años 80 en las recreativas y ordenadores de 8 bits (Amstrad, Spectrum, etc.).



El juego consiste en pilotar una nave que se mueve por la parte inferior de la pantalla, y disparar y destruir un ciempiés que viene avanzando desde la parte superior. El ciempiés está compuesto de varios bloques, y se mueve de un lado a otro, bajando al llegar al final de cada línea. Al disparar a un bloque del ciempiés, éste se destruye, y al destruir todos los bloques, el ciempiés queda destruido y finalizamos el nivel.

A lo largo del escenario hay una serie de obstáculos, que hacen que el ciempiés rebote y baje de línea antes de llegar a la pared. En el juego original también aparecen otra serie de enemigos, como arañas, pero esa parte la obviaremos en nuestro juego.

Obviamente, al utilizar sólo la consola del sistema, nuestra apariencia será mucho más modesta...



En las siguientes secciones veremos indicaciones de qué pasos seguir para desarrollar el videojuego en etapas, aplicando los conceptos vistos hasta ahora.

1. Proyecto y primeros pasos (1 punto)

Para empezar, vamos a crear un nuevo proyecto en nuestro IDE (Visual Studio, Xamarin, o el que estemos usando), llamado **CentipedeConsole**. Nos creará la clase principal `Program.cs`, y aparte iremos añadiendo otras clases adicionales en los siguientes pasos. De momento, renombraremos esta clase principal a `CentipedeConsole.cs`.

1.1. La configuración del juego

Para gestionar la configuración del juego, definiremos una serie de constantes en una clase `Configuracion`. Por ejemplo podría quedar así (aunque puedes añadir cualquier otra constante que consideres necesaria):

```
class Configuracion
{
    public const int ANCHO_PANTALLA = 80;
    public const int ALTO_PANTALLA = 25;
    public const int PAUSA_BUCLE = 70;
    public const int VIDAS_INICIALES = 3;

    public static Random random = new Random();
}
```

1.2. Dinámica de pantallas

Definiremos las 3 pantallas del juego, en tres clases respectivas:

- Una de bienvenida llamada `Bienvenida` con un pequeño rótulo e instrucciones de juego
- La pantalla del juego en sí en la clase `Partida`
- Una pantalla de créditos al finalizar, en la clase `Creditos`

El programa principal `CentipedeConsole` se va a encargar de coordinar el paso entre pantallas, tal y como configuraremos aquí:

- Desde *Bienvenida* entraremos en *Partida* pulsando *Enter*, o saldremos del juego pulsando *Escape*
- Desde *Partida* saldremos a *Creditos* al pulsar *Escape*, o cuando se nos agoten las vidas (esto último ya lo añadiremos más adelante)
- Desde *Creditos* volveremos a *Bienvenida* pulsando cualquier tecla.

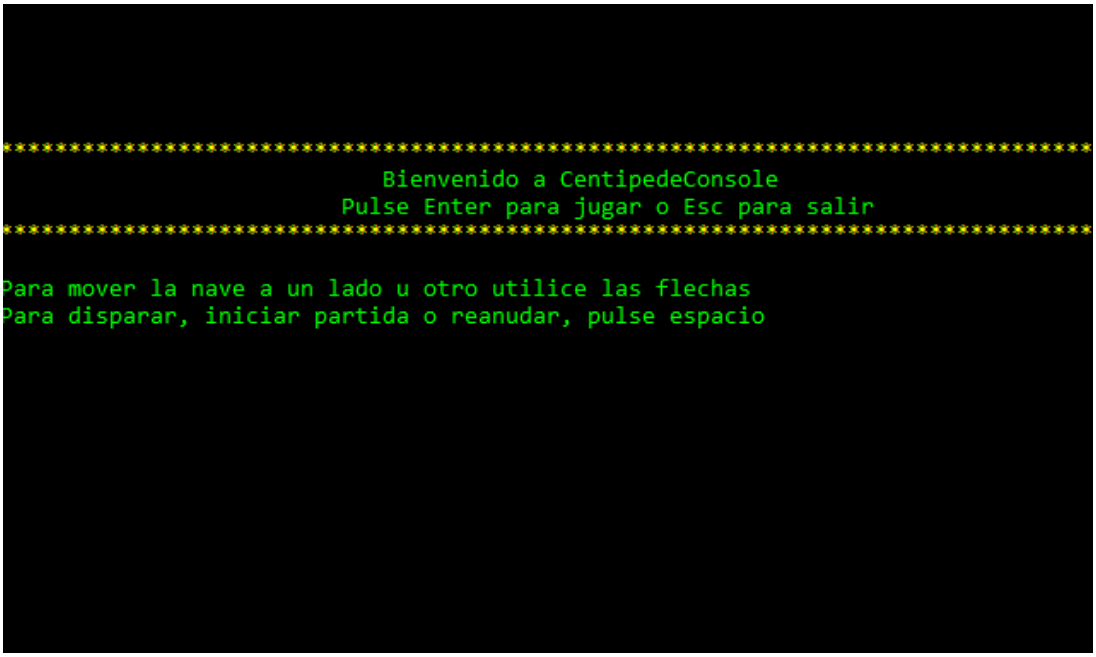
1.2.1. La pantalla de bienvenida

El contenido de la clase `Bienvenida` será algo parecido a esto. Pasaremos a la pantalla de juego al pulsar la tecla *Enter*, y también saldremos del juego desde esta pantalla pulsando la tecla *Escape*.

> **TAREA**: Completa el código del método `Lanzar` mostrando el rótulo del juego y las instrucciones para entrar y jugar, a tu gusto.

```
class Bienvenida
{
    bool salir;

    public void Lanzar()
    {
        ConsoleKeyInfo tecla;
        Console.Clear();

        // Completa aquí el código para mostrar el título
        // e instrucciones del juego

        Console.ResetColor();

        do
        {
            tecla = Console.ReadKey(true);
            if (tecla.Key == ConsoleKey.Escape)
                salir = true;
            else if (tecla.Key == ConsoleKey.Enter)
                salir = false;
        }
        while (tecla.Key != ConsoleKey.Escape &&
               tecla.Key != ConsoleKey.Enter);
    }

    // Obtiene si queremos salir del juego
    public bool GetSalir()
    {
        return salir;
    }
}
```

Una posible apariencia de esta pantalla de bienvenida puede ser la siguiente:



También podéis hacer un rótulo algo más artístico para el título del juego (aunque no es obligatorio), algo así:

1.2.2. La pantalla de partida

En la clase `Partida` definiremos el bucle principal del juego. De momento simplemente recogeremos que, cuando se pulse *Escape*, se salga de la pantalla (para ir a la de *Créditos*).

```
class Partida
{
    private ConsoleKeyInfo tecla = new ConsoleKeyInfo();

    // Lanza la partida principal
    public void Lanzar()
    {
        Console.Clear();
        // Este mensaje lo quitaremos cuando definamos el contenido
        Console.WriteLine("Pantalla de juego");
        do
        {
            if (Console.KeyAvailable)
            {
                while (Console.KeyAvailable)
                {
                    tecla = Console.ReadKey(true);
                }
            }
        }
        while (tecla.Key != ConsoleKey.Escape);
    }


}
```

1.2.3. La pantalla de créditos

En la clase `Creditos` mostraremos el mensaje de GAME OVER, junto con un texto para que el usuario pulse cualquier tecla para volver a la pantalla inicial.

> **TAREA**: Completa el código con el mensaje y el rótulo, con el estilo que quieras.

```
class Creditos
{
    // Lanza la pantalla de bienvenida, y se guarda si
    // queremos salir o jugar en la variable "salir"
    public void Lanzar()
    {
        Console.Clear();

        // Muestra aquí los mensajes con el estilo que quieras

        Console.ResetColor();
        Console.ReadKey(true);
    }
}
```

1.2.4. La coordinación de pantallas

En la clase `CentipedeConsole` vamos a coordinar el paso entre pantallas, mientras dure el juego.

```
class CentipedeConsole
{
    static void Main()
    {
        Bienvenida b = new Bienvenida();
        Creditos c = new Creditos();

        // Configuración inicial de la consola
        Console.CursorVisible = false;
        Console.WindowWidth = Configuracion.ANCHO_PANTALLA;
        Console.WindowHeight = Configuracion.ALTO_PANTALLA;

        do
        {
            b.Lanzar();
            if (!b.GetSalir())
            {
                Partida p = new Partida();
                p.Lanzar();
                c.Lanzar();
            }
        } while (!b.GetSalir());

        Console.Clear();
    }
}
```

2. Definiendo la jerarquía de clases (1 punto)

2.1. La clase *Sprite*

Para gestionar los diferentes elementos que intervienen en el juego (el arqueólogo y la(s) momia(s)) definiremos una clase padre abstracta `Sprite` con las características generales de todos ellos.

```
abstract class Sprite
{
    protected int x;
    protected int y;
    protected string imagen;

    protected Sprite(int x, int y, string imagen)
    {
        this.x = x;
        this.y = y;
        this.imagen = imagen;
    }

    public int GetX()
    {
        return x;
    }

    public int GetY()
    {
        return y;
    }

    public string GetImagen()
    {
        return imagen;
    }

    public void SetX(int cx)
    {
        x = cx;
    }

    public void SetY(int cy)
    {
        y = cy;
    }

    public void SetImagen(string img)
    {
        imagen = img;
    }

    public virtual void Dibujar()
    {
        Console.SetCursorPosition(x, y);
        Console.Write(imagen);
        Console.ResetColor();
```

```
    }
}
```

2.1. La clase *Nave*

Uno de los subtipos de Sprite será la nave que controlaremos, el personaje principal. Así, crearemos una clase `Nave` que heredará de `Sprite`, incorporando así los atributos generales (coordenadas X e Y, e imagen a dibujar).

```
class Nave : Sprite
{
    public Nave(int cx, int cy): base(cx, cy, "" + (char)30)
    {
    }

    public override void Dibujar()
    {
        Console.ForegroundColor = ConsoleColor.White;
        base.Dibujar();
    }
}
```

> **NOTA**: observa que la imagen de la nave es el carácter con código ASCII 30. Si consultas una tabla ASCII, verás que es una flecha hacia arriba.

2.2. La clase *Obstaculo*

Los obstáculos que se encontrará el ciempiés en su camino los definiremos con la clase `Obstaculo`, de forma similar a la nave anterior:

```
class Obstaculo : Sprite
{
    public Obstaculo(int cx, int cy): base(cx, cy, "T")
    {
    }

    public override void Dibujar()
    {
        Console.ForegroundColor = ConsoleColor.Yellow;
        base.Dibujar();
    }
}
```

2.3. El ciempiés

El tratamiento del ciempiés es más complejo que el resto de sprites del juego, ya que no es un sólo elemento, sino una serie de eslabones o bloques unidos, de forma que conforme vamos disparando al ciempiés, va perdiendo bloques. Para facilitar el tratamiento del ciempiés, vamos a utilizar varias clases para definirlo.

2.3.1. Los bloques del ciempiés

Definiremos una clase `BloqueCiempies`, que representará cada eslabón o bloque del ciempiés. Su estructura será similar a las clases anteriores (*Nave* u *Obstaculo*): un constructor para inicializar sus valores, y un método Dibujar para dibujar el bloque con el símbolo que hayamos elegido (en nuestro ejemplo, usaremos una O mayúscula verde):

```
class BloqueCiempies : Sprite
{
    public BloqueCiempies(int cx, int cy) : base (cx, cy, "O")
    {
    }

    public override void Dibujar()
    {
        Console.ForegroundColor = ConsoleColor.Green;
        base.Dibujar();
    }
}
```

2.3.2. La cabeza del ciempiés

Aunque no es necesario, para poder identificar mejor la cabeza del ciempiés y ver hacia dónde se dirige en cada momento, vamos a definir un subtipo de la clase anterior *BloqueCiempies*, que nos sirva para definir otra imagen para la cabeza. Esta clase en principio sólo va a tener un constructor que llame al padre, y cambie la imagen por defecto (la O mayúscula verde) por otra (en nuestro ejemplo, usaremos la carita sonriente del símbolo ASCII 2, con color verde también):

```
class CabezaCiempies : BloqueCiempies
{
    public CabezaCiempies(int cx, int cy): base(cx, cy)
    {
        imagen = "" + (char)2;
    }
}
```

2.3.3. Ensamblando el ciempiés: clase *Ciempies*

Definimos ahora la clase `Ciempies`. Esta clase no va a heredar de *Sprite*, ya que no es un sprite propiamente dicho, sino que en su interior va a contener un array de bloques de ciempiés (clase *BloqueCiempies* vista antes). En el constructor indicaremos como parámetro de qué tamaño va a ser el ciempiés (cuántos bloques va a tener, nunca superior al ancho de la pantalla) y crearemos un array de esos bloques, asignando a cada uno una posición predeterminada para empezar desde la esquina superior izquierda, quedando la cabeza en el extremo derecho del ciempiés.

```
class Ciempies
{
    private BloqueCiempies[] bloquesCiempies;

    public Ciempies(int longitud)
    {
        longitud = longitud % Configuracion.ANCHO_PANTALLA;
        bloquesCiempies = new BloqueCiempies[longitud];
        bloquesCiempies[0] = new CabezaCiempies(longitud-1, 0);
        for (int i = 1; i < longitud; i++)
            bloquesCiempies[i] = new BloqueCiempies(longitud-1-i, 0);
    }

    public void Dibujar()
    {
        for (int i = 0; i < bloquesCiempies.Length; i++)
            bloquesCiempies[i].Dibujar();
    }
}
```

3. La pantalla del juego (1 punto)

En cuanto a la pantalla del juego (clase `Partida`), en ella irá el bucle del juego, un bucle que repite los 5 pasos que se dan en cada iteración del juego, que son:

1. Dibujar los elementos del juego (personaje, enemigos, disparos, etc)
2. Mover al personaje
3. Mover a los enemigos y cualquier otro elemento móvil
4. Detectar colisiones (disparo que alcanza al enemigo, choque del enemigo con el personaje, etc.) y actualizar el estado del juego (restar vidas, sumar puntos, cambiar posición de los elementos afectados, etc.)
5. Pausar un poco el juego, si va demasiado rápido

3.1. El marco de la pantalla

Comenzaremos por dibujar el marco de la pantalla, desde una función `DibujarMarco` que añadiremos a la propia clase `Partida`. Usaremos una constante para indicar el grosor de los bordes, que podemos añadir a nuestra clase `Configuracion`:

```
public const int ANCHO_BORDE_LATERAL = 20;
```

En cuanto al método `DibujarMarco`, podría quedar así:

```
private void DibujarMarco()
{
    Console.BackgroundColor = ConsoleColor.Gray;

    for (int i = 0; i < Configuracion.ALTO_PANTALLA; i++)
    {
        Console.SetCursorPosition(0, i);
        Console.Write(new string(' ', Configuracion.ANCHO_BORDE_LATERAL));
        Console.SetCursorPosition(Configuracion.ANCHO_PANTALLA -
            Configuracion.ANCHO_BORDE_LATERAL, i);
        Console.Write(new string(' ', Configuracion.ANCHO_BORDE_LATERAL));
    }

    Console.ResetColor();
}
```

Invocaremos a este método al inicio del método `Lanzar`, y quitaremos el mensaje provisional que tenía antes esta pantalla:

```
public void Lanzar()
{
    Console.Clear();
    DibujarMarco();
    do
    {
        // El resto de código no cambia de momento
```

3.2. Definiendo un array de obstáculos

Vamos a dejar definidos ya los elementos que no van a moverse en el juego: los obstáculos. Define un método llamado `CrearObstaculos` en la clase `Partida` que reciba como parámetro un tamaño (entero), y devuelva un array de objetos *Obstaculo* de ese tamaño, cada uno en unas coordenadas. Estas coordenadas deben cumplir una serie de requisitos:

- No puede haber dos obstáculos en el array con las mismas coordenadas, ni en coordenadas adyacentes (para evitar que el ciempiés pueda quedar atrapado)
- La coordenada Y de los obstáculos debe estar entre 1 y 20 (inclusive), para evitar que pueda haber obstáculos en la primera fila (donde empieza el ciempiés) y en las 4 últimas (donde se moverá nuestra

nave)

- La coordenada X de los obstáculos debe estar entre 21 y 58 (inclusive), para que no se salgan de los bordes laterales del juego. Dejaremos libre la primera y última columnas (X = 20 y X = 59) para evitar que el ciempiés no encuentre forma de bajar. Aplicamos el mismo criterio que para el primer requisito: que no haya dos obstáculos (en este caso obstáculo y pared) adyacentes.

```
static Obstaculo[] CrearObstaculos(int cantidad)
{
    Obstaculo[] obstaculos = new Obstaculo[cantidad];
    for(int i = 0; i < obstaculos.Length; i++)
    {
        int x = Configuracion.random.Next(
            Configuracion.ANCHO_BORDE_LATERAL + 2,
            Configuracion.ANCHO_PANTALLA - Configuracion.ANCHO_BORDE_LATERAL - 1);
        int y = Configuracion.random.Next(1,
            Configuracion.ALTO_PANTALLA - 4);
        obstaculos[i] = new Obstaculo(x, y);
    }
    return obstaculos;
}
```

> **TAREA**: mejora la función anterior para que cumpla con todos los requisitos indicados: que no haya dos obstáculos en la misma posición, ni en posiciones adyacentes

3.3. Inicializar elementos del juego

Vamos ahora a inicializar una nave, un ciempiés y los obstáculos en el juego. En primer lugar añadimos atributos nuevos a la clase `Partida`:

```
class Partida
{
    private ConsoleKeyInfo tecla = new ConsoleKeyInfo();
    private Obstaculo[] obstaculos = CrearObstaculos(20);
    private Nave nave = new Nave(Configuracion.ANCHO_PANTALLA / 2,
        Configuracion.ALTO_PANTALLA - 1);
    private Ciempies ciempies = new Ciempies(10);

    ...
```

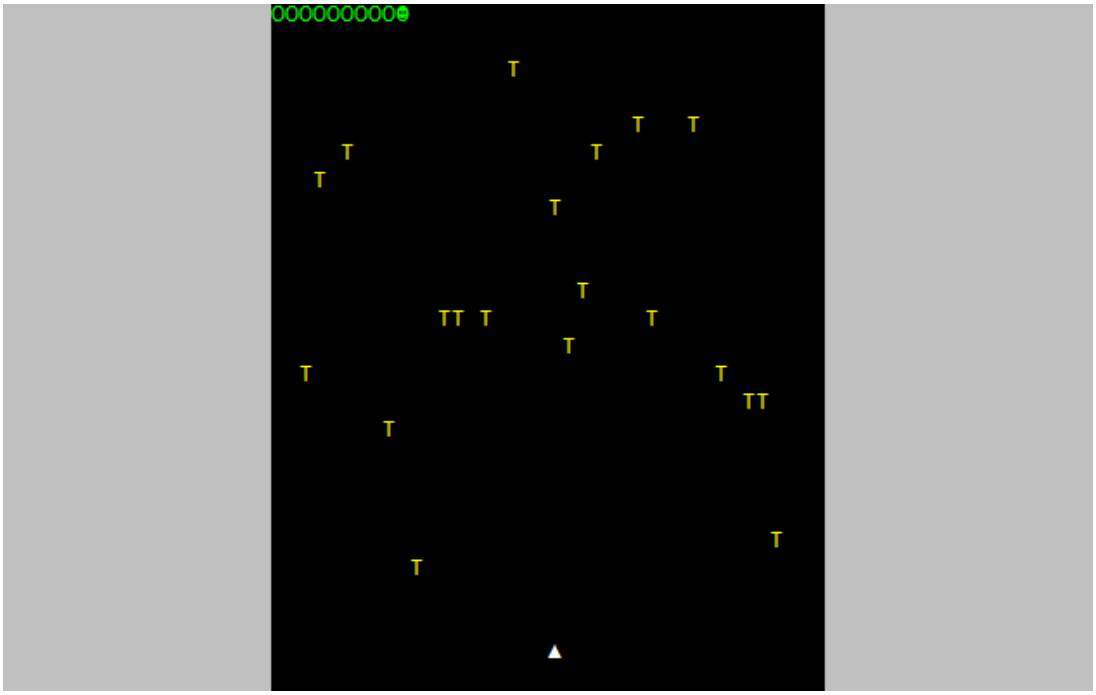
Creamos un método `DibujarElementos` que se encargue de dibujarlos:

```
private void DibujarElementos()
{
    for (int i = 0; i < obstaculos.Length; i++)
    {
        obstaculos[i].Dibujar();
    }
    nave.Dibujar();
    ciempies.Dibujar();

    // Añadimos esta línea para que la animación no sea muy rápida
    System.Threading.Thread.Sleep(Configuracion.PAUSA_BUCLE);
}
```

Llamamos a este método dentro del bucle `do..while` en el método `Lanzar`:

```
public void Lanzar()
{
    Console.Clear();
    DibujarMarco();

    do
    {
        DibujarElementos();

        if (Console.KeyAvailable)
        {
            while (Console.KeyAvailable)
            {
                tecla = Console.ReadKey(true);
            }
        }
    }
    while (tecla.Key != ConsoleKey.Escape);
}
```

Si pruebas a ejecutar el juego ahora, en la pantalla de `Partida` deberías ver algo como esto (la disposición de los obstáculos es aleatoria y puede variar):



> **TAREA**: deberás modificar el constructor de la clase `Ciempies` para que empiece a dibujarse a partir del marco lateral, en lugar de la esquina izquierda

4. Moviendo los elementos del juego (2 puntos)

En este apartado vamos a dar movimiento tanto a la nave como al ciempiés.

4.1. Movimiento de la nave

Para mover la nave usaremos las teclas de los cursores (izquierda y derecha). Definimos en la clase `Partida` un método `EventosTeclado` que recoja el evento en cuestión y mueva la nave:

```
private void EventosTeclado()
{
    if (Console.KeyAvailable)
    {
        while (Console.KeyAvailable)
        {
            tecla = Console.ReadKey(true);
        }
        if (tecla.Key == ConsoleKey.LeftArrow)
        {
            nave.Mover(nave.GetX() - 1);
        }
        else if (tecla.Key == ConsoleKey.RightArrow)
        {
            nave.Mover(nave.GetX() + 1);
        }
    }
}
```

Llamamos a este método desde el bucle `do..while`, en `Lanzar`:

```
public void Lanzar()
{
    ...

    do
    {
        DibujarElementos();
        EventosTeclado();
    }
    while (tecla.Key != ConsoleKey.Escape);
}
```

En la clase `Nave` añadimos un método `Mover` que nos permitirá cambiar la posición horizontal de la nave:

```
public void Mover(int nuevaX)
{
    // Borramos posición antigua
    Console.SetCursorPosition(x, y);
    for (int i = 0; i < imagen.Length; i++)
        Console.Write(" ");

    // Cambiamos a la nueva posición
    SetX(nuevaX);
}
```

> **TAREA**: modifica el método `Mover` para que la nave no se salga por los bordes laterales del juego.

4.2. Movimiento del ciempiés

El movimiento del ciempiés es algo complejo de describir. Globalmente, se mueve hacia la derecha o hacia la izquierda, alternativamente. Al encontrar una pared u obstáculo, baja a la siguiente fila y cambia su dirección horizontal. En el juego original, si llega al final de la pantalla, vuelve hacia arriba de nuevo, aunque nosotros haremos, por simplificar, que cuando llegue al final de la pantalla finalice la partida.

Pero... ¿cómo podemos plasmar eso en un elemento compuesto de eslabones o bloques? Básicamente, lo que vamos a hacer es que cada eslabón o bloque se mueva siguiendo ese patrón. Inicialmente, el ciempiés siempre va a partir de una posición fija (arriba a la izquierda del escenario), con la primera fila completamente vacía de obstáculos. Por lo tanto, comenzará moviéndose hacia la derecha.

4.2.1. Tipos de movimiento del ciempiés

Como el ciempiés va a admitir tres movimientos (izquierda, derecha y abajo), vamos a definir un *enum* en la clase `BloqueCiempies` que sirva para representar cada movimiento. También definiremos un atributo del tipo del *enum* que indicará qué movimiento tiene actualmente el ciempiés, y lo inicializaremos a la derecha (como hemos dicho, inicialmente el ciempiés se moverá a la derecha).

```
enum movimientoCiempies { IZQUIERDA, DERECHA, ABAJO }

class BloqueCiempies : Sprite
{
    private movimientoCiempies movimientoActual = movimientoCiempies.DERECHA;

    ...

    public movimientoCiempies GetMovimientoActual()
    {
        return movimientoActual;
    }
}
```

4.2.2. Ubicación de los obstáculos alrededor del ciempiés

Según el movimiento del ciempiés, puede que se encuentre con obstáculos a su izquierda, a su derecha o debajo de él, o que no haya obstáculo. Definiremos un nuevo *enum* y una variable que guarde el estado actual de obstáculos en torno al ciempiés:

```
enum movimientoCiempies { IZQUIERDA, DERECHA, ABAJO }
enum obstaculosCiempies { IZQUIERDA, DERECHA, ABAJO, NO_OBSTACULO }

class BloqueCiempies : Sprite
{
    private movimientoCiempies movimientoActual = movimientoCiempies.DERECHA;
    private obstaculosCiempies obstaculoActual = obstaculosCiempies.NO_OBSTACULO;

    ...
```

Mientras el ciempiés se mueva en horizontal, no tendrá obstáculos a los lados. En el momento en que se encuentre uno y vaya hacia abajo, necesitaremos almacenar a qué lado estaba el obstáculo, para movernos en la dirección opuesta tras bajar.

4.2.3. El método para mover cada bloque del ciempiés

Ahora vamos a definir un método llamado `Mover` dentro de esta clase `BloqueCiempies`. Lo que haremos en este método es:

1. Si no hay obstáculo junto al bloque, seguir con la dirección actual (izquierda o derecha)
2. Si hay obstáculo junto al ciempiés, marcar a qué lado está (dependiendo de en qué dirección se esté moviendo el ciempiés ahora), y poner dirección hacia abajo
3. Si la dirección actual es hacia abajo, comprobar a qué lado estaba el obstáculo que nos hizo ir hacia abajo, y cambiar de dirección. En realidad, este tercer paso tendremos que ponerlo como primero en

nuestro algoritmo.

Inicialmente sólo vamos a considerar como obstáculos las paredes laterales (más adelante tendremos en cuenta los obstáculos intermedios), así que nuestro método Mover va a ser algo así:

```
public void Mover()
{
    if (movimientoActual == movimientoCiempies.ABAJO)
    {
        if (obstaculoActual == obstaculosCiempies.DERECHA)
            movimientoActual = movimientoCiempies.IZQUIERDA;
        else
            movimientoActual = movimientoCiempies.DERECHA;
        obstaculoActual = obstaculosCiempies.NO_OBSTACULO;
    }
    if (GetX() == Configuracion.ANCHO_BORDE_LATERAL &&
        movimientoActual == movimientoCiempies.IZQUIERDA)
    {
        obstaculoActual = obstaculosCiempies.IZQUIERDA;
        movimientoActual = movimientoCiempies.ABAJO;
    }
    else if (GetX() == Configuracion.ANCHO_PANTALLA -
       Configuracion.ANCHO_BORDE_LATERAL - 1 &&
       movimientoActual == movimientoCiempies.DERECHA)
    {
        obstaculoActual = obstaculosCiempies.DERECHA;
        movimientoActual = movimientoCiempies.ABAJO;
    }
}
```

4.2.4. Movimiento completo del ciempiés

Ahora vamos a nuestra clase `Ciempies`. Definimos un método `Mover` que mueva cada bloque, según su movimiento actual:

```
public void Mover()
{
    for (int i = 0; i < bloquesCiempies.Length; i++)
    {
        Console.SetCursorPosition(bloquesCiempies[i].GetX(),
                                  bloquesCiempies[i].GetY());
        Console.Write(" ");

        bloquesCiempies[i].Mover();

        if (bloquesCiempies[i].GetMovimientoActual() ==
            movimientoCiempies.IZQUIERDA)
            bloquesCiempies[i].SetX(bloquesCiempies[i].GetX() - 1);
        else if (bloquesCiempies[i].GetMovimientoActual() ==
            movimientoCiempies.DERECHA)
            bloquesCiempies[i].SetX(bloquesCiempies[i].GetX() + 1);
        else if (bloquesCiempies[i].GetMovimientoActual() ==
            movimientoCiempies.ABAJO)
            bloquesCiempies[i].SetY(bloquesCiempies[i].GetY() + 1);
    }
}
```

4.2.5. Mover el ciempiés desde el bucle del juego

Ahora vamos a la clase `Partida`. Añadimos un método `MoverElementos` que nos permitirá mover los elementos que tengan un movimiento automático del juego, como ahora el ciempiés y, más tarde los disparos.

```
private void MoverElementos()
{
    ciempies.Mover();
}
```

Llamamos a este método en el bucle `do..while` del método `Lanzar`:

```
public void Lanzar()
{
    ...

    do
    {
        DibujarElementos();
        EventosTeclado();
        MoverElementos();
    }
    while (tecla.Key != ConsoleKey.Escape);
}
```

4.3. Disparos

Vamos ahora a añadir disparos al juego, para que la nave pueda disparar al ciempiés. En primer lugar añadimos la clase `Disparo` al programa:

```
class Disparo : Sprite
{
    public Disparo(int cx, int cy) : base(cx, cy, "|")
    {
    }

    public override void Dibujar()
    {
        Console.ForegroundColor = ConsoleColor.Red;
        base.Dibujar();
    }
}
```

En la clase `Partida` creamos un atributo de tipo `Disparo`, inicialmente a *null*:

```
class Partida
{
    private ConsoleKeyInfo tecla = new ConsoleKeyInfo();
    private Obstaculo[] obstaculos = CrearObstaculos(20);
    private Nave nave = new Nave(Configuracion.ANCHO_PANTALLA / 2,
        Configuracion.ALTO_PANTALLA - 1);
    private Ciempies ciempies = new Ciempies(10);
    private Disparo disparo = null;

    ...
```

En el método `EventosTeclado` añadimos que, cuando se pulse la tecla de la barra espaciadora, se cree un disparo en la posición donde está la nave.

```
private void EventosTeclado()
{
    if (Console.KeyAvailable)
    {
        while (Console.KeyAvailable)
        {
            tecla = Console.ReadKey(true);
        }
        if (tecla.Key == ConsoleKey.LeftArrow)
        {
            nave.Mover(nave.GetX() - 1);
        }
        else if (tecla.Key == ConsoleKey.RightArrow)
        {
            nave.Mover(nave.GetX() + 1);
        }
        else if (tecla.Key == ConsoleKey.Spacebar)
        {
            if (disparo == null)
                disparo = new Disparo(nave.GetX(), nave.GetY());
        }
    }
}
```

Observa que sólo creamos el disparo si éste es *null*, con lo que sólo habrá un disparo a la vez en la pantalla. Es una de las dificultades del juego: saber elegir bien el momento en que disparar. Ahora vamos a dibujar el disparo, junto al dibujado del resto de elementos (nave, ciempiés...), en el método `DibujarElementos`:

```
private void DibujarElementos()
{
    for (int i = 0; i < obstaculos.Length; i++)
    {
        obstaculos[i].Dibujar();
    }
    nave.Dibujar();
    ciempies.Dibujar();
    if (disparo != null)
        disparo.Dibujar();

    // Añadimos esta línea para que la animación no sea muy rápida
    System.Threading.Thread.Sleep(Configuracion.PAUSA_BUCLE);
}
```

Finalmente, en el método `MoverElementos` movemos el disparo:

```
private void MoverElementos()
{
    // Mover ciempiés
    ciempies.Mover();
    // Mover disparo
    if (disparo != null)
    {
        Console.SetCursorPosition(disparo.GetX(), disparo.GetY());
        Console.Write(" ");
        if (disparo.GetY() == 0)
            disparo = null;
        else
            disparo.SetY(disparo.GetY() - 1);
    }
}
```

Observa que el disparo se mueve si no es *null* y, en cuanto llega a la parte superior de la ventana (Y == 0), se pone a *null* para poder disparar de nuevo.

5. Detección de colisiones (1,5 puntos)

En esta sección vamos a identificar distintos tipos de colisiones que tenemos pendientes:

- Del disparo con un obstáculo
- Del disparo con el ciempiés
- Del ciempiés con un obstáculo
- Del ciempiés con la nave

5.1. Detección de colisión disparo-obstáculo

En el videojuego real, cuando la nave dispara a un obstáculo, éste se va destruyendo poco a poco con cada disparo. Como es difícil hacer eso en una aplicación de consola, vamos a optar por destruir el obstáculo directamente con el primer disparo que le llegue.

Hay que tener en cuenta que, cuando un obstáculo está destruido, no debe dibujarse en pantalla, y debe "dejar pasar" al ciempiés y a los disparos por el espacio que ocupaba.

5.1.1. Cambios en la clase *Obstaculo*

Para controlar esto, vamos a añadir un atributo booleano a la clase `Obstaculo`, llamado `destruido`, que nos indique si el obstáculo ha sido destruido o no. Lo inicializaremos a *false* en el constructor (al crear cada obstáculo al principio, ninguno estará destruido), y añadiremos también sus correspondientes *getter* y *setter*.

```
class Obstaculo : Sprite
{
    private bool destruido;

    public Obstaculo(int cx, int cy) : base(cx, cy, "T")
    {
        destruido = false;
    }

    public bool GetDestruido()
    {
        return destruido;
    }

    public void SetDestruido(bool destruido)
    {
        this.destruido = destruido;
    }
    ...
```

En el método `Dibujar` de la clase, sólo dibujaremos si el obstáculo no está destruido:

```
public override void Dibujar()
{
    if (!destruido)
    {
        Console.ForegroundColor = ConsoleColor.Yellow;
        base.Dibujar();
    }
}
```

5.1.2. Cambios en la clase *Partida*

En la clase `Partida` vamos a añadir un nuevo método que compruebe la colisión del disparo con los obstáculos:

```
private void ColisionDisparoObstaculos()
{
    int i = 0;
    if (disparo != null)
    {
        while (i < obstaculos.Length && disparo != null)
        {
            if (!obstaculos[i].GetDestruido() &&
                obstaculos[i].GetX() == disparo.GetX() &&
                obstaculos[i].GetY() == disparo.GetY())
            {
                Console.SetCursorPosition(obstaculos[i].GetX(),
                    obstaculos[i].GetY());
                Console.Write(" ");
                obstaculos[i].SetDestruido(true);
                disparo = null;
            }
            i++;
        }
    }
}
```

Observad que, si las coordenadas del disparo coinciden con las de un obstáculo no destruido, lo destruimos, lo borramos de la pantalla y ponemos el disparo a *null*.

Ahora vamos a añadir un método que centralice todas las comprobaciones de colisiones que haremos. Este método usará el método anterior:

```
private void ComprobarColisiones()
{
    ColisionDisparoObstaculos();
}
```

Llamamos ahora a este método desde el final del bucle `do..while`, en el método `Lanzar`:

```
public void Lanzar()
{
    Console.Clear();
    DibujarMarco();

    do
    {
        DibujarElementos();
        EventosTeclado();
        MoverElementos();
        ComprobarColisiones();
    }
    while (tecla.Key != ConsoleKey.Escape);
}
```

A partir de este punto, cualquier otra colisión que queramos comprobar la añadiremos con un nuevo método en `ComprobarColisiones`, y se incorporará automáticamente al bucle del juego.

5.2. Detección de colisión disparo-ciempiés

La colisión disparo-ciempiés la vamos a tratar de una forma más sencilla que en el videojuego original. En él, al alcanzar el disparo un eslabón del ciempiés, ese eslabón concreto se destruía, y de las dos mitades que quedaban (salvo que fuera el último eslabón) se generaban dos ciempiés. Por simplificarlo bastante, vamos a hacer que cuando el disparo alcance cualquier eslabón del ciempiés, eliminaremos un eslabón de la cadena y haremos el ciempiés un eslabón más corto.

5.2.1. Cambios en la clase *Ciempies*

Para controlar el tamaño del ciempiés con los eslabones perdidos por disparos, vamos a añadir un nuevo atributo a la clase `Ciempies` que indique cuántos eslabones o bloques se han perdido (por disparos de la nave, obviamente). Lo inicializamos a 0, y definimos dos nuevos métodos:

- Un método llamado `CiempiesDestruido` que devuelva un booleano indicando si todos los eslabones del ciempiés se han destruido (es decir, si el número de eslabones perdidos coincide con la longitud del array de eslabones).
- Otro método llamado `DestruyeBloque` que aumente el número de bloques perdidos en una unidad (lo llamaremos cada vez que un disparo alcance al ciempiés).

```
class Ciempies
{
    private BloqueCiempies[] bloquesCiempies;
    private int numBloquesPerdidos;

    public Ciempies(int longitud)
    {
        numBloquesPerdidos = 0;
        ... // Resto del constructor no cambia
    }

    public bool CiempiesDestruido()
    {
        return numBloquesPerdidos == bloquesCiempies.Length;
    }

    public void DestruyeBloque()
    {
        numBloquesPerdidos++;
    }

    ...
```

En el método `Dibujar` de la clase `Ciempies` sólo dibujaremos tantos bloques como queden sin destruir (empezando por la cabeza):

```
public void Dibujar()
{
    for (int i = 0; i < bloquesCiempies.Length - numBloquesPerdidos; i++)
        bloquesCiempies[i].Dibujar();
}
```

Finalmente, vamos a añadir un método más a esta clase que nos ayude a detectar las colisiones de un disparo con cualquiera de los bloques. El método recibirá un objeto de tipo `Disparo` como parámetro por referencia. Internamente recorrerá el array de bloques que queden por destruir, y si las coordenadas de alguno coinciden con las del disparo, incrementará en uno el número de bloques destruidos y pondrá el disparo a null para destruirlo también (por eso debemos pasar el disparo por referencia).

```
public void DetectarColisionConDisparo(ref Disparo disparo)
{
    int i = 0;
    while(i < bloquesCiempies.Length - numBloquesPerdidos && disparo != null)
    {
        if (bloquesCiempies[i].GetX() == disparo.GetX() &&
            bloquesCiempies[i].GetY() == disparo.GetY())
        {
            DestruyeBloque();
            disparo = null;
        }
        i++;
    }
}
```

5.2.2. Cambios en la clase *Partida*

Ahora llamamos a este método desde `ComprobarColisiones` en la clase `Partida`:

```
private void ComprobarColisiones()
{
    ColisionDisparoObstaculos();
    ciempies.DetectarColisionConDisparo(ref disparo);
}
```

5.3. Detección de colisión ciempiés-obstáculo

Este apartado es especialmente complicado de plantear. Hasta ahora la lógica del movimiento del ciempiés la tenemos planteada en el método *Mover* de la clase *BloqueCiempies* (donde establecemos en qué dirección se moverá el ciempiés basándonos en el paso previo y los obstáculos detectados), y el método *Mover* de la clase *Ciempies* (donde cambiamos las coordenadas de cada bloque en función del movimiento que se haya establecido para cada uno).

5.3.1. Cambios en la clase *BloqueCiempies*

Vamos a modificar el método `Mover` de la clase `BloqueCiempies`. Tenemos que pasarle como parámetro el array de obstáculos del juego para que compruebe si el bloque choca con alguno de ellos. Una vez establecido el movimientoActual, vamos a comprobar si hay obstáculo en esa dirección, y si lo hay, cambiaremos el movimiento para ir hacia abajo.

```
public void Mover(Obstaculo[] obstaculos)
{
    int i;
    bool hayObstaculo;

    ... // Código anterior del método

    hayObstaculo = false;
    i = 0;
    while (i < obstaculos.Length && !hayObstaculo)
    {
        if (!obstaculos[i].GetDestruido() &&
            obstaculos[i].GetY() == GetY())
        {
            if (movimientoActual == movimientoCiempies.IZQUIERDA &&
                obstaculos[i].GetX() == GetX() - 1)
            {
                hayObstaculo = true;
                obstaculoActual = obstaculosCiempies.IZQUIERDA;
                movimientoActual = movimientoCiempies.ABAJO;
            }
            else if (movimientoActual == movimientoCiempies.DERECHA &&
                obstaculos[i].GetX() == GetX() + 1)
            {
                hayObstaculo = true;
                obstaculoActual = obstaculosCiempies.DERECHA;
                movimientoActual = movimientoCiempies.ABAJO;
            }
        }
        i++;
    }
}
```

Lógicamente, para que este método pueda recibir el array de obstáculos tenemos que pasárselo también al método `Mover` de la clase `Ciempies`:

```
public void Mover(Obstaculo[] obstaculos)
{
    for (int i = 0; i < bloquesCiempies.Length; i++)
    {
        Console.SetCursorPosition(bloquesCiempies[i].GetX(),
                                  bloquesCiempies[i].GetY());
        Console.Write(" ");

        bloquesCiempies[i].Mover(obstaculos);

        ...
```

Y, finalmente, cuando invoquemos a este método desde la clase `Partida` le pasamos el array:

```
private void MoverElementos()
{
    ciempies.Mover(obstaculos);
    ...
```

5.4. Detección de colisión ciempiés-nave

Finalmente, nos queda por detectar si el ciempiés choca con la nave. Para ello, vamos a añadir un nuevo método a la clase `Ciempies` donde le pasemos como parámetro un objeto de tipo Nave. Lo que haremos en este método es comprobar si alguno de los bloques no destruidos del ciempiés está en las mismas coordenadas que la nave, y devolver un booleano indicando si hay colisión o no:

```
public bool DetectarColisionConNave(Nave nave)
{
    bool colisiona = false;
    int i = 0;
    while (i < bloquesCiempies.Length - numBloquesPerdidos && !colisiona)
    {
        if (bloquesCiempies[i].GetX() == nave.GetX() &&
            bloquesCiempies[i].GetY() == nave.GetY())
            colisiona = true;
        i++;
    }
    return colisiona;
}
```

Faltaría llamar a este método desde la clase *Partida*, pero eso lo haremos en la siguiente sección para detectar el fin de partida.

6. Estado del juego y fin de partida (1,5 puntos)

Para terminar una versión más o menos básica y jugable del juego, nos quedan por pulir dos aspectos:

- Un marcador de puntos que se actualice cada vez que alcanzamos al ciempiés, o a algún obstáculo. Este punto no es indispensable, pero le da un aliciente más al juego.
- Definir unos indicadores de fin de partida, que nos permitan salir de la partida a la pantalla de créditos.

6.1. Marcador de puntos

Vamos a definir un marcador de puntos en una zona fuera del área de juego, por ejemplo, en el borde gris de la zona derecha. Para gestionarlo, en la clase Partida vamos a definir un atributo `puntos`:

```
class Partida
{
    private ConsoleKeyInfo tecla = new ConsoleKeyInfo();
    private Obstaculo[] obstaculos = CrearObstaculos(20);
    private Nave nave = new Nave(Configuracion.ANCHO_PANTALLA / 2,
        Configuracion.ALTO_PANTALLA - 1);
    private Ciempies ciempies = new Ciempies(10);
    private Disparo disparo = null;
    private int puntos = 0;

    ...
```

Definimos también un método `DibujarPuntos` para que los dibuje:

```
private void DibujarPuntos()
{
    Console.SetCursorPosition(Configuracion.ANCHO_PANTALLA -
        Configuracion.ANCHO_BORDE_LATERAL + 2, 1);
    Console.ForegroundColor = ConsoleColor.Blue;
    Console.BackgroundColor = ConsoleColor.Gray;
    Console.Write("Puntos: " + puntos.ToString("0000"));
    Console.ResetColor();
}
```

Llamaremos de momento a esta función en el bucle `do..while` de la partida:

```
public void Lanzar()
{
    Console.Clear();
    DibujarMarco();

    do
    {
        DibujarPuntos();
        ...
```

6.1.1. Incremento de los puntos

Incrementaremos los puntos cuando:

- Destruyamos un obstáculo
- Destruyamos un bloque de ciempiés

En la clase `Configuracion` podemos añadir un par de constantes con los puntos que ganaremos en cada caso:

```
class Configuracion
{
    public const int ANCHO_PANTALLA = 80;
    public const int ALTO_PANTALLA = 25;
    public const int PAUSA_BUCLE = 70;
    public const int VIDAS_INICIALES = 3;
    public const int ANCHO_BORDE_LATERAL = 20;
    public const int PUNTOS_OBSTACULO = 10;
    public const int PUNTOS_BLOQUE_CIEMPIES = 50;

    public static Random random = new Random();
}
```

> **TAREA**: Ahora vamos a modificar los métodos `ColisionDisparosObstaculos` en la clase `Partida` y `DetectarColisionConDisparo` de la clase `Ciempies` para que se incrementen los puntos cuando suceda cada una de estas dos cosas.

6.2. Fin de la partida

El fin de la partida (fin del bucle *do..while* del juego en la clase *Partida*) se puede dar por dos motivos:

- Porque la nave haya destruido al ciempiés
- Porque el ciempiés haya colisionado con la nave y la haya destruido

- Existiría una tercera forma: porque el ciempiés haya alcanzado la esquina inferior derecha del escenario, con lo que ya no puede moverse a ninguna otra parte. Tal y como hemos planteado el juego eso no es posible, porque antes colisionaría con la nave.

Vamos a declarar en la clase `Partida`, dentro del método `Lanzar`, una variable booleana que inicialmente será `false`. Haremos que el bucle, en lugar de ser infinito, se repita mientras esta variable siga siendo *false*:

```
public void Lanzar()
{
    bool finDePartida = false;

    Console.Clear();
    DibujarMarco();

    do
    {
        DibujarPuntos();
        DibujarElementos();
        EventosTeclado();
        MoverElementos();
        ComprobarColisiones();
    }
    while (!finDePartida && tecla.Key != ConsoleKey.Escape);
}
```

Una de las condiciones para terminar la partida es que el ciempiés quede destruido. Esto lo podemos comprobar fácilmente con el método `CiempiesDestruido` que habíamos añadido antes a la clase `Ciempies`. Otra forma de acabar la partida es que el ciempiés colisione con la nave. Esto lo podemos controlar con el método `DetectarColisionConNave` de la clase `Ciempies`.

> **TAREA**: llama a estos dos métodos desde el método `Lanzar` de la clase `Partida` para modificar adecuadamente el valor de la variable booleana.

7. Mejoras adicionales (2 puntos)

Como mejoras adicionales para completar el juego, se proponen las siguientes:

- (**0,5 puntos**) Añadir múltiples disparos (por ejemplo, hasta 5 a la vez)
- (**0,25 puntos**) Movimiento adicional de la nave: que se pueda mover hacia arriba y abajo en la región despejada inferior (últimas 4 filas)
- (**0,5 puntos**) Vuelta atrás del ciempiés: que cuando llegue al final de la pantalla (esquina inferior derecha; esto lo podemos conseguir si la nave se puede mover arriba y abajo), vuelva a empezar desde el principio (esquina superior izquierda)

- (**0,75 puntos**) Añadir múltiples niveles: cada vez que destruyamos el ciempiés, comenzaremos otra partida con un ciempiés más largo y con más obstáculos

8. Criterios de calificación

Aunque se han ido detallando los puntos en cada uno de los apartados anteriores, aquí aparecen agrupados y resumidos:

- Crear el proyecto y definir configuración y paso de pantallas: **1 punto**
- Definir la jerarquía de clases: sprite, nave, obstáculo y ciempiés: **1 punto**
- Definir la situación inicial de la pantalla de partida (bordes y dibujar elementos): **1 punto**
- Movimiento de los elementos (nave, ciempiés y disparos): **2 puntos**
- Detectar colisiones (disparo-obstáculo, disparo-ciempiés, etc): **1,5 puntos**
- Estado del juego y fin de partida (marcador de puntos y fin): **1,5 puntos**
- Mejoras adicionales propuestas: **2 puntos**