# Práctica de Programación Orientada a Objetos
## Convocatorias de Junio y Septiembre

Este enunciado describe la funcionalidad de la práctica para las convocatorias de junio y septiembre. Dado que la metodología de prácticas para estas convocatorias es diferente a la convocatoria de febrero, se presenta la práctica en un solo enunciado. No obstante, se recomienda realizar un desarrollo incremental, tal como se hizo en la convocatoria de febrero. Asimismo, se podrán concertar entrevistas a petición de los alumnos para supervisar el trabajo.

## 1. Descripción general del juego

El juego se desarrolla en escenarios inundados de agua. Al estar en un entorno líquido, supondremos que las entidades no sufren el efecto de la gravedad, pero sí el efecto de las corrientes en cada planta del escenario. En una planta hay una corriente hacia la derecha o izquierda siendo su sentido alterno entre plantas contiguas. Las corrientes afectan a las entidades cuando decidan no realizar un movimiento.

Un escenario da paso a otros escenarios a través de una serie de puertas. Éstas se encuentran ocultas y sólo son abiertas por la explosión de una bomba cercana. Una puerta abierta representa un sumidero en el escenario, lo que implica que en esa planta las corrientes estarán dirigidas hacia el sumidero con el propósito de atraer a las entidades hacia la puerta. Establecemos como restricción que sólo podrá haber un sumidero por planta. Los escenarios a los que se accede a partir de otro están determinados y se establecen en un fichero de configuración. Por ejemplo, si desde el escenario 2 se puede acceder al 8, 4 y 1, entonces en el escenario 2 habrá tres puertas situadas de manera aleatoria en cada ejecución del juego. El juego se da por finalizado al atravesar una puerta que no lleve a ningún escenario. Por tanto, el objetivo del juego es escapar del laberinto de escenarios en el menor tiempo posible.

Una entidad situada encima de una puerta abierta (sumidero) pasa al escenario destino. Nótese que los escenarios existen durante toda la ejecución del juego. Esto quiere decir que si hemos estado en un escenario y han quedado puertas abiertas y otras entidades, el escenario se encontrará en el mismo estado si regresamos a través de otros escenarios. No obstante, sólo estará activo el escenario en el que se encuentre el jugador. Entendemos por escenario activo aquel donde a las entidades se les aplica un turno de ejecución.

## 2. Entidades del juego

El **jugador** sólo tiene una vida y asociado a ella un nivel de vida que varía de 0 a 10. Un nivel 0 significa que el jugador está muerto, mientras que un nivel 10 indica que el jugador está sano. Cualquier otro valor implica que el jugador está herido y podrá ser perseguido por *entidades depredadoras*. El jugador pierde vida cuando entra en contacto con determinadas entidades del juego y recupera vida alimentándose de peces comunes. La acción de *comer* un pez ha de ser explícita, es decir, indicada por el usuario del juego cuando el jugador se sitúe sobre una entidad que sea comestible. El jugador podrá ser *infectado* por ciertas entidades del juego. En este estado irá perdiendo vida paulatinamente, siendo el único modo de superar la infección un cambio de escenario.

En el juego existen diversos **tipos de entidades**. Las entidades consideradas *vivas* (animadas) son las siguientes:

- **Pez común**. Se caracteriza por el movimiento errático *típico de los peces*. Es decir, en un momento dado puede decidir no moverse o moverse, y en este último caso en cualquiera de las direcciones posibles. Además, es el único pez comestible.
- **Pez Espada**. Se mueve como cualquier pez y destaca por hacer estallar burbujas cuando choca con ellas.
- **Tiburón**. Es un pez depredador que persigue a cualquier entidad *herida*. Si en el escenario hubiera varias, perseguirá a la que tenga menos vida. Si no hay ninguna se mueve como cualquier pez. Reduce la vida de las entidades heribles por contacto.
- **Medusa**. Su movimiento es alterno buscando el techo o el fondo del escenario. Por tanto, su sentido de movimiento será hacia arriba o abajo, siempre que sea posible. También reduce el nivel de vida de las entidades que pueden ser heridas.
- **Vampiro**. Es una entidad con la capacidad de ser herida, es decir, que tiene un nivel de vida. Cuando está herida, infecta a cualquier entidad viva del escenario que entre en contacto con ella. No tiene movimiento autónomo, y sólo se mueve por las corrientes. Tras su muerte, revivirá transcurrido un periodo de tiempo aleatorio y se situará en la posición en la que murió. Una entidad infectada por un vampiro muere al cabo de un tiempo, a no ser que cambie de escenario. Finalmente, se alimenta de cualquier entidad comestible con el objetivo de recuperar su nivel de vida.

Durante el desarrollo del juego el jugador hace uso de dos **armas**: *bombas* y *burbujas*. Las bombas son utilizadas como medio para matar a las entidades del juego y descubrir las puertas a otros escenarios. Por otro lado, las burbujas pueden servir para dejar atrapadas a ciertas entidades del escenario, como por ejemplo, los peces comunes para conservar el alimento.

Las *bombas*, al igual que cualquier otra entidad del juego, se ven afectadas por las corrientes. Como consecuencia del estallido de una bomba: (i) se destruyen los soportes superior e inferior de la posición donde se encuentre la bomba, (ii) mueren todas las entidades en un cierto *radio de acción* y (iii) explotan todas las burbujas en otro radio de

acción más amplio. Por último, la explosión de una bomba cambia el sentido de las corrientes de todo el escenario, excepto las corrientes que van a un sumidero.

Las *burbujas* son soltadas en una posición contigua al jugador y siguen el curso natural de las corrientes. Tienen la capacidad de atrapar a la primera entidad viva que encuentren en su camino. Cuando una entidad es atrapada, su movimiento y acción quedan inhibidos hasta que quede liberada. Esta situación se produce por la acción de un pez espada o por la explosión de una bomba, y siempre que no se encuentre en el radio de acción que provoca la muerte de la entidad. Las burbujas se desplazan hacia el techo. Esto significa que ascienden a la planta superior siempre que sea posible. Finalmente, el jugador sólo podrá lanzar una burbuja cuando haya transcurrido al menos 3 segundos desde el último lanzamiento.

## 3. Escenario

Los escenarios del juego están formados por una matriz bidimensional de celdas. Cada entidad ocupará una celda y al desplazarse pasará a una celda contigua. Todas las celdas del escenario, excepto las de la primera fila, pueden tener soporte en su parte inferior. Con ello, el escenario queda organizado en plantas donde la corriente circula en un sentido.


Un **fichero de configuración** que describa un escenario debe incluir información sobre las dimensiones, las celdas que tienen soporte, los escenarios destino y las entidades del escenario. A continuación se muestra el formato de un fichero de configuración. El orden de las declaraciones es estricto y los comentarios serán ignorados durante el procesamiento.

```
# Dimensiones, ancho x alto

8x5

# Soportes, se ignora la primera planta

0 1 1 0 1 1 1 0
1 1 0 1 1 0 1 1
0 1 1 1 0 1 1 0
1 1 0 1 1 0 1 1

# Puertas

3, 5, 8

# Entidades, se establece la posición (X, Y)

Jugador (0, 0)

PezEspada (1, 0)

PezComun (3, 3)

PezComun (4, 4)

…
```

El propósito del juego es ofrecer la ficción de que las entidades en un escenario están en continua actividad. Con este objetivo el escenario también hará las funciones de *planificador del juego* de modo que cuando sea avisado por un temporizador recorrerá todas las entidades aplicando sus turnos. Para ello se hará uso del **Temporizador** visto en clase. El periodo de notificación es el factor que condiciona la velocidad del juego. Cuanto menor sea este periodo, más rápido irá el juego. Para poder poner en marcha el hilo que despierte al escenario para la asignación de turnos y poder pararlo (parando de este modo todas las entidades del juego) la clase ofrecerá dos métodos. Finalmente, en el juego sólo el escenario en el que se encuentre el jugador estará activo, es decir, dará turnos a las entidades.

## 4. Partida

La Partida representa la fachada del juego desde la interfaz de usuario. Es responsable de cargar los escenarios del juego y controlar los cambios de escenario del jugador. Además ofrece funcionalidad para actuar sobre el jugador. Asimismo se encarga de controlar el tiempo empleado por el jugador en salir de un escenario y el tiempo total en escapar del laberinto de escenarios.

Se supone que los escenarios residen en una carpeta de la aplicación con nombre "escenarios". Los escenarios deben nombrarse utilizando números y han de tener la extensión *.esc*. La partida comienza por el escenario 1 y avanza cuando el jugador pase por alguna de las puertas del escenario actual. Se cargará un escenario del disco sólo si el jugador es la primera vez que llega a ese escenario. Es decir, si el jugador pasa a un escenario previamente visitado, lo encontrará en el estado en el que lo dejó. Se entiende

que la partida finaliza con éxito si una puerta lleva a un escenario que no existe en la carpeta de escenarios. Al terminar se deberá informar del tiempo que ha tardado el jugador en escapar del laberinto de escenarios.

La clase ofrece un método para arrancar la partida. En este método inicia un temporizador para controlar el tiempo transcurrido en el escenario actual y otro para el tiempo total del juego, y arranca el primer escenario. Los objetos partida también deben ofrecer un método para parar, es decir, detener el escenario actual y los temporizadores, y otro reanudar para continuar con la ejecución.

Todos los escenarios tienen el mismo tiempo límite, de manera que si transcurre dicho tiempo sin que el jugador haya salido del escenario, la partida termina sin éxito. Cuando un jugador vuelve a visitar un escenario, el tiempo límite para salir se establece al máximo, es decir, no se recuerda el tiempo que le quedó en la anterior visita. Por tanto, la partida puede terminar sin éxito bien por muerte del jugador o porque se haya agotado el tiempo establecido.

Asimismo queremos que esta clase ofrezca la funcionalidad necesaria para guardar y recuperar partidas en un fichero. Nótese que esta funcionalidad se podría implementar de diferentes formas. El propósito es utilizar serialización, lo cual simplificará notablemente el proceso. Téngase en cuenta además que la serialización funciona por alcance, de modo que si alguno de los objetos no es serializable no funcionará correctamente. También es importante reseñar que es posible que el estado completo de todos los objetos no sea almacenado. Esto último puede ocurrir cuando hacemos uso de clases que no hemos implementado (por ejemplo, la clase Observable). Finalmente, hay que recordar que los enumerados debemos adaptarlos para que funcionen con la serialización (ver apartado 7.2).

## 5. Control de Errores

Existen numerosas situaciones en las que la aplicación puede finalizar si existe un error. Es responsabilidad del programador tratar estas situaciones de error e intentar alcanzar un estado consistente de la aplicación para seguir funcionando.

De acuerdo con la técnica del Diseño por Contrato, los errores se producirán como consecuencia de la ruptura del contrato entre la rutina cliente y la rutina a la que invoca (rutina servidora), esto es, por fallo en la precondición, postcondición o en el invariante. Normalmente, un fallo en la precondición se debe a un error del cliente el cual invoca de manera incorrecta al método: un argumento no válido, con un valor nulo,… En este caso, el error se notificará lanzando una ***excepción no comprobada o Runtime*** (para las situaciones anteriores se pueden utilizar las excepciones del lenguaje *IllegalArgumentException* y *NullPointerException* respectivamente). Por otro lado, un fallo en la postcondición se debe a que la rutina servidora, habiéndole pasado argumentos válidos (cumplen la precondición establecida) no puede terminar cumpliendo la postcondición. En ese caso, se debe informar a la rutina cliente que no se ha podido terminar la ejecución con éxito, bien devolviendo un valor especial, o bien

lanzando una ***excepción comprobada***. El optar por una u otra dependerá de cómo de importante es que el cliente no se olvide de manejar la situación de error y también de que en ocasiones no existe la posibilidad de devolver un valor especial, como es el caso de los constructores.

A continuación se relacionan algunas situaciones de error comunes que son notificadas con excepciones no comprobadas y que conviene tener en cuenta:

- Conversión de cadenas a enteros. Si la conversión no puede realizarse, el método no devuelve un valor especial, sino que lanza la excepción *NumberFormatException*.
- El método *nextToken* de un *StringTokenizer* lanza un error si no quedan elementos. Lo mismo sucede con el método *next* de los iteradores. La excepción que se produce es *NoSuchElementException*. Esta excepción se puede evitar comprobando si quedan elementos antes de aplicar las operaciones *next*.
- El método *substring* de las cadenas lanza la excepción *IndexOutOfBoundsException* si se invoca con índices fuera de rango. Esto puede suceder cuando se toma como valor de índice el resultado de una operación *indexOf* previa, que puede devolver un valor -1.

Otra de las excepciones Runtime predefinidas en el lenguaje Java es *IllegalStateException.* Esta excepción se utiliza para notificar que se está invocando un método en un momento que no es el adecuado, esto es, el objeto no se encuentra en el estado adecuado para atender a la operación que se ha demandado. Por ejemplo, en el caso de la clase Partida, no tiene sentido invocar al método parar si la partida no ha comenzado.

El método que procesa el fichero de configuración de un escenario es susceptible a errores de configuración. Este tipo de errores no dependen del programador, sino de los datos que proporciona el usuario externamente. Un fichero de configuración puede ser muy extenso y convendría saber el motivo por el que no se ha podido crear un escenario: error de E/S, dimensiones incorrectas, no hay dimensiones, etc. Por tanto, crearemos la excepción comprobada C**onfiguracionIncorrecta** que notifique con detalle el error producido.

Nótese que aunque no es obligatorio capturar las excepciones *Runtime*, en ocasiones puede ser necesario hacerlo para transformar la excepción no comprobada en otra que sea comprobada. Por ejemplo, capturar la excepción *NumberFormatException* para indicar que ha habido un error de configuración en el escenario.

En esta práctica hay que controlar las *situaciones de error de todo el programa* y trataremos la violación de las precondiciones de los métodos para las clases Escenario y Partida. En tal caso es conveniente utilizar las excepciones de *Runtime* del lenguaje.

Además de tener en cuenta las precondiciones en el código (modificando los métodos para que chequeen las precondiciones y lancen la excepción *runtime* apropiada), debemos documentarlas en *javadoc* para lo que debemos definir una *plantilla de documentación* en JBuilder (ver apartado 7.4).

Por ejemplo, el método `addEntidad` de `Escenario` quedaría como sigue:

```
/**
* Añade una entidad al escenario
* @param Entidad entidad
*
* <br><br><b>Precondiciones: </b><br>
* <ul>
* <li> La entidad no puede ser nula
* </ul>
*/
public boolean addEntidad (Entidad entidad) {
      if (entidad == null)
            throw new NullPointerException("La entidad es nula");
      …
}
```

## 6. Interfaz de usuario

Para el desarrollo de la interfaz del juego se ofrece una librería de clases gráficas que realizan la visualización. Junto a la librería se incluye la documentación javadoc para su uso. Es el propósito de la práctica aprender a usar la documentación del código para averiguar cómo utilizar la librería (ver apartado 7.3). Además, se ofrece un fichero con las imágenes de las entidades del juego para su visualización.

## 7. Otras cuestiones de implementación

A continuación presentamos información que puede servir de ayuda y debéis tener en cuenta en la realización de la práctica.

### 7.1 Reflexión

Se hará uso de la reflexión de Java cuando encontremos fragmentos de código que realicen un análisis de casos exhaustivo que no se puede evitar con la herencia, y que por tanto limitan el mantenimiento de la aplicación. Asimismo, deberá evitarse la declaración de métodos *static* "crear" para la instanciación de las entidades del juego.

## 7.2 Enumerados y serialización

A continuación se muestra la definición de un enumerado preparado para la serialización. Es obligatorio el uso de enumerados en las situaciones donde sea conveniente, evitando la simulación de enumerados por constantes dentro de las clases.

```
public class Direccion implements Serializable{
      private final transient String nombre;

      public static final Direccion DERECHA = new Direccion (“Derecha”);
      public static final Direccion IZQUIERDA = new Direccion (“Izquierda”);
      public static final Direccion ARRIBA = new Direccion (“Arriba”);
      public static final Direccion ABAJO = new Direccion (“Abajo”);

      private Direccion(String nombre){ this.nombre = nombre;}

      public String toString() {return nombre;}

      //Añadir para la serialización
      private static int nextCodigo = 0;
      private final int codigo = nextCodigo++;
      private static final Direccion [] VALORES =
            {DERECHA, IZQUIERDA, SALTA, DISPARA};
      private Object readResolve() throws ObjectStreamException{
            return VALORES[codigo];
      }
}
```

## 7.3 Uso de librerías en JBuilder

Para poder utilizar la librería que contiene las clases de la interfaz gráfica hay que especificar en las propiedades del proyecto donde se encuentra el fichero `vista.jar`

1) En el menú `Proyecto`, seleccionamos la opción `Propiedades de Proyecto`. En la ventana que se abre tenemos que seleccionar la pestaña `Bibliotecas necesarias`.

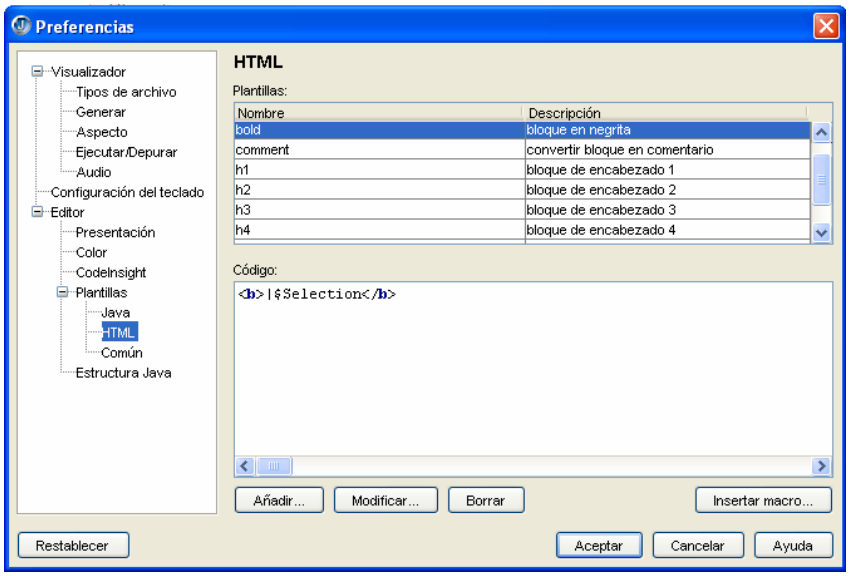
2) Dando al botón `Abrir` se abre una ventana para seleccionar la biblioteca. Debemos pulsar el botón `Nuevo` para especificar la ubicación de la biblioteca con la interfaz gráfica.

3) En el asistente que se abre debemos especificar el nombre que queremos darle a la biblioteca y la ubicación dándole al botón `Añadir`.

## 7.4 Definición de plantillas de documentación

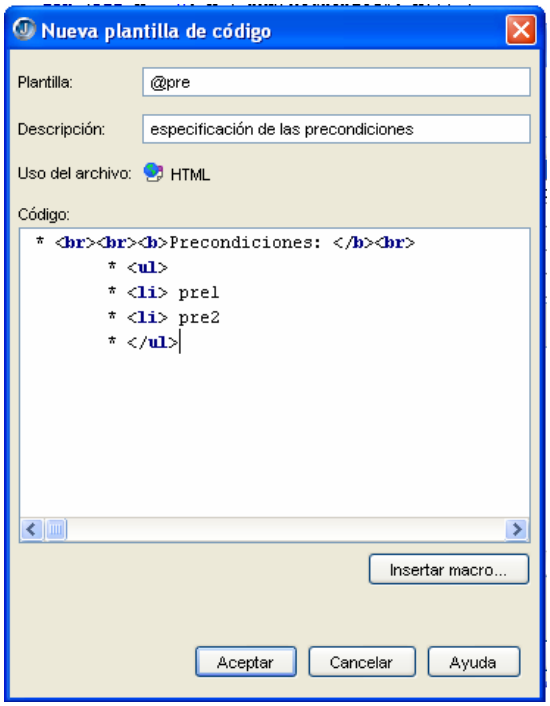
En *JBuilder* podemos definir una plantilla de documentación que nos permita documentar las precondiciones de los métodos implementados. Nótese que también pueden definirse plantillas de código que resultan muy útiles para la programación. A continuación se explica cómo definir una plantilla de documentación:

1) Acceder al menú `Herramientas` y dentro de este a la opción de menú `Preferencias`. Aparecerá la siguiente ventana:

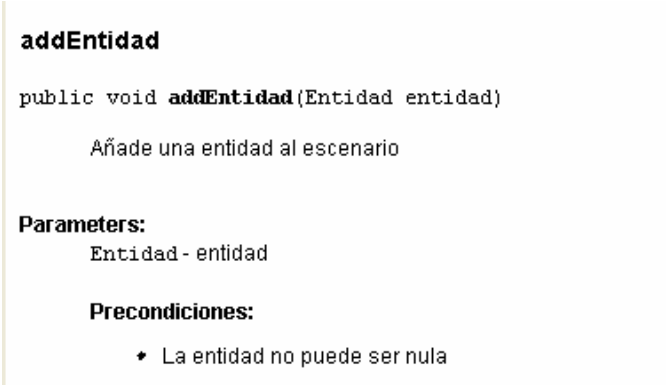



Seleccionar en el panel de la izquierda: `Editor > Plantillas > HTML`

2) Pulsar el botón Añadir … Rellenar la plantilla y pulsar Aceptar.




3) A partir de ahora cada vez que estemos dentro de un comentario de documentación y queramos hacer uso de la plantilla definida no hay más que pulsar ctrl.+J y seleccionar la plantilla @pre. La documentación generada con el *javadoc* quedaría como sigue:



**addEntidad**

`public void addEntidad(Entidad entidad)`

Añade una entidad al escenario

**Parameters:**

`Entidad` - entidad

**Precondiciones:**

- La entidad no puede ser nula

## 8. Entrega

El plazo de entrega de la práctica finaliza el ***viernes 16 de junio*** (para la convocatoria de junio) y el ***viernes 8 de septiembre*** (para la convocatoria de septiembre).

- Se debe generar un recopilatorio (fichero .jar) que ha de incluir el código de la aplicación. El recopilatorio debe ejecutarse con el comando: **java –jar videojuego.jar**. Por último, la generación del recopilatorio debe realizarse desde proyecto JBuilder, de manera que pueda actualizarse el recopilatorio cada vez que compila la aplicación.

- La práctica se entregará **en SUMA y** también una copia **en papel** del código fuente de la aplicación ordenando las clases de acuerdo a la jerarquía de herencia (no por orden alfabético).