

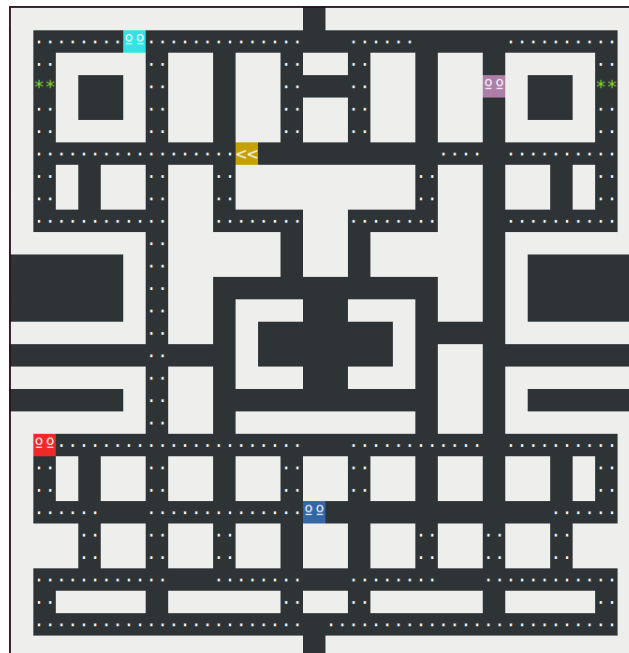
Fundamentos de la programación 2











Práctica 2. Pac-Man

Indicaciones generales:

- La línea 1 del programa (y siguientes) deben contener los nombres de los alumnos de la forma:
`// Nombre Apellido1 Apellido2`
- **Lee atentamente** el enunciado e implementa el programa tal como se pide, con las clases y métodos que se especifican, respetando los parámetros y tipos de los mismos. Adicionalmente pueden implementarse los métodos auxiliares que se consideren oportunos, comentando su cometido, parámetros, etc.
- El programa, además de ser correcto, debe estar bien estructurado y comentado. Se valorarán la claridad, la concisión y la eficiencia.
- La entrega se realizará a través del campus virtual, subiendo únicamente los archivos fuente `.cs` con el programa completo.
- El **plazo de entrega** finaliza el ... de abril.

En esta práctica vamos a implementar el conocido juego *Pac-Man* creado por Toru Iwatani en 1980¹. Puede jugarse online en <http://www.webpacman.com/> para familiarizarse su mecánica. Implementaremos una versión de consola que tendrá el siguiente aspecto:



En esta representación cada casilla ocupa dos caracteres en pantalla (como en juegos anteriores). Los muros se representan con casillas blancas, los pasillos con negras, los fantasmas como  ,  y , la comida como , las vitaminas como  y Pacman, dependiendo de su dirección de movimiento, será , ,  o .

Pacman se moverá según la dirección establecida en cada momento, que podrá cambiarse con las flechas de cursor. Los fantasmas inicialmente estarán encerrados en el rectángulo central con un muro que cierra ese recinto durante un lapso de tiempo prefijado. Los bordes izquierdo y derecho del

¹<https://es.wikipedia.org/wiki/Pac-Man>

tablero están conectados, de modo que si un personaje sale a la izquierda por un pasillo vuelve a aparecer por la derecha en la misma fila (lo mismo ocurre con los bordes inferior y superior). Cada vez que Pacman come una vitamina los fantasmas volverán a sus posiciones iniciales dentro del recinto. La partida termina cuando Pacman se come toda la comida y las vitaminas (gana), o bien cuando es capturado por un fantasma (pierde), o el jugador interrumpe el juego.

Para implementar el juego vamos a utilizar varias clases. En primer lugar, se proporciona ya implementada la clase `Coor` para representar coordenadas y direcciones en el plano. Esta clase tiene dos campos `fil` y `col` (fila y columna) como propiedades (son directamente accesibles con el operador `'.'`). Además define la sobrecarga de los operadores `'+'` y `'-'` que serán útiles para expresar "desplazamientos": dada una posición `pos=(fil,col)` y un vector de dirección `dir=(x,y)` podremos obtener directamente la posición resultante del desplazamiento como `pos+dir` (tanto `pos` como `dir` son de tipo `Coor` y el resultado será la nueva coordenada (`pos.fil+dir.fil`, `pos.col+dir.col`)). Por último, sobrecarga también los operadores `=='` y `!='` para poder comparar coordenadas de la manera habitual (en vez de comparar las referencias).

La clase `Tablero` que contendrá el estado actual del juego y los métodos apropiados para gestionar ese estado:

```
class Tablero{
    // contenido de las casillas
    enum Casilla {Libre,Muro,Comida,Vitamina,MuroCelda};

    // matriz de casillas (tablero)
    Casilla [,] cas;

    // representacion de los personajes (pacman y fantasmas)
    struct Personaje {
        public Coor pos, dir, // posicion y direccion actual
                ini; // posicion inicial (para fantasmas)
    }

    // vector de personajes, 0 es pacman y el resto fantasmas
    Personaje [] pers;

    // colores para los personajes
    ConsoleColor [] colors = {ConsoleColor.DarkYellow, ConsoleColor.Red,
        ConsoleColor.Magenta, ConsoleColor.Cyan, ConsoleColor.DarkBlue };

    const int lapCarcelFantasmas = 3000; // retardo para quitar el muro a los fantasmas
    int lapFantasmas; // tiempo restante para quitar el muro
    int numComida;    // numero de casillas restantes con comida o vitamina

    Random rnd; // generador de aleatorios

    // flag para mensajes de depuracion en consola
    private bool Debug = true;

    ...
}
```

Los niveles de juego se leerán de archivos de texto en la constructora de la clase, según veremos. La matriz de casillas `cas` determina el contenido de cada casilla (libre, muro, comida, vitamina, muroCelda). La información de los personajes (Pacman y fantasmas) se almacena en el vector `pers`. Para cada personaje se almacena su posición actual, dirección de movimiento y posición inicial (esta posición se utilizará para mandar a los fantasmas a casa cuando Pacman come una vitamina). El atributo `lapCarcelFantasmas` es el tiempo en milisegundos que permanecen los fantasmas encerrados su rectángulo central, `lapFantasmas` se utilizará para hacer una cuenta regresiva con ese tiempo de encierro y `numComida` es un contador de casillas con comida o vitaminas que se irá decrementando a medida que Pacman recorra el laberinto. El atributo `rnd` será el generador de números aleatorios que se utilizará para cambiar la dirección de movimiento de los fantasmas. Por último, el atributo `Debug` indica si estamos en *modo desarrollo* o *modo versión*; en el modo desarrollo se escribirá información en pantalla para depurar el programa (posiciones y direcciones de los personajes, etc) y se utilizará una semilla dada en el generador de números aleatorios, como veremos después.

Para desarrollar este juego es importante proceder ordenadamente, probando y depurando bien el código antes de pasar a la siguiente etapa. Será útil el manejo de excepciones en fragmentos críticos de código para delimitar la localización de posibles errores de ejecución.

1. Lectura de nivel y renderizado

En primer lugar, implementaremos los siguientes métodos de la clase `Tablero`:

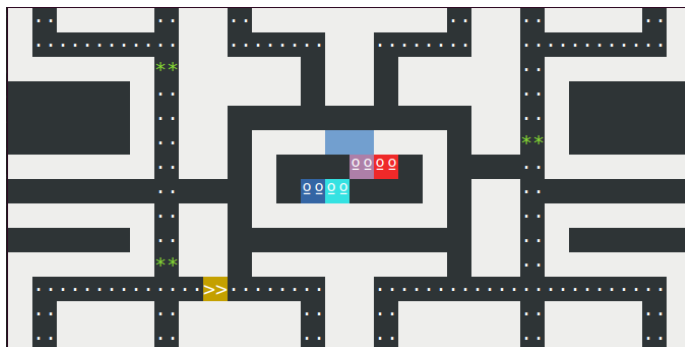
- `Tablero(string file)`: constructora de la clase. Toma como argumento el nombre de un archivo de nivel y carga dicho nivel en memoria. Los niveles se codifican mediante una matriz de enteros con el siguiente significado: 0 = Libre, 1 = Muro, 2 = Comida, 3 = Vitamina, 4 = MuroCelda, 5-6-7-8 = Fantasmas, 9 = Pacman. Pacman arrancará con dirección (0,1) y los fantasmas con dirección (1,0). Se inicializará `lapFantasmas` que comienza la cuenta regresiva para quitar el muro del recinto de los fantasmas. Nótese que a priori no sabemos el tamaño del tablero y habrá que determinarlo a partir del propio archivo de nivel.

En este método también se inicializará el generador de aleatorios `rnd`. Si el atributo `Debug` es `false` se inicializará normalmente, pero si es `true`, arrancaremos con una semilla dada `rnd = new Random(100)`; de este modo el generador de aleatorios producirá siempre la misma secuencia y tendremos *partidas reproducibles*, lo cual facilitará la depuración durante el desarrollo.

- `void Dibuja()`: dibuja en pantalla el estado actual del tablero. Este método debe producir información adicional de depuración si `Debug` es `true`, incluyendo al menos la posición y dirección actual de cada uno de los personajes.

Para probar estos métodos, en el método `Main` de la clase principal crearemos un tablero e invocaremos al método `Dibuja`. A continuación se muestra el archivo del nivel 0 y el aspecto que tendrá al sacarlo en pantalla con el método `Dibuja`:

```
1 2 1 1 1 1 2 1 1 2 1 1 2 1 1 1 1 1 1 1 2 1 1 1 1 2 1 1 1 1 2 1
1 2 2 2 2 2 2 1 1 2 2 2 2 2 1 1 2 2 2 2 2 1 1 2 2 2 2 2 2 1
1 1 1 1 1 1 3 1 1 1 1 1 1 0 1 1 0 1 1 1 1 1 2 1 1 1 1 1 1 1
0 0 0 0 0 1 2 1 1 1 1 1 0 1 1 0 1 1 1 1 1 2 1 0 0 0 0 0 0
0 0 0 0 0 1 2 1 1 0 0 0 0 0 0 0 0 0 0 1 1 2 1 0 0 0 0 0 0
0 0 0 0 1 2 1 1 0 1 1 1 4 4 1 1 1 0 1 1 3 1 0 0 0 0 0 0
1 1 1 1 1 1 2 1 1 0 1 0 0 0 6 5 0 1 0 0 0 2 1 1 1 1 1 1 1
0 0 0 0 0 0 2 0 0 0 1 0 8 7 0 0 0 1 0 1 1 2 0 0 0 0 0 0 0
1 1 1 1 1 1 2 1 1 0 1 1 1 1 1 1 1 1 0 1 1 2 1 1 1 1 1 1 1
0 0 0 0 0 1 2 1 1 0 0 0 0 0 0 0 0 0 0 1 1 2 1 0 0 0 0 0 0
1 1 1 1 1 1 3 1 1 0 1 1 1 1 1 1 1 1 0 1 1 2 1 1 1 1 1 1 1
1 2 2 2 2 2 2 2 9 2 2 2 2 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 1
1 2 1 1 1 1 2 1 1 1 1 1 2 1 1 2 1 1 1 1 1 2 1 1 1 1 2 1
1 2 1 1 1 1 2 1 1 1 1 1 2 1 1 2 1 1 1 1 1 2 1 1 1 1 2 1
```



2. Movimiento de Pacman

A continuación vamos a implementar el movimiento de Pacman y el cambio de dirección. Los métodos a implementar son:

- `bool Siguiente(Coor pos, Coor dir, Coor newPos)`: dada la posición `pos` y la dirección `dir` calcula la siguiente posición `newPos`, teniendo en cuenta que el personaje puede salir por un borde y aparecer por el opuesto si hay pasillo que los conecte (por ejemplo, salir por debajo y aparecer arriba). El método devolverá `true` si la casilla calculada en `newPos` no es muro (es decir, se puede avanzar a esa posición); `false` en caso contrario.
- `void MuevePacman()`: calcula la siguiente posición de Pacman utilizando el método anterior y lo desplaza a dicha posición si es posible (no hay muro). Además, si la casilla destino tiene comida o vitamina se la come.

- `bool CambiaDir(char c)`: dependiendo del valor del carácter `c` ('l', 'r', 'u', 'd'), calcula un nuevo vector de dirección para Pacman. Pero antes de hacer efectivo el cambio de dirección utilizará el método **Siguiente** para comprobar que Pacman puede efectivamente moverse en esa dirección. De este modo evitaremos que Pacman quede estancado en el laberinto si le mandamos en una dirección inviable.

El movimiento de Pacman tiene una particularidad que hace el juego *más jugable*: **tiene memoria**. Por ejemplo, si Pacman se desplaza a la derecha por un pasillo horizontal y le indicamos que cambie de dirección hacia arriba, no se produce ningún cambio inmediato, pero cambiará su dirección hacia arriba en cuando llegue a la primera bifurcación que se lo permita (pasillo ascendente). Este comportamiento evita que se quede bloqueado (si se hiciese efectivo el cambio de dirección automáticamente y no pudiese ir en esa dirección, tendría que pararse). Además, permite al jugador anticipar el siguiente cambio de dirección sin tener que *pulsar la tecla en el instante exacto* que pasa por la bifurcación. Además, si se indica otro cambio de dirección antes de que se haya hecho efectivo el anterior, nos olvidamos del cambio anterior y guardamos el nuevo. Por ejemplo, si Pacman se mueve a la derecha por un pasillo horizontal y se pulsa el cursor ascendente, y antes de cambiar la dirección se pulsa el cursor descendente, el efecto será que irá hacia abajo en el siguiente pasillo descendente. Para implementar este mecanismo utilizamos un *buffer* que guarda la *siguiente dirección*. En este caso el buffer contiene un solo carácter (`dir`) y se leerá con el siguiente método (en la clase principal):

- `void LeeInput(ref char dir)`: comprueba si ha habido alguna pulsación de teclado. Si se ha pulsado alguno de los cursores, devolverá en `dir` la dirección correspondiente ('l', 'r', 'u', 'd'). En otro caso, dejará `dir` con el valor de entrada (por eso pasa por referencia). Más adelante podrán reconocerse otras teclas para salir del juego, pausarlo, salvar el estado, etc.

Utilizando este método, junto con `CambiaDir` y `Dibuja` podemos hacer una primera versión del bucle principal del juego (en la clase principal, en el método `Main`):

```

Tablero t = new Tablero(...);
t.Dibuja();

int lap=200; // retardo para bucle ppal
char c=' ';

while (...) {
    // input de usuario
    LeeInput(ref c);

    // procesamiento del input
    if (c!=' ' && t.CambiaDir(c)) c=' ';
    t.MuevePacman();

    // IA de los fantasmas: TODO

    // rederizado
    t.Dibuja();

    // retardo
    System.Threading.Thread.Sleep (lap);
}

```

3. Movimiento de los fantasmas

En nuestra versión del juego vamos a hacer una IA elemental para los fantasmas: se moverán de manera aleatoria, sin importar la posición de Pacman. De hecho, en cada momento eligen aleatoriamente una de las direcciones posibles. Se excluye la dirección opuesta a la que llevan, para que no se *den la vuelta*, excepto cuando no queda otra opción (están en un callejón).

Para implementar este algoritmo utilizaremos la clase `ListaPares`. Es esencialmente la clase `Lista` vista en clase, pero almacenan elementos de tipo `Coor` en vez de enteros simples. Incorporan

además algún método adicional (como `cCoor nEsimo(int n)`) que será útil para nuestro juego y en principio no será necesario añadir ningún otro método.

Dado un fantasma, los pasos detallados del algoritmo para determinar su nueva dirección son:

1. Crear una lista vacía de pares `lst`.
2. Para cada una de las direcciones posibles $[(1, 0), (0, 1), (-1, 0), (0, -1)]$:
 - Si la posición *siguiente* (método **Siguiente** descrito arriba) en esa dirección está libre (no hay muro) y no contiene fantasma insertamos dicha posición en `lst`.
3. Si `lst` solo contiene una dirección (el fantasma está en un callejón) dejamos la lista tal cual. Pero si hay más de una dirección en la lista, eliminamos de dicha lista la dirección opuesta a la que tiene el fantasma en ese momento (para que no se dé la vuelta).
4. Por último, elegimos aleatoriamente una de las direcciones que quedan en `lst`.

Así pues, en la clase **Tablero** implementaremos los siguientes métodos:

- `bool HayFantasma(Coor c)`: determina si la posición `c` contiene un fantasma.
- `int PosiblesDirs(int fant, out ListaCoor lst)`: dado un fantasma `fant` calcula en `lst` la lista de posibles direcciones de acuerdo con explicado arriba. Devuelve el número de direcciones posibles (la longitud de la lista `lst`).

Nota: para poder depurar es importante considerar las direcciones en un orden prefijado, como $(1, 0), (0, 1), (-1, 0), (0, -1)$.

- `void SeleccionaDir(int fant)`: utiliza el método anterior para obtener la lista de posibles direcciones para el fantasma `fant` y después elige una aleatoria (será útil el método `nEsimo` de la clase `ListaPares`).
- `EliminaMuroFantasmas()`: cambia el contenido de las casillas `MuroCelda` a `Libre`, i.e., abre el muro de la celda de los fantasmas y estos quedan liberados.
- `void MueveFantasmas(int lap)`: para cada uno de los 4 fantasmas selecciona su dirección de movimiento con `SeleccionaDir` y lo mueve en esa dirección si es posible (si **Siguiente** devuelve `true`).

Recordemos que inicialmente `lapFantasmas = 3000`, i.e., los fantasmas están encerrados durante 3000 milisegundos en su celda. Con el valor `lap` vamos descontando tiempo, hasta que llega la hora de liberar a los fantasmas (con `EliminaMuroFantasmas`).

Ahora, desde el bucle principal podremos invocar al método `MueveFantasmas` para visualizar y probar el movimiento de los fantasmas.

Además, modificaremos el método `MuevePacman` implementado en el apartado anterior, de modo que, cuando Pacman se coma una vitamina los fantasmas vuelvan a su posición inicial (guardadas en el personaje).

4. Colisiones y final de nivel

El nivel terminará cuando Pacman se come toda la comida del nivel o es capturado por un fantasma. Necesitaremos métodos para detectar ambas situaciones:

- `bool Captura()`: determina si hay algún fantasma en la misma posición que Pacman.

- `bool finNivel()`: comprueba si Pacman se ha comido toda la comida del nivel. Esto es inmediato utilizando el atributo `numComida`.

Con estos dos métodos podemos ahora controlar el bucle principal del juego. Tal como estamos diseñando el juego, hay dos posibilidades de colisión (captura). Puede producirse tras el movimiento de Pacman (Pacman colisiona contra un fantasma) o bien, tras el movimiento de los fantasmas (un fantasma atrapa a Pacman). Ambas situaciones deben controlarse en el bucle principal y el resultado será el mismo: termina el bucle. También terminará si Pacman se ha comido toda la comida.

5. Opcional: Paso de niveles, pausa de juego, salvar partida...

Una vez implementado en núcleo central del juego, en la clase `Main` añadiremos la funcionalidad necesaria para completarlo. El juego debe arrancar con el nivel inicial y cargar el siguiente nivel cuando el jugador supera el actual. Pueden incorporarse opciones para pausar el juego con una tecla, salvar el estado actual de la partida para continuar después, etc.

Puede también guardarse información personalizada para distintos jugadores con los niveles que ya han superado, de modo que cuando un jugador arranca el juego y se identifica, puede arrancar con el primer nivel no superado.

Otra opción interesante sería el **diseñador de niveles**, una funcionalidad que permita diseñar nuevos niveles de modo interactivo. Para ello se puede partir de un tablero formado únicamente por muros y permitir al usuario desplazarse por el mismo, excavando pasillos, añadiendo los personajes, la comida, etc, y por último salvando el nivel en el formato explicado, para poder después cargarlo en el juego.