Grado en Ingeniería Informática

Metodología de la Programación

Sesión 4

Clases y objetos

Docentes:

Raúl Marticorena

Índice de contenidos

1. Introducción

En esta sesión se plantea la construcción de una aplicación en Java que permita jugar a dos contrincantes al clásico juego del **tres en raya** (conocido como *tic-tac-toe* en otros países). La solución dada debe seguir el **paradigma de la Programación Orientada a Objetos (POO)**.

El juego se desarrolla en un tablero situado horizontalmente que tiene 9 celdas (3 filas x 3 columnas). Los jugadores, por turnos, colocan una pieza con un color o forma sobre una de las celdas vacías del tablero.

Ilustración 1: Juego del tres en raya

El juego finaliza cuando uno de los jugadores consigue colocar tres piezas en celdas contiguas en la dirección horizontal, vertical o en una de las dos diagonales, siendo éste, el ganador del juego. En caso de llenarse el tablero con 9 piezas, sin darse la anterior condición, se finaliza el juego en tablas, sin tener un ganador.

2. Objetivos

- Construir las clases correspondientes a la implementación de un juego de tres en raya.
- Generar los objetos necesarios en tiempo de ejecución para poder jugar al tres en raya.

3. Contenidos específicos del tema

A continuación se introduce primero el concepto de tipos enumerados en Java para a continuación desglosar el diseño y solución para la aplicación, con los distintos modelos de clases.

3.1 Previo: Tipos enumerados

Para la resolución del juego usaremos adicionalmente un nuevo elemento en Java, los **tipos enumerados**.

Una enumeración permite definir un tipo con un conjunto restringido de valores nominados (ver [Eckel, 2007] o la documentación en línea en https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html).

A continuación se presenta un ejemplo de definición de un tipo enumerado para definir calificaciones de asignaturas (ver Código 1).

```
public enum Calificación {
   SUSPENSO, APROBADO, NOTABLE, SOBRESALIENTE, MATRÍCULA_DE_HONOR;
}
```

Código 1: Ejemplo de tipo enumerado en Java

En Java, aunque en el código fuente se declara con `enum`, a nivel de implementación y *bytecodes*, son realmente clases[1]. En relación a los tipos enumerados [McLaughlin and Flanagan, 2004], se caracterizan por las siguientes propiedades:

- Extender o heredar de `java.lang.Enum` por defecto (el concepto de herencia se verá en temas posteriores).
- No tener constructores públicos.
- Contener solo valores **`public`**, **`static`** y **`final`**.
- No representar valores numéricos enteros.
- Poder ser comparados con == o `equals` **indistintamente**.
- El método `toString()` está redefinido, devolviendo el texto correspondiente de la enumeración (aunque puede ser a su vez redefinido). Ejemplo: `Calificación.NOTABLE.toString()` devuelve la cadena de texto `"NOTABLE"`.
- Simétricamente se puede utilizar el método `valueOf` para obtener el valor correspondiente. Ejemplo: `Calificación.valueOf("SUSPENSO")` devuelve `Calificación.SUSPENSO`.
- Disponer de un método `values()` que devuelve un *array* estático de elementos del tipo enumerado, con todos sus valores.
  - **Muy útil para realizar recorridos sobre todos los elementos de la enumeración, en particular con los bucles for-each.**
- Definir un método `ordinal()` que devuelve la posición entera de cada valor enumerado empezando en cero, según el orden de declaración.
  - **No se recomienda su empleo, puesto que modificaciones de los valores de la enumeración, hacen muy frágil el código cliente.**
- Cuando se utilizan en instrucciones *switch-case* el compilador obliga a NO colocar como prefijo el nombre de la clase enumeración. Ejemplo: `case APROBADO:` en lugar de `case Calificación.APROBADO:`.
- Es posible definir otro métodos e incluso constructores (estos últimos deben ser **privados**).

1Se deja como ejercicio el uso del decompilador (`javap`) para comprobar esta afirmación.

3.2 Modelo de clases

El modelo de clases inicial deducido de la observación del mundo real (del juego) se muestra en la Ilustración 2, a falta de completar con alguna clase adicional, como el árbitro que controla el juego (ver Ilustración 3).

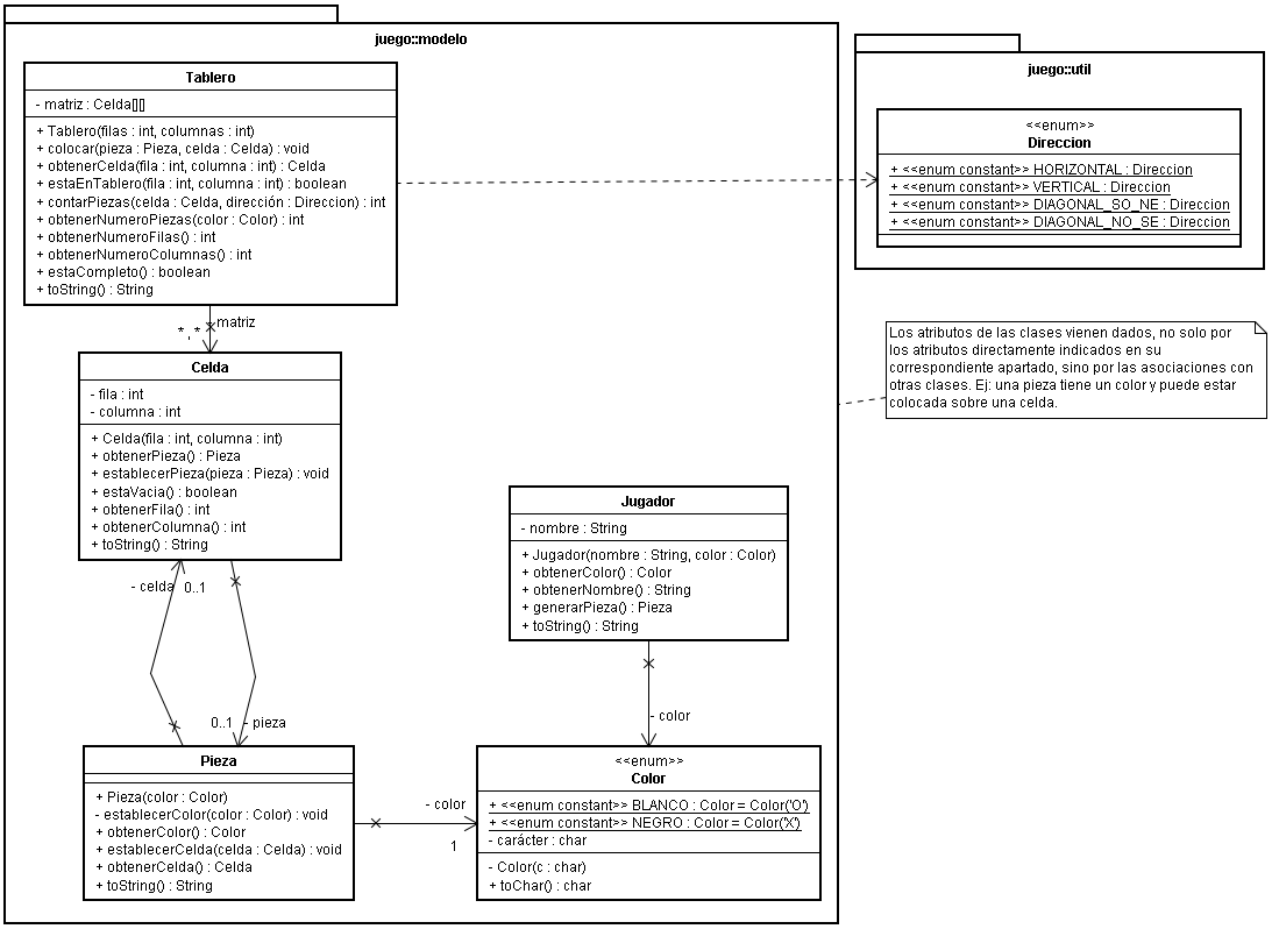

Ilustración 2: Diagrama de clases basico del paquete `juego.modelo` y `juego.util`

Los métodos con nombre de la forma `consultarX` u `obtenerX`, devuelven o retornan el valor del atributo `X` consultado (bien el valor almacenado en memoria o bien un dato calculado). Estos métodos solo consultan y NUNCA deben modificar el estado del objeto.

Los métodos `establecerX` cambian el estado del objeto asignando al atributo X el nuevo valor pasado como argumento. Estos métodos sí cambian el estado del objeto.

Esta convención es un estándar en Java, conocido como el uso de métodos getter y setter (obtener vs. Establecer) para manejar los atributos a través de métodos siguiendo el principio de encapsulación.

Los métodos `toString` devuelven una cadena de texto equivalente al estado actual del objeto y permiten mostrar su estado normalmente en pantalla a través de `System.out.printf` o `System.out.print`.

Se comentan a continuación aquellos métodos en cada clase que tienen una funcionalidad más complicada:

Clase `Celda`:

- `estaVacia`: consulta si la celda tiene o no asignada una pieza. Devuelve `true` en caso de estar vacía y `false` en caso contrario.

Enumeración `Color`:

- `Color`: constructor privado que permite inicializar el carácter o letra almacenado en la enumeración. Por lo tanto tendrá un atributo privado para almacenar adicionalmente dicha letra.
- `toChar`: devuevle la letra asignada al color.

Enumeración `Direccion`:

- Sin comentarios.

Clase `Jugador`:

- `generarPieza`: devuelve un nuevo objeto `Pieza` del color correspondiente al jugador actual sobre el que se invoca.

Clase `Pieza`:

- Sin comentarios.

Clase `Tablero`:

- `colocar`: toma el objeto `Pieza` y lo asigna a la `Celda` correspondiente. El proceso debe ser simétrico asignando también a la `Celda` esa `Pieza`.
- `estaEnTablero`: devuelve `true` si las coordenadas pertenecen a los límites del tablero y `false` en caso contrario. Por ejemplo en un tablero 8x8 solo son validas coordenadas entre (0,0) a (7,7).
- `contarPiezas`: dada una `Celda` y una `Direccion` concreta, cuenta el número de piezas consecutivas del mismo color a partir de la `Celda` de inicio. Para ello se debe comprobar en ambos sentidos, en la dirección dada.
  - Si la `Celda` de inicio está vacía el resultado es 0.
  - Si la celda de inicio no está vacía el resultado mínimo es 1 y a partir de ahí se incrementará (o no) según el número de piezas del mismo color consecutivas a partir de dicha celda.
  - Se vuelve a recordar que hay que revisar en AMBOS sentidos de la `Dirección` dada.
    - Pista: para moverse en el tablero en filas y columnas, se puede incrementar/decrementar el contador de filas/columnas de uno en uno Si queremos movernos en horizontal o vertical (no en diagonal) basta con sumar/incrementar cero según queramos desplazarnos.
- `obtenerNumeroPiezas`: devuelve el número de piezas actuales en el tablero del `Color` indicado. En un tablero vacío el resultado siempre es cero, con independencia del color pasado. En un tablero lleno, el número coincide con el número de celdas.
- `estaCompleto`: devuelve true si el tablero está ocupado o bien false si queda alguna celda vacía.

A continuación se muestra el diagrama de clases para el paquete de control y se comentan los métodos de la clase `ArbitroTresEnRaya`:

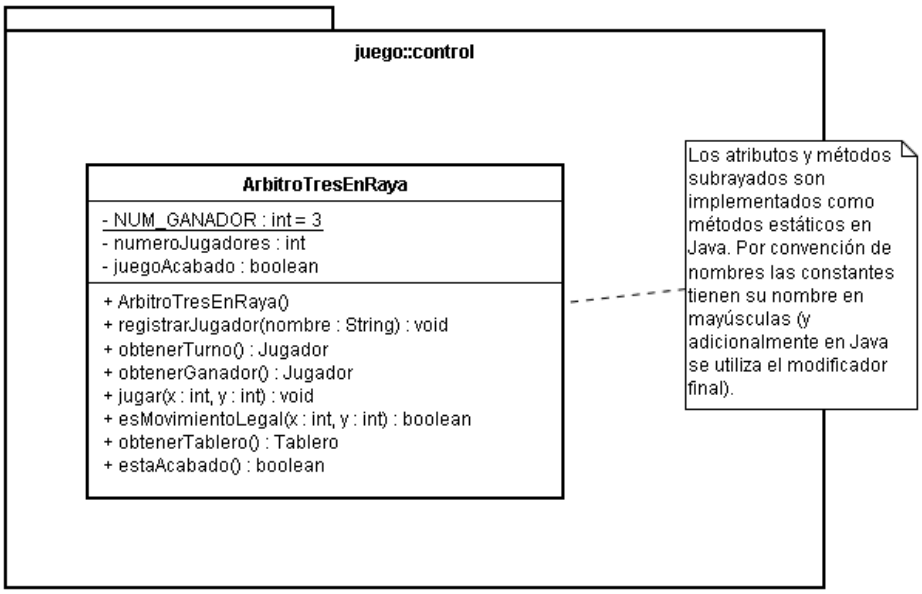

Ilustración 3: Diagrama de clases con el arbitro del paquete `juego.control`

Clase `ArbitroTresEnRaya`:

- `jugar`: dada una posición del tablero, coloca una pieza del jugador con turno actual en dicha posición. Realizada la jugada, el turno debería cambiar al jugador contrario y puede ser el momento adecuado para comprobar si la partida ha finalizado o no. Por simplicidad se asume que no es necesario comprobar si es legal o no jugar en esa posición (se supondrá que siempre se ha comprobado previamente por otro objeto, (en nuestro caso en el `main` de la clase `TresEnRaya` que se pide como ejercicio posteriormente), que era legal el movimiento antes de invocar a este método).
- `esMovimientoLegal`: consulta si es legal o no colocar una pieza en dicha posición. Si la celda está vacía y está en los límites del tablero, será legal. Si la partida ha finalizado, no será legal realizar ya ningún movimiento.
- `estaAcabado`: comprueba que la partida ha finalizado bien por victoria o por tablas.
- `registrarJugador`: registra al jugador (creando un nuevo objeto Jugador con el nombre proporcionado) según el orden de llamada. La primera vez registra al primer jugador. La segunda al segundo jugador, y ante siguientes llamadas, simplemente no se hace nada, puesto que los dos jugadores ya están "registrados". Cuando se registra al segundo jugador (cuando se invoca por segunda vez) se debería establecer el turno siempre al primer jugador registrado.
- `obtenerTurno`: devuelve el jugador con turno actual. Según se vayan realizando movimientos este turno irá alternando de valor entre los dos jugadores.
- `obtenerGanador`: devuelve el jugador que tiene tres piezas de su color consecutivas o `null` en caso de que no haya todavía un ganador.

El diagrama muestra las clases y métodos públicos de los módulos, y adicionalmente algunos atributos privados (no todos, **algunos se deducen de las asociaciones**). Es decisión del alumno implementar el resto de propiedades (atributos y métodos) con modificadores de acceso más restrictivos ("amigable" y fundamentalmente `private`).

El diagrama en la Ilustración 2 muestra también las asociaciones entre las clases del modelo `juego.modelo` y `juego.util`, quedando pendientes de resolver las dependencias con el paquete `juego.control` mostrado en la Ilustración 3.

3.3 Aclaraciones a los diagramas UML

Para la realización de los modelos se han utilizado diagramas de clases que siguen la notación UML (Unified Modeling Language), donde las clases se representan a través de cajas con tres partes (nombre de la clase, atributos y métodos).

Las clases se agrupan en **paquetes** representados con iconos de carpetas. Los nombres de paquetes en UML, se escriben de forma similar a como se hace en Java, pero utilizando como separador `::` en lugar del punto.

Los símbolos - y + en atributos y métodos, indican el nivel de acceso para las propiedades: `private` o `public` respectivamente (simplificamos el ejercicio utilizando solo esos dos modificadores de acceso).

Además se dibujan líneas que conectan las clases entre sí, denominadas **asociaciones**. Las asociaciones representan las relaciones entre los objetos generados. Se suele indicar la **multiplicidad** de la asociación con un par de números en los extremos, indicando el número mínimo y máximo de objetos implicados. Si solo hay un número se interpreta como multiplicidad mínima y máxima. Si el número es indeterminado (sin límite) se utiliza un símbolo *.

Por ejemplo si entre la clase `Pieza` y `Color` hay una flecha, y en el extremo que apunta a `Color` aparece un 1 quiere decir que un objeto de tipo `Pieza` tendrá un color (no puede ser que no tenga color, ni que tenga dos colores asociados a la vez).

Las relaciones con final en punta de flecha, indican que la **navegación** de la relación **solo permite ir en ese sentido** (un objeto conoce al otro, pero no al revés) y se traduce que en el código de la clase tendremos un atributo de dicho tipo, permitiendo la navegación, pero en la otra clase no existirá atributo que permita la navegación contraria. En nuestro ejemplo, un `Color` no tiene atributos de tipo *array* o colección de `Pieza`, puesto que no sabe (ni nos interesa en este caso) a qué piezas se ha asignado dicho color.

En UML, el tipo de retorno de los métodos o el tipo de los atributos, se coloca a la derecha de la signatura de los métodos (al revés que en Java, que va a la izquierda).

Finalmente, los cuadros con una esquina doblada, son simplemente notas o comentarios aclaratorios, que se pueden vincular a cualquier elemento en UML (similar al concepto de los comentarios en nuestro código fuente o comentarios/notas en un documento de texto).

Para una descripción más detallada de UML, se recomienda consultar la bibliografía complementaria.

3.4 Ejercicios

1. Escribir en una hoja el árbol correspondiente a la estructura de paquetes y clases indicada en el enunciado. Construir el correspondiente árbol de directorios y ficheros `.java` físicamente en el ordenador.
2. Establecer el orden de implementación de las clases mostradas en el diagrama (enfoque *bottom-up* de menor a mayor número de dependencias) e implementar las clases del paquete `juego.modelo` y `juego.util`.
    - Nota: aunque en un juego de tres en raya el tablero es de 3x3 y la multiplicidad debería ser 9, en el diagrama de clases se ha dejado abierto (*,*) con vistas a que el tablero sea **reutilizable con otros tamaños, incluso tableros rectangulares**. Para ello, en su implementación se debe utilizar un array de dos dimensiones, de tipo objetos de tipo `Celda`, y permitir dimensionar su tamaño en filas y columnas, en el constructor de la clase.
    - En un primer intento se puede omitir el método `contarPiezas`, dada su complejidad inicial, aunque será necesaria su implementación cuando se aborde la implementación definitiva de la clase `ArbitroTresEnRaya`.
    - Como pista, deberíamos ir en orden de los más simple a lo más complejo: de aquellas clases que dependen menos de otras, a aquellas que dependen de todas las demás. En último lugar deberíamos abordar `ArbitroTresEnRaya` y la interfaz en modo texto `TresEnRaya`.

3. Con las clases del paquete `juego.modelo` y `juego.util`, ya implementadas, probar la correcta implementación añadiendo un método `main` a la clase `Tablero` que:

- genere una instancia de un tablero 3 x 3
- obtenga las celdas de la diagonal `NO_SE`.
- genere tres instancias de pieza,de un color cualquiera (por ejemplo, negras).
- coloque las tres piezas en las tres celdas de la diagonal `NO_SE`.
- muestre el contenido actual del tablero en pantalla, una vez colocadas las tres piezas[2].

Ej:

X--

-X-

--X

Nota: el uso de métodos `main` en una clase para implementar "pruebas" sobre nuestra clase, tal y como estamos haciendo aquí sobre la clases `Tablero` (e indirectamente sobre `Celda`, `Pieza`, `Color`, etc.) es un **modo "manual" de realizar pruebas unitarias y de integración sobre nuestras clases.** Dado que en los objetivos de esta asignatura **NO** se incluye la programación de pruebas automáticas utilizando herramientas más avanzadas (e.g. JUnit en Java como estándar *de facto*), se recomienda utilizar esta técnica como solución alternativa, cuando se quiera probar la funcionalidad de una clase y no se quiera trabajar con las pruebas automáticas proporcionadas por los profesores de la asignatura.

4. Completar la implementación con la clase: `juego.control.ArbitroTresEnRaya`.
5. Plantear el pseudocódigo del juego del tres en raya en base a los objetos y acciones previamente diseñados.
6. Implementar dicho pseudocódigo con una última clase de nombre: `juego.textui.TresEnRaya` que contiene el método raíz `main`. El método `main` contiene el orden temporal de creación e invocación de métodos sobre objetos: inicializando el juego e iterando hasta que se finaliza la partida.

   Inicialmente supondremos que los jugadores siempre se llaman Abel y Caín y que siempre inicia el turno Abel.

   El interfaz en modo texto debe ser similar al presentado a continuación:

```
0   ---
1   ---
2   ---
    012
El turno es de: Abel con fichas O
Introduce columna:2
Introduce fila:0
0   --O
1   ---
2   ---
    012
El turno es de: Caín con fichas X
Introduce columna:1
```

2 Para convertir un `char` a `String`, se puede utilizar la clase de envoltura `Character` con su método `toString`. Ej: `String s = Character.toString('a'); // s = "a";`

```
Introduce fila:1
0   --O
1   -X-
2   ---
    012
El turno es de: Abel con fichas O
```

Para leer valores de teclado y convertirlos a enteros se realiza de la siguiente forma:

```
Scanner scanner = new Scanner(System.in); // objeto que engancha con el teclado
int var = scanner.nextInt(); // lee un valor entero de teclador
// usar el scanner tantas veces como se requieran lecturas de teclado
...
// finalmente cuando no tengamos que realizar más lecturas, debe ser invocado solo
// una vez
scanner.close();
```

7. Modificar el código de `juego.textui.TresEnRaya` para leer los nombres de los dos jugadores de la propia invocación de línea de comandos. En Java se toman los valores del *array* `args` recibido como argumento en el método `main`.
8. En Java una constante global se declara añadiendo los modificadores: **`static final`** (Ej: **`public static final int`** `MAX = 10;`), realizar los cambios necesarios para utilizar constantes simbólicas en el código anterior[3].

4. Resumen

En esta sesión de prácticas se ha presentado el **primer caso de estudio de una aplicación orientada a objetos**. Como resultado se debe desarrollar una primera versión de un juego de tres en raya implementado en Java, con una primera experiencia en el desarrollo de clases y objetos en tiempo de ejecución, que llevan a cabo la realización del juego.

5. Bibliografía

[Arnold et al., 2001] Arnold, K., Gosling, J., and Holmes, D. (2001). El Lenguaje de Programacion JAVA. Serie Java. Pearson Educacion – Addison Wesley, 3a edition.

[Eckel, 2007] Eckel, B. (2007). Piensa en Java. Prentice Hall, 4 edition

[McLaughlin and Flanagan, 2004] McLaughlin, B. and Flanagan, D. Java 1.5 Tiger. A Developer's Notebook. O'Reilly.

[Prieto el al., 2012] Prieto, N., Casanova, A., Marqués, F. Llorens, M., Galiano, I., Gómez, J.A., González, J., Herrero, C., Martínez-Hinajero, C., Moltó, G. y Piris, J. (2012) Empezar a programar usando Java. Editorial Universitat Politécnica de Valencia

3 Los nombres de constantes se escriben con mayúsculas en Java, por convención de nombres.

6. Recursos

Bibliografía complementaria:

[Holub, E. 2019] Allen Holub's UML Quick Reference. Disponible en https://holub.com/uml/[4]

[Oracle, 2017] Lesson: Language Basics. The Java Tutorials. Enum types (2017). Disponible en http://docs.oracle.com/javase/tutorial/java/javaOO/enum.html

4 Aunque la referencia realiza un resumen de todo UML, debéis centraros en los apartados de diagramas de clases. UML es muy amplio, y nuestro objetivo es solo utilizar los elementos básicos para representar las clases en diagramas.

Licencia

Autor: Raúl Marticorena

Área de Lenguajes y Sistemas Informáticos

Departamento de Ingeniería Civil

Escuela Politécnica Superior

UNIVERSIDAD DE BURGOS

2019