Grado en Ingeniería Informática

Metodología de la Programación

Práctica Obligatoria 2 – Primera Convocatoria

Buscaminas 2.0

Docentes:
Raúl Marticorena
Félix Nogal

Universidad de Burgos

Índice de contenidos

1. Introducción

El objetivo fundamental es **continuar implementando** la variante **simplificada** del juego del **Buscaminas.** Partiendo de lo realizado en la práctica anterior, se incluyen ahora los **siguientes conceptos relacionados con lo visto en teoría:**

- Inclusión de una **jerarquía de herencia de árbitros,** con una interfaz, clase abstracta y clases concretas (se recomienda revisar el Tema 4. Herencia).
- **Inclusión y uso** de **interfaces y clases genéricas** (se recomienda revisar el Tema 5. Genericidad).
- Inclusión de **excepciones** comprobables y su lanzamiento con un enfoque defensivo con el consiguiente tratamiento de excepciones (**comprobables y no comprobables**) a incluir en el código (se recomienda revisar el Tema 6. Programación Defensiva y Tratamiento de Excepciones).

A continuación se recuerdan de manera general las reglas ya tenidas en cuenta en la anterior versión.

Es un juego abstracto de **tablero** de 8x8 celdas, para un **solo jugador**. En dicho tablero, se oculta inicialmente el contenido de sus celdas. Aleatoriamente se colocan **10 minas** ocultas, en el tablero. El jugador irá descubriendo el contenido de las celdas, intentando no explotar una mina. Al descubrir una celda, puede ocurrir que la celda esté vacía o bien la celda contenga una mina. En este último caso, se considera que la mina **explota** y se **pierde** la partida.

Al descubrir una celda vacía, se descubren **automáticamente** las celdas vacías contiguas. Ese proceso automático de descubrir celdas vacías, se detiene al llegar a una celda que es a su vez contigua a una o varias minas, o bien alcanza los límites del tablero. Esas celdas descubiertas que son contiguas a una o varias minas, muestran el número de minas adyacentes. Utilizando esos números como **pista o ayuda**, el jugador puede marcar con una **bandera** las celdas que son sospechosas de ser minas, para no descubrirlas por error.

Esta dinámica de poner **banderas** en celdas sospechosas y descubrir nuevas celdas, se repite hasta que el jugador descubre todas las celdas sin minas, ganando la partida (ver Ilustración 1), o bien perdiendo al descubrir accidentalmente alguna mina.

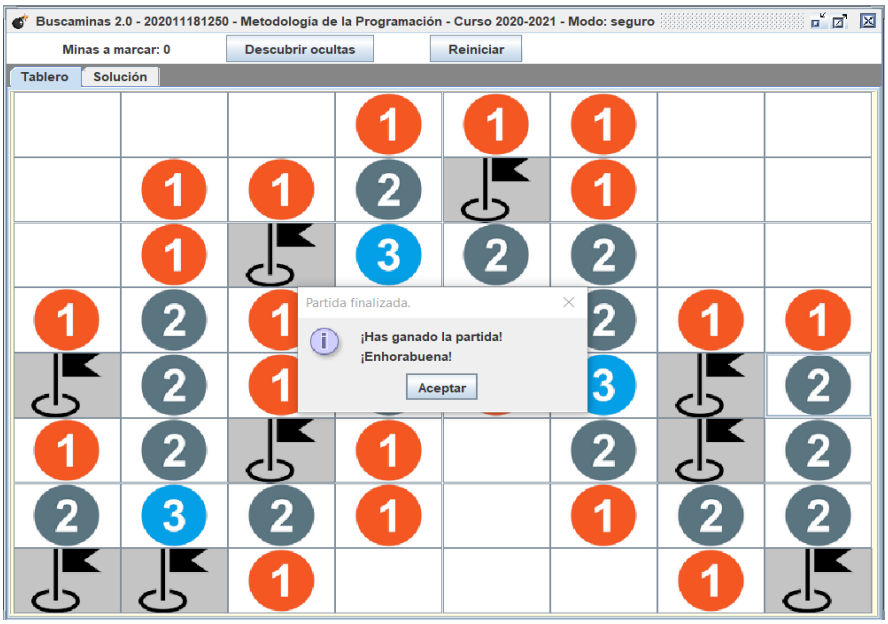

Ilustración 1: Estado final de la partida con victoria

En esta nueva version, **podremos jugar en dos modos:**

- **Seguro**: se garantiza que la primera celda elegida NO contiene nunca una mina, colocando las minas en relación a dicha celda inicial, y siempre con una semilla preestablecida a la hora de generar las posiciones aleatorias de las minas.
- **Inseguro**: NO se garantiza que en la primera celda elegida NO contenga una mina. Las minas se colocan de manera aleatoria, sin una semilla inicial establecida. No es predecible la situación inicial de la partida.

Para desarrollar este juego se establece la siguiente estructura de paquetes y módulos/clases (ver Tabla 1):

Paquete	Módulos / Clases	Nº	Descripción

`juego.modelo`	`Celda`
	`Estado`
	`Tablero`
		`2 clases`
		`1 enumeración`
			Modelo fundamental.

`juego.control`	***`Arbitro`***
	`ArbitroAbstracto`
	`ArbitroSeguro`
	`ArbitroInseguro`
	`DistanciaChebyshev`
		`1 interfaz`
		`1 clase abstracta`
		`3 clases concretas`
			Lógica de negocio (reglas del juego).
			`Se proporciona la interfaz que NO debe ser modificada.`

`juego.util`	`CoordenadasIncorrectasException` **`Sentido`**	`1 clase (excepción)` `1 enumeración`	Utilidad.
`juego.textui`	**`Buscaminas`**	`1 clase`	Interfaz de usuario en modo texto. Se proporciona **parcialmente** el código a completar por los alumnos.
`juego.gui`	`Sin determinar`	`Sin determinar`	Interfaz gráfica (se proporcionan los binarios en formato `.jar`)
`juego.gui.images`	`Sin determinar`	`Sin determinar`	Imágenes en la aplicación (se proporcionan junto con el fichero `.jar`)

Tabla 1: Resumen de paquetes y módulos

La interfaz gráfica se ha subcontratado a la empresa ECMA (filas en color verde en Tabla 1), siguiendo los diagramas e indicaciones que se detallan en el presente enunciado y se proporcionan los ficheros binarios en formato `.jar` en la plataforma UBUVirtual.

Es labor de los alumnos implementar los **ficheros fuente** `.java` restantes necesarios para el correcto cierre y ensamblaje del sistema, tanto en modo texto y gráfico, junto con el resto de productos indicados en el apartado 3 Descripción.

Para ello se deben seguir las **indicaciones dadas** y los **diagramas disponibles**, respetando los diseños y signaturas de los métodos, con sus correspondientes modificadores de acceso, quedando **a decisión del alumno la inclusión de atributos y/o métodos privados (`private`), `protected` o amigables, siempre de manera justificada. NO se pueden añadir atributos ni métodos amigables o públicos adicionales.**

2. Objetivos

- Construir siguiendo los diagramas e indicaciones dadas la implementación del juego en Java.
  - Completando una aplicación en modo texto.
  - Completando una aplicación en modo gráfico.
- Incluir conceptos avanzados de **herencia**, **genericidad** y **excepciones** en esta nueva versión.
- Generar la documentación correspondiente al código en formato HTML.
- Comprobar la completitud de la documentación generada previamente.
- Aportar los *scripts* correspondientes para realizar el proceso completo de compilación, documentación y ejecución (tanto en modo texto como gráfico).

3. Descripción

A continuación se desglosan los distintos diagramas y comentarios, a tener en cuenta de cara a la implementación de la práctica. Algunos métodos de consulta (Ej: `obtener...`, `consultar...`, etc.) y asignación (Ej: `establecer...`, `colocar...`, etc.) que no conllevan ningún proceso adicional, salvo la lectura o escritura de atributos, no se comentan por motivos de brevedad.

3.1 Paquete `juego.modelo`

El paquete está formado por dos clases y una enumeración. En la Ilustración 2 se muestra el diagrama de clases correspondiente.

Comentarios respecto a `Celda`:

- Inicialmente toda celda estará vacía y solo contiene las coordenadas con su posición en el tablero. Su estado inicial es siempre `OCULTA`.
- El método `clonar` crea un clon independiente, con el mismo estado de la celda clonada.
- El método `colocarMina` coloca una mina en la celda.
- El método `equals`[1] compara la igualdad de valores entre dos celdas.
  - Este método se generará automáticamente desde Eclipse: hacemos click con el botón derecho sobre el código fuente de la clase `Celda` y seleccionamos *Source/Generate hashCode() and equals()…* seleccionando **todos** los atributos de la clase. Si se cambian los atributos de la clase, debe volver a generarse. Este método se utilizará en los tests automáticos.
- El método `establecerSiguienteEstadoDescubrir`, cambia el estado de la celda al descubrirla. Dependiendo de su estado previo:
  - Si estaba `OCULTA` pasará a estar `DESCUBIERTA`.
  - Si estaba `MARCADA` o `DESCUBIERTA` no se hace nada.
- El método `establecerSiguienteEstadoMarcar`, cambia el estado de la celda, al marcarla con una bandera o desmarcarla quitando la bandera. Dependiendo de su estado previo:
  - Si estaba `OCULTA` pasará a estar `MARCADA` (se pone la bandera).

1 Para más información sobre los métodos `hashCode` y `equals`, consultar la documentación en línea de la clase `Object`.

- Si estaba `MARCADA` pasará a estar `OCULTA` (se quita la bandera).
- Si estaba `DESCUBIERTA` no se hace nada.

- El método `hashCode` devuelve el código *hash* del objeto, en función de sus atributos.
  - Este método se generará automáticamente desde Eclipse, siguiendo las mismas indicaciones previas que con el método `equals`.
- El método `obtenerNumeroMinasAdyacentes`, obtiene el numéro de minas adyacentes a la celda actual, teniendo en cuenta los ochos sentidos indicados en la enumeracion `Sentido`.
- El método `obtenerTextoEstado`, devuelve el estado de la celda en formato texto.
  - Si la ceda está descubierta:
    - Si contiene una mina devolverá `" M "`.
    - Si es una celda vacía adyacente a minas, devuelve `" X "` donde `X` será un número entre [1,8].
    - Si es una celda vacía no adyacente a minas, devuelve `" . "` como texto.
  - Si no está descubierta:
    - Devuelve `" E "` donde se reemplazará `E` por la letra correspondiente al estado de la celda.
  - Nota: siempre se añade un espacio en blanco a izquierda y derecha en dicha cadena retornada.
- El método `obtenerTextoSolucion`, devuelve el texto solución del estado actual de la celda.
  - Si tiene una mina, devuelve `" M "`.
  - Si no:
    - Si es una celda vacía adyacente a minas, devuelve `" X "` donde `X` será un número entre [1,8].
    - Si es una celda vacía no adyacente a minas, devuelve `" - "` como texto.
- El método `tieneCoordenadasIguales` devuelve `true` si la celda pasada como argumento tiene iguales coordenadas, con independencia del resto de su estado, `false` en caso contrario.
- El método `toString` devuelve en formato texto el estado actual de la celda, con sus coordenadas, número de minas adyacentes y estado. Con el siguiente formato `"[(X,Y)-Z-ESTADO]"`. Por ejemplo: `"[(1,2)-0-OCULTA]"` para la celda (1,2) con cero minas adyacentes, que está oculta.`A lo largo del tiempo una celda podrá estar vacía o contener`

Comentarios respecto a `Estado`:

- Tipo enumerado que contiene valores `DESCUBIERTA`, `MARCADA` y `OCULTA`.
- Permite almacenar y consultar el correspondiente carácter asociado a cada estado: `'0'` (cero), `'P'` y `'-'` respectivamente.

Comentarios respecto a `Tablero`:

- Un tablero se considera como un conjunto de celdas, cada una en una posición (fila,columna). Suponiendo que el tablero es de 8 filas x 8 columnas, entonces tenemos: (0,0) las coordenadas de la esquina superior izquierda, (0,7) las coordenadas de la esquina superior derecha, (7,0) las coordenadas de la esquina inferior izquierda y (7,7) las coordenadas de la esquina inferior derecha. Se numera de izquierda a derecha y en sentido descendente.
- El conjunto de celdas de un tablero debe **implementarse ahora** utilizando la **versiones genéricas de la interfaz `java.util.List<E>` y la clase concreta `java.util.ArrayList<E>`** (en **ningún caso** se utilizarán en esta práctica un *array* de celdas de dos dimensiones). Es

conveniente acordarse que una tipo genérico se puede definir en función de si mismo como se ha visto en el Tema 5 (Ej: `Pila<Pila<Circulo>>`). Al instanciar un tablero se crean y asignan las correspondientes celdas vacías, con sus correspondientes coordenadas.

- El método `clonar` crea un clon independiente, con el mismo estado del tablero original.
- El método `clonarCelda` obtiene un clon de la celda en las coordenadas correspondientes. Si las coordenadas no pertenecen al tablero se **lanza la excepción** `CoordenadasIncorrectasException`.
- El método `clonarCeldas`, devuelve una **lista de celdas** con el conjunto de clones de las celdas del tablero, en orden consecutivo, recorriendo en orden filas y columnas, de izquierda a derecha y en sentido descendente.
- El método `colocarMinas()` coloca aleatoriamente 10 minas en el tablero.
- El método `colocarMinas(Celda, Distancia, Random)`, coloca aleatoriamente las 10 minas en el tablero. Las coordenadas aleatorias se obtienen iterativamente, a partir del método `nextInt` del objeto `java.util.Random` pasado como argumento, obteniendo primero la fila y luego la columna (en el rango [0,7]). Con la fila y columna generada, hay que comprobar dichas coordenadas, dado que nunca se puede colocar una mina en la celda inicial descubierta, ni adyacente a ella (Nota: utilizar la distancia de Chebyshev para asegurar esto) y tampoco se puede colocar una mina en una celda que ya se ha colocado una mina previamente. En tales casos, se ignoran las coordenadas generadas y se pasa a obtener otras nuevas. Es **MUY IMPORTANTE** cumplir todas las condiciones dadas, para asegurarnos la **correcta inicialización** del tablero con sus minas y posterior uso de los tests automáticos.
- El método `contarMinasExplotadas` devuelve el número de celdas descubiertas que contienen minas.
- El método `descubrir`, descubre la celda con las coordenadas dadas, junto con todas sus celdas adyacentes que estén vacías, deteniéndose en aquellas celdas adyacentes a minas o en los límites del tablero. Si las coordenadas no pertenecen al tablero se **lanza la excepción** `CoordenadasIncorrectasException`.
- El método `marcarDesmarcar`, cambia el estado de la celda de marcada a desmarcada, o viceversa, según el estado de la celda. Si las coordenadas no pertenecen al tablero se **lanza la excepción** `CoordenadasIncorrectasException`.
- El método `obtenerCelda`, devuelve la referencia a la celda del tablero. Si las coordenadas no pertenecen al tablero se **lanza la excepción** `CoordenadasIncorrectasException`. Nota: su modificador de acceso es amigable.
- El método `obtenerSolucion` devuelve el estado del tablero en formato cadena, mostrando la solución de la partida, al visualizar las minas ocultas, las celdas vacías y el conteo de minas adyacentes. Ej: tras descubrir la celda inicial.

```
0       -  -  -  -  -  -  1  M
1       -  -  -  -  -  -  1  1
2       1  1  -  -  -  1  1  1
3       M  1  -  -  -  1  M  1
4       2  2  -  1  2  3  2  1
5       M  1  -  1  M  M  2  1
6       1  1  -  2  4  5  M  2
7       -  -  -  1  M  M  3  M

        0  1  2  3  4  5  6  7
```


- El método `toString` devuelve el estado actual del tablero en formato cadena tal y como se mostraría a un jugador en plena partida. Ej: tras descubrir la celda inicial y adyacentes (se utiliza un " . " para dichas celdas), con las celdas descubiertas adyacentes a minas, indicando el número de las mismas (Ej: 1, 2 o 3), con tres celdas marcadas con bandera con " P " y con el resto de celdas ocultas con " - ". Si por un casual descubrimos una celda con mina, aparecerá " M ".

```
0       .  .  .  .  .  .  1  -
1       .  .  .  .  .  .  1  -
2       1  1  .  .  .  1  1  -
3       P  1  .  .  .  1  -  -
4       -  2  .  1  2  3  -  -
5       P  1  .  1  P  -  -  -
6       1  1  .  2  -  -  -  -
7       .  .  .  1  -  -  -  -

        0  1  2  3  4  5  6  7
```

-

A continuación se muestra el diagrama de clases completo para el paquete `juego.modelo`.

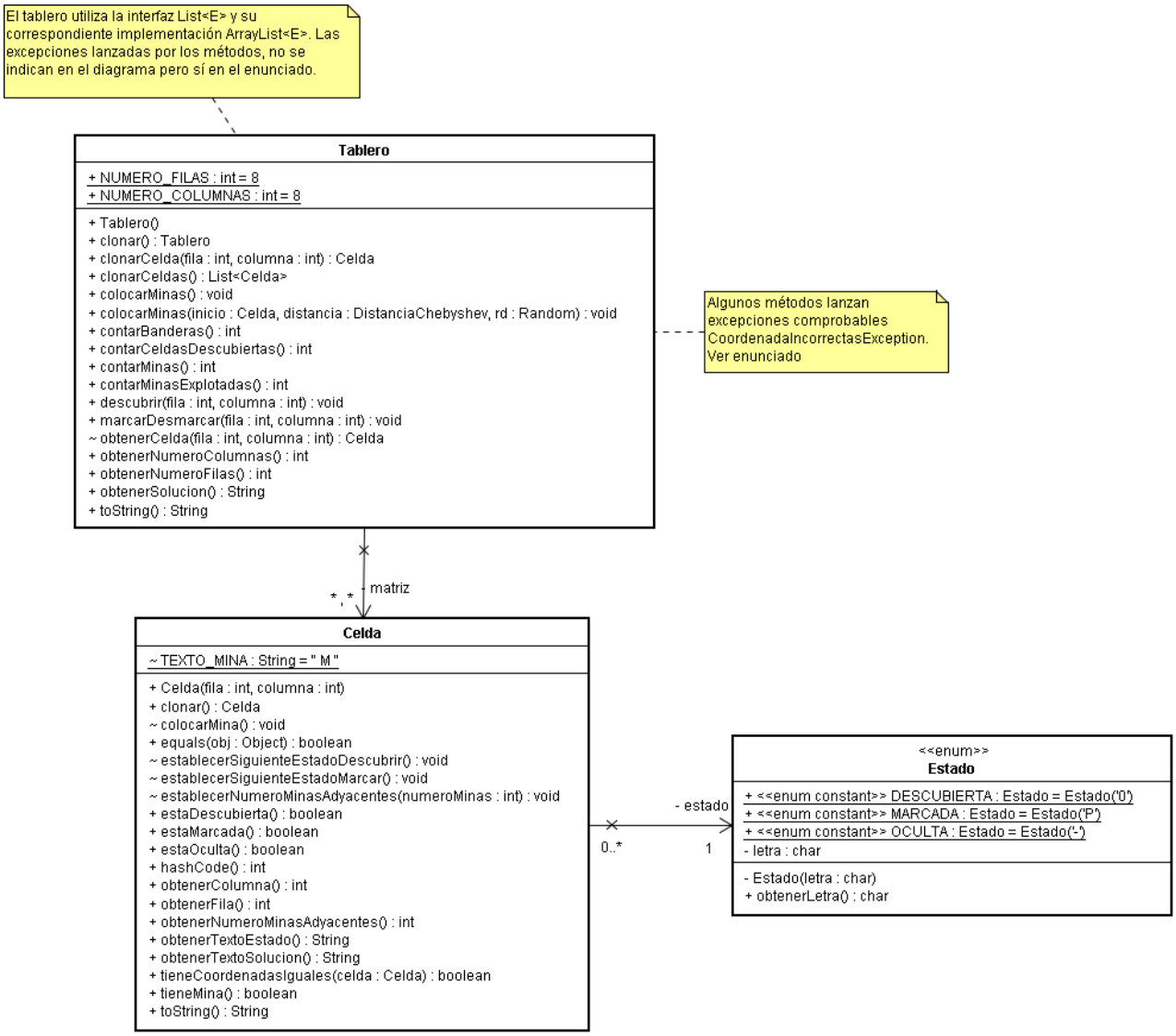

Ilustración 2: Diagrama de clases de `juego.modelo`

Dada la complejidad de algún método en la clase `Celda` y en particular en `Tablero`, y en previsión de generar código duplicado, se recomienda **dividir el código de métodos largos, en métodos privados más pequeños y reutilizarlos, siempre que sea posible.**

3.2 Paquete `juego.control`

El paquete contiene una interfaz, una clase abstracta y tres clases concretas. Define la lógica de negocio o reglas del juego a implementar (ver Ilustración 3). Varios de los métodos lanzan excepciones comprobables (revisar el enunciado y códigos proporcionados).

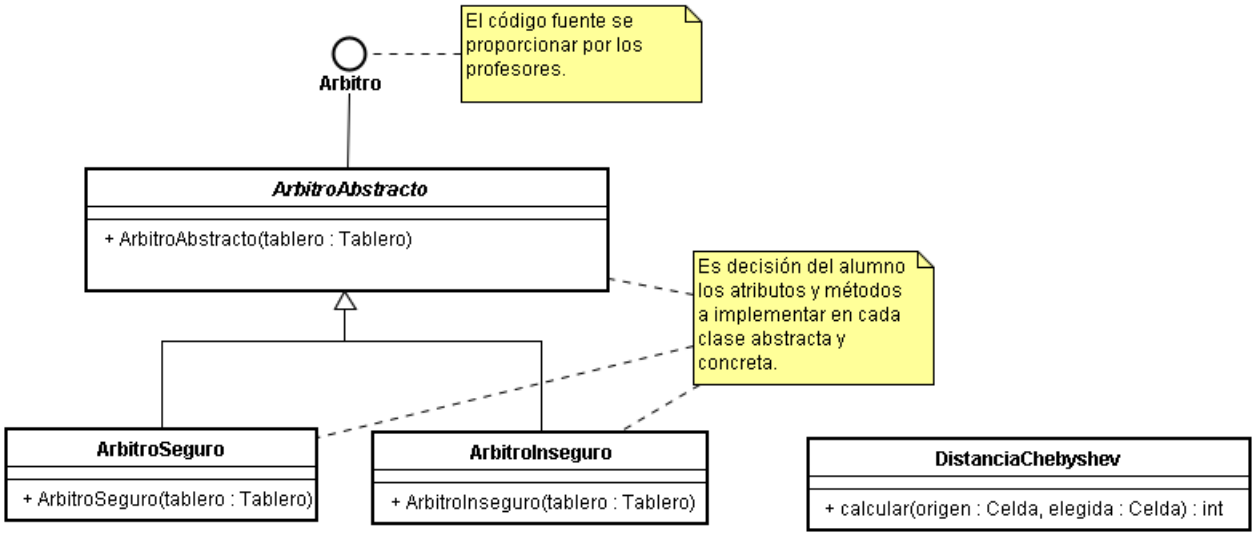

Ilustración 3: Diagrama de clases de `juego.control`

Comentarios respecto a la interfaz Arbitro:

- El código fuente con los comentarios de la funcionalidad esperada de cada método, con sus excepciones comprobables a lanzar, **se proporciona por los profesores.**
- **NO se puede modificar dicho fichero.**

Comentarios respecto a la clase abstracta `ArbitroAbstracto`:

- Implementa la interfaz `Arbitro`.
- El constructor asigna el tablero actual sobre el que se desarrolla la partida.
- Es responsabilidad del alumno dedidir qué atributos y métodos concretos se implementan en dicha clase abstracta, en relación con el resto de la jerarquía de herencia.

Comentarios respecto a la clase concreta `ArbitroSeguro`:

- Hereda de `ArbitroAbstracto`.
- El constructor asigna el tablero actual sobre el que se desarrolla la partida.
- Este árbitro tiene que asegurar que **nunca** se puede explotar una mina en el **primer movimiento** y que las minas se han colocado de manera **pseudoaleatoria**, igual que en la primera práctica.
- Es responsabilidad del alumno dedidir qué atributos y métodos concretos se implementan en dicha clase concreta, en relación con el resto de la jerarquía de herencia.

Comentarios respecto a la clase concreta ArbitroInseguro:

- Hereda de `ArbitroAbstracto`.
- El constructor asigna el tablero actual sobre el que se desarrolla la partida.
- Este arbitro **NO asegura** que no podamos explotar una mina en el **primer movimiento**. Pudiendo perder la partida en la primera tirada. Las minas se colocan de manera totalmente **aleatoria** sin ninguna semilla prefijada.
- Es responsabilidad del alumno dedidir qué atributos y métodos concretos se implementan en dicha clase concreta, en relación con el resto de la jerarquía de herencia.

Comentarios respecto a la clase `DistanciaChebyshev`:

- Se trata de una clase meramente funcional (no tiene estado).[2]
- El método `calcular` realizar el cálculo de la distancia de Chebyshev entre dos puntos (en nuestro caso dos celdas con sus coordenadas fila y columna). Se calcula como el máximo del valor absoluto de las diferencias a lo largo de cualquiera de sus dimensiones, siguiendo la siguiente formula:

$$D_{\text{Chebyshev}}(x, y) := \max_i(|x_i - y_i|).$$

3.3 Paquete `juego.util`

Contiene una una clase excepción y una enumeración, como se puede ver en Ilustración 4.

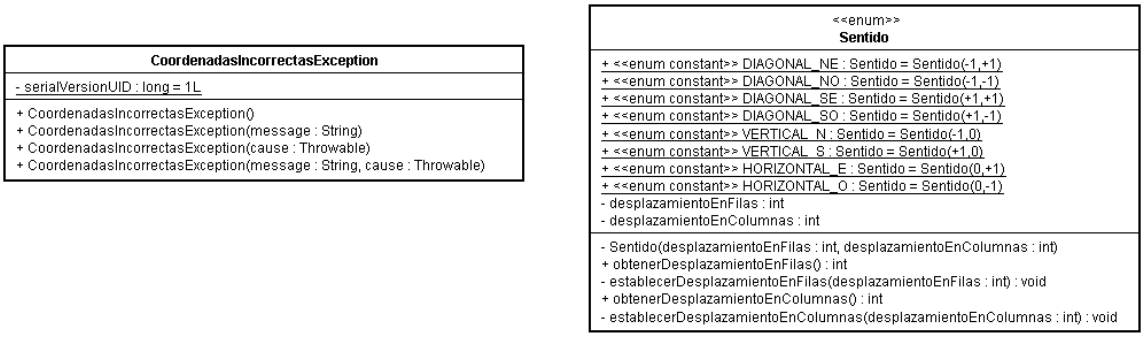

Ilustración 4: Diagrama de clases `juego.util`

Comentarios respecto a `CoordenadasIncorrectasException`:

- Clase excepción **comprobable** que hereda **obligatoriamente** de `java.lang.Exception`.
- Incluye cuatro constructores habituales en una excepción (ver Ilustración 4).
- **Básicamente indica que las coordenadas utilizadas (fila/columna) para una celda, no se encuentran dentro de los límites de las dimensiones del tablero de nuestro juego.**

Comentarios respecto a `Sentido`:

- La enumeración `Sentido` contiene los ocho **sentidos** que se tienen en cuenta para considerar las celdas en diagonal, horizontal o vertical a las que se podrá mover en los movimientos típicos de las piezas (a excepción del `Caballo`), junto con su desplazamiento en filas y columnas correspondiente a cada caso. Se recuerda el uso del método `values` para obtener un *array* con todos los valores definidos en el tipo enumerado para simplificar el código. Es muy importante utilizar esta enumeración con los correspondientes valores de desplazamiento para simplificar el código.
- **Este fichero ya resuelto se proporciona en UBUVirtual.**

2 En el último tema teórico de la asignatura, se verá como se resuelven estas clases con programación funcional, mediante el uso de expresiones lambda. desde la versión 8 de Java.

3.4 Paquete `juego.textui`

En este paquete se implementa, la interfaz en modo texto, que reutiliza las paquetes anteriores. Las clase raíz del sistema es `juego.textui.Buscaminas`. Adicionalmente se puede jugar añadiendo dos argumentos **opcionales en línea de comandos:** un primer argumento con valores `"seguro"` o `"inseguro"` y luego `"trampas"` (ignorando mayúsculas y minúsculas).

Se puede jugar **eligiendo uno u otro arbitro** y adicionalmente jugar con trampas. Por ejemplo, se puede elegir jugar con las siguientes opciones:

- `seguro trampas`
- `inseguro trampas`
- `seguro`
- `inseguro`
- `trampas`

Por defecto, si no se indica alguna de las opciones (o ninguna) se juega en modo seguro y sin trampas.

Un ejemplo de sintaxis de invocación (sin detallar cómo debe configurarse el `classpath`) jugando en modo seguro sin trampas, es:

```
$> java juego.textui.Buscaminas seguro
```

La salida en pantalla debe ser similar a la siguiente.

```
Bienvenido al juego del Buscaminas
Jugando en modo seguro
Jugando sin trampas
Para finalizar el juego introduzca "salir".
Disfrute de la partida...


0        -  -  -  -  -  -  -  -
1        -  -  -  -  -  -  -  -
2        -  -  -  -  -  -  -  -
3        -  -  -  -  -  -  -  -
4        -  -  -  -  -  -  -  -
5        -  -  -  -  -  -  -  -
6        -  -  -  -  -  -  -  -
7        -  -  -  -  -  -  -  -

         0  1  2  3  4  5  6  7

Introduce jugada: (máscara fc / fc(M|m)):
```

Suponiendo que en la primera jugada se quiere descubir la celda (0,0):

```
Introduce jugada: (máscara fc / fc(M|m)): 00

0        .  .  .  1  -  -  -  -
1        .  1  1  2  -  -  -  -
2        .  1  -  -  -  -  -  -
3        1  2  -  -  -  -  -  -
4        -  -  -  -  -  -  -  -
5        -  -  -  -  -  -  -  -
6        -  -  -  -  -  -  -  -
7        -  -  -  -  -  -  -  -

         0  1  2  3  4  5  6  7
```


Si queremos marcar y poner una bandera en la celda (0,4) añadimos una letra m (en mayúsculas o minúsculas indistintamente):

```
Introduce jugada: (máscara fc / fc(M|m)): 04M

0         .  .  .  1  P  -  -  -
1         .  1  1  2  -  -  -  -
2         .  1  -  -  -  -  -  -
3         1  2  -  -  -  -  -  -
4         -  -  -  -  -  -  -  -
5         -  -  -  -  -  -  -  -
6         -  -  -  -  -  -  -  -
7         -  -  -  -  -  -  -  -

          0  1  2  3  4  5  6  7
```

Si descubrimos una mina en la celda (1,4) finaliza la partida, mostrando el estado actual, el mesaje de derrota y el tablero con la solución:

```
Introduce jugada: (máscara fc / fc(M|m)): 14

0         .  .  .  1  P  -  -  -
1         .  1  1  2  M  -  -  -
2         .  1  -  -  -  -  -  -
3         1  2  -  -  -  -  -  -
4         -  -  -  -  -  -  -  -
5         -  -  -  -  -  -  -  -
6         -  -  -  -  -  -  -  -
7         -  -  -  -  -  -  -  -

          0  1  2  3  4  5  6  7

Lo siento. ¡Has perdido la partida!

0         -  -  -  1  1  1  -  -
1         -  1  1  2  M  1  -  -
2         -  1  M  3  2  2  -  -
3         1  2  1  2  M  2  1  1
4         M  2  1  2  1  3  M  2
5         1  2  M  1  -  2  M  2
6         2  3  2  1  -  1  2  2
7         M  M  1  -  -  -  1  M

          0  1  2  3  4  5  6  7

Partida finalizada.
```

Si la jugada introducida **no es legal,** se debe informar del error al usuario, solicitando de nuevo que introduzca la jugada y sin saltar el turno. No hay límite en el número de reintentos.

Si en algún momento el usuario introduce `"descubrir"` (ignorando mayúsculas o minúsculas) se descubren todas las celdas ocultas, no marcadas con bandera, perdiendo la partida si alguna contenía una mina,o ganando en caso contrario.

En caso de victoria, se mostrará siempre el estado del tablero y el mensaje `"Enhorabuena! Has ganado la partida."`.

Si en algún momento el usuario introduce `"salir"` se interrumpe la partida y se finaliza simplemente mostrando en pantalla: `Partida finalizada.`

Este fichero se proporciona parcialmente resuelto en UBUVirtual. Solo hay que completar el método `main`, reutilizando el resto de métodos proporcionados, sin modificar el resto.

3.5 Paquete `juego.gui/juego.gui.images`

Estos paquetes implementan la interfaz gráfica del juego y se proporciona ya resuelta por los profesores. La clase raíz del sistema es `juego.gui.Buscaminas`.

Adicionalmente se puede pasar un argumento en línea de comandos (`"seguro"` o `"inseguro"`) **para jugar con uno u otro árbitro**. Si no se indica nada, el valor por defecto es `"seguro"`.

Ejemplo de invocación (sin detallar cómo debe configurarse el `classpath`, quedando como ejercicio) :

```
$> java juego.gui.Buscaminas inseguro
```

La interfaz inicial será similar a la mostrada en la Ilustración 5. Para descubrir una celda, se utiliza el botón izquierdo del ratón y para marcar/desmarcar colocando una bandera en la celda, con el botón derecho. En la parte superior se muestar el contador de minas pendientes de marcar con bandera.

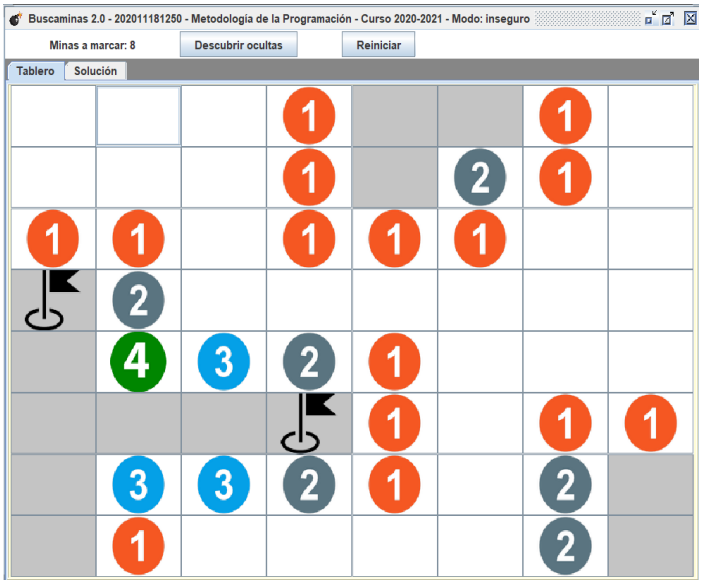

Ilustración 5: Interfaz gráfico tras descubrir la celda inicial y marcar dos banderas

Al igual que en el modo texto, se puede jugar haciendo trampas (ver Ilustración 6), puesto que en la pestaña solución, se mostrará la solución en formato texto, por si se requiere ayuda para resolver la partida.

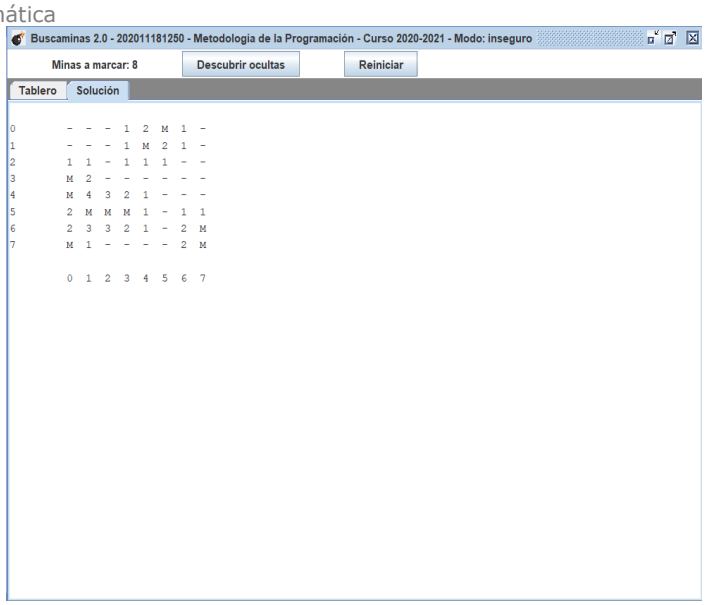

Ilustración 6: Solución texto mostrada en la pestaña

El botón ***Descubrir ocultas***, se utiliza para agilizar la finalización de la partida (por ejemplo, cuando ya hemos marcado las 10 banderas y queremos resolver), descubriendo TODAS las celdas que estén ocultas y no estén marcadas.

El botón ***Reiniciar***, reinicia la partida en cualquier momento que lo solicite el usuario, volviendo al estado inicial.

Las clases correspondientes a estos paquetes **se proporcionan en formato binario en un fichero con nombre** `buscaminas-gui-lib-2.0.0.jar` y se debe usar con los paquetes construidos previamente configurando el `classpath` y **sin descomprimir en ningún caso el fichero .jar**. No se utilizarán en estas prácticas módulos Java incluidos desde Java 9 (**no debemos tener ningún ficheros** `module-info.java`). Se recuerda que para evitar errores en la ejecución gráfica **la versión de Java a utilizar debe ser la 11**.

4. Entrega de la práctica

4.1 Fecha límite de entrega

- Ver fecha indicada en UBUVirtual de la tarea **[MP] Entrega de Práctica Obligatoria 2 - EPO2-1C.**

4.2 Formato de entrega

- Se enviará un fichero `.zip` o `.tar.gz` a través de la plataforma **UBUVirtual** completando la tarea **[MP] Entrega de Práctica Obligatoria 2 – EPO2-1C.**
- Los ficheros fuente estarán codificados **OBLIGATORIAMENTE** en formato **UTF-8** (Tip: comprobar en Eclipse, en *File/Properties* del proyecto que el valor *Text file encoding* está configurado a dicho valor).
- **TODOS** los ficheros fuente `.java` deben incluir los **nombres y apellidos de los autores** en su cabecera y deben estar **correctamente indentados. En caso contrario la calificación es cero.**

- El fichero comprimido seguirá el siguiente formato de nombre **sin utilizar tildes**:
  - **`Nombre PrimerApellido-Nombre PrimerApellido.zip` o**
  - **`Nombre PrimerApellido-Nombre PrimerApellido.tar.gz`**
- Ej: si los alumnos Félix Nogal y Raúl Marticorena, su fichero `.zip` se llamará sin utilizar acentos **`Felix Nogal-Raul Marticorena.zip`.**
- **Se puede realizar la práctica individualmente o por parejas. Se calificará con los mismos criterios en ambos casos. Si se hace por parejas, la nota de la práctica es la misma para ambos miembros.**
- **Aunque la práctica se haga por parejas, se enviará individualmente por parte de cada uno de los dos integrantes a través de UBUVirtual. Verificar que ambas entregas son iguales en contenido. En caso de NO coincidencia, se penalizará un 25% a ambos.**
- **NO se admiten envíos posteriores a la fecha y hora límite, ni a través de otro medio que no sea la entrega de la tarea en UBUVirtual. Si no se respetan las anteriores normas de envío la calificación es directamente cero.**
- **Cualquier situación de plagio detectado en las prácticas, conlleva la aplicación del reglamento de exámenes.**

Se debe crear la siguiente estructura de directorios y ficheros en el **disco y entregar un único fichero con dicha estructura comprimida. Es obligatorio entregar un único fichero**:

- `/leeme.txt`: fichero de texto, que contendrá los nombres y apellidos de los integrantes y las aclaraciones que los alumnos crean oportunas poner en conocimiento de los profesores.
- 🗀 `lib`: contiene las siguientes bibliotecas (descargar desde UBUVirtual)
  - `doccheck.jar` para poder generar el chequeo de la documentación
  - `buscaminas-gui-lib-2.0.0.jar`[*]: biblioteca con la interfaz gráfica **proporcionada por los profesores** para poder ejecutar la aplicación en modo gráfico.
  - `junit-platform-console-standalone-1.6.2.jar`: biblioteca utilizada para la ejecución de los tests con JUnit-5.
- 🗀 `src`: ficheros fuentes (`.java`) y ficheros de datos necesarios para poder compilar el producto completo.
- 🗀 `test`: ficheros fuentes (`.java`) con los tests automáticos **proporcionados por los profesores**.
- 🗀 `bin`: ficheros binarios (`.class`) y ficheros de datos necesarios.
- 🗀 `doc`: documentación HTML generada con **`javadoc`** de todos los ficheros fuentes.
- 🗀 `doccheck`: documentación HTML generada con el doclet **`DocCheck`** a partir de todos los ficheros fuentes.
- `/compilar.bat` o `/compilar.sh`: fichero de comandos con la invocación al compilador `javac` para generar el contenido de la carpeta `bin` a partir de los ficheros fuente en la carpeta `src`.
- `/documentar.bat` o `/documentar.sh`: fichero de comandos con la invocación al generador de documentación `javadoc` para generar el contenido del directorio `doc` a partir de los ficheros fuente en la carpeta `src`.
- `/chequear.bat` o `/chequear.sh`: fichero de comandos con la invocación al chequeo de documentación para generar el contenido del directorio `doccheck` a partir de los ficheros fuente en la carpeta `src` y del fichero `doccheck.jar` en `lib`.
- `/ejecutar_textui.bat` o `/ejecutar_textui.sh`: fichero de comandos con la invocación a la máquina virtual java para ejecutar la clase raíz del sistema en modo texto con cualquiera de las configuraciones descritas en el presente enunciado, utilizando las clases en el directorio `bin` y las bibliotecas necesarias en el directorio `lib`.
- `/ejecutar_gui.bat` o `/ejecutar_gui.sh`: fichero de comandos con la invocación a la máquina virtual `java` para ejecutar la clase raíz del sistema en modo gráfico, con cualquiera de las

* Si se publica alguna corrección, el fichero cambiará en número de versión

configuraciones descritas en el presente enunciado utilizando las clases en el directorio `bin` y las bibliotecas necesarias en el directorio `lib`.

Los *scripts* se pueden entregar bien para Windows (`.bat`) o bien para GNU/Linux o Mac OS X (`.sh`), pero se debe elegir **solo una plataforma**. Nota: tener en cuenta que en algunos *scripts* es necesario crear la carpeta de destino al principio. Ver los ejemplos de scripts proporcionados en sesiones previas de prácticas. **Probar previamente siempre antes de la entrega, el correcto despliegue del producto entregado.**

4.3 Comentarios adicionales

- **No se deben modificar los ficheros binarios ni los tests proporcionados.** En caso de ser necesario, por errores en el diseño/ mplementación de los mismos, se notificará para que sea el responsable de la asignatura quien publique en UBUVirtual la corrección y/o modificación. Se modificará el número de versión del fichero en correspondencia con la fecha de modificación y se publicará un listado de erratas.
- **Se ruega la utilización de los tests automáticos por parte de los alumnos** para verificar la corrección de la entrega realizada.
- **Se realizará un cuestionario invidual para probar la autoría de la misma.** El peso es de 5% sobre la nota final con nota de corte 4 sobre 10 para superar la asignatura.

4.4 Criterios de valoración

- La práctica es obligatoria, entendiendo que su no presentación en la fecha marcada, supone una calificación de cero sobre el total de 10 que se valora la práctica. La nota de corte en esta práctica es de 5 sobre 10 para poder superar la asignatura. Se recuerda que la práctica tiene un peso del **15% de la nota final de la asignatura**.
- Se valorará **negativamente** métodos con un **número de líneas grande** (>30 líneas) sin contar comentarios ni líneas en blanco, ni llaves de apertura o cierre, indentando el código con el formateo por defecto de Eclipse. En tales casos, se debe dividir el método en métodos privados más pequeños. Siempre se debe evitar en la medida de lo posible la repetición de código.
- No se admiten ficheros cuya estructura y contenido no se adapte a lo indicado, con una valoración de cero.
- No se corrigen prácticas que **no compilen**, ni que contengan **errores graves en ejecución** con una valoración de cero.
- No se admite **código no comentado** con la especificación para **javadoc** con una valoración de cero. Revisar los errores que puedan mostrarse en la consola al generar la documentación.
- No se admite no entregar la documentación **HTML** generada con **javadoc** con el doclet por defecto y con el doclet **DocCheck**, con una valoración de cero.
- Se valorará negativamente el porcentaje de fallos como resultado de filtrar la documentación **tanto con el propio javadoc**, como con el doclet **DocCheck**. Revisar los errores que también salen en la ejecución con la generación de la co
    - **NO se valorarán negativamente** los errores de tipo *Category 1: Package Error * No documentation for package. Need a package.html file*, los errores de tipo *Category 4: Text/link Error * Html Error in First Sentence --Missing a period -- {@inheritDoc}*, **así como los errores en el tipo enumerado de los métodos generados automáticamente.**

- **Es obligatorio seguir las convenciones de nombres vistas en teoría** en cuanto a los nombres de paquetes, clases, atributos y métodos.
- Se penalizarán los **errores en los tests automáticos proporcionados.**
- Aun superando todos los tests, si es **imposible completar una partida completa** tanto en el modo texto o gráfico, la calificación de la prática estará **penalizada un 40%**.
- Porcentajes/pesos[3] aproximados en la valoración de la práctica:

Apartado	Cuestiones a valorar	Porcentaje (Peso)
Cuestiones generales de funcionamiento	Integrada la solución con la interfaz gráfica (5%). Documentada sin errores DocCheck (5%). Scripts correctos (1%). Utiliza package-info. Extras de documentación (1%)	10,00%
Buscaminas (textui)	Corrección del algoritmo propuesto. Correcto funcionamiento. Uso correcto de tratamiento de excepciones.	5,00%
Arbitro ArbitroAbstracto ArbitroSeguro ArbitroInseguro	Uso de modificadores de acceso. Atributos correctos. Código no repetido en las implementaciones. Código no excesivamente largo en los métodos. Métodos de cambio de estado correctos. Métodos de consulta y clonación correctos. Tratamiento de excepciones correcto. Uso correcto de la herencia, con correcta decisión de atributos y métodos, en cada clase. Uso correcto de redefiniciones. Uso correcto de tratamiento de excepciones.	40,00%
DistanciaChebyshev	Correcta implementación de la clase funcional.	1,50%
Tablero	Uso de modificadores de acceso. Atributos correctos. Código no repetido en las implementaciones. Código no excesivamente largo en los métodos. Métodos de cambio de estado correctos. Métodos de consulta correctos. Clonación correcta de celdas y tablero. Colocación de minas correcta. Descubrir celdas correcto. Uso de constantes simbólicas. Migración correcta de array a solución genérica en la clase. Uso correcto de estructuras genéricas. Uso correcto de tratamiento de excepciones.	30,00%
Celda	Uso de modificadores de acceso. Atributos correctos. Clonación correcta. Cambios de estado correctos. Textos generados correctos. Métodos de cambio de estado correctos. Métodos de consulta correctos. Métodos hashCode y equals bien generados.	10,00%
Estado	Correcta implementación de enumeración estado.	1,50%
CoordenadasIncorrectasException	Excepción correcta	2,00%

Recursos

Bibliografía complementaria:

[Oracle, 2020] Lesson: Language Basics. The Java Tutorials. (2020). Disponible en http://docs.oracle.com/javase/tutorial/java/.

[JDK11, 2020] Random (Java SE 11 & JDK 11). Documentación en línea de la clase `java.util.Random`. Disponible en https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Random.html

Buscaminas. (2020, 24 de agosto). *Wikipedia, La enciclopedia libre*. Fecha de consulta: 08:41, octubre 18, 2020 desde https://es.wikipedia.org/w/index.php?title=Buscaminas&oldid=128709090.

Buscaminas (2020) Buscaminas Pro - jugar en línea gratis. Disponible en https://buscaminas-pro.com/

3 Los porcentajes pueden varias ligeramente al haber redondeado decimales.

Anexo. Recomendaciones en el uso de la interfaz `List` y la clase `java.util.ArrayList`

La interfaz `java.util.List` es implementada por la clase `java.util.ArrayList`. Dicha clase implementa un *array* que **dinámicamente** puede cambiar sus elementos, aumentando o disminuyendo su tamaño. Recordad que en contraposición, los *arrays* en Java son constantes en tamaño una vez determinado.

En esta **segunda práctica** utilizaremos la versión **GENÉRICA tanto de la interfaz como de la clase concreta.**

Una vez instanciado un `ArrayList` (con cualquiera de los constructores que aporta), tendremos una lista vacía. Cuando se añaden elementos se pueden utilizar los métodos:

`public boolean` `add(E e)`

Añade el elemento al final de la lista.

O bien:

`public void` `add(int index, E element)`

Inserta el elemento en la posición especificada. Desplaza el elemento en esa posición (si hay alguno) y todos los demás elementos a su derecha (añade 1 a sus índices)

Lanza una excepción IndexOutOfBoundsException – si el índice está fuera del rango (`index < 0 || index > size()`)

Mientras que el primer método añade siempre al final incrementando a su vez el tamaño (`size()`), el segundo permite añadir de forma indexada siempre que el índice **NO** esté fuera del rango (`index < 0 || index > size()`).

Si se quiere inicializar dimensionando de manera adecuada el número de elementos y se desconoce el valor a colocar en ese momento, se permite la inicialización con valores nulos (`null`) como se muestra en el ejemplo.

Ej:

```
// En este ejemplo se utiliza una variable de tipo interfaz para manejar el ArrayList.
// Siempre que sea posible se deben utilizar interfaces para manejar objetos concretos
List<Integer> array = new ArrayList<Integer>(10); // capacidad 10 pero tamaño 0
System.out.println("Tamaño del array list:" + array.size()); // se muestra 0 en pantalla

// cualquier intento de realizar una invocación a add(index,element) con index != 0
// provocaría una excepción de tipo IndexOutOfBoundsException
for (int i = 0; i < 10; i++){
    array.add(null);
}
// tamaño de array (size()) es 10 a la finalización del bucle
```

Para modificar una posición determinada se utiliza el método:

`public E` `set(int index,` **`E`** `element)`

Lanza una excepción IndexOutOfBoundsException – si el índice está fuera del rango (`index < 0 || index >= size()`)

Para consultar el elemento en una posición determinada se utiliza el método:

`public E get(int` `index)`

Lanza una excepción IndexOutOfBoundsException – si el índice está fuera del rango (`index < 0 || index >= size()`)

Para añadir todos los elementos de otro `ArrayList`, al final del `ArrayList` actual se puede utilizar el método:

```
public boolean addAll(Collection<? Extends E> c)
```

teniendo en cuenta que un `ArrayList` es una `Collection`.

Para extraer eliminando además el elemento del `ArrayList` se puede utilizar el método:

```
public E remove(int index)
```

Si queremos recorrer todos los elementos de un `List` con genericidad se puede utilizar un bucle `for-each`. Estos bucles facilitan el recorrido de estructuras iterables, avanzando automáticamente y llevando el control de finalización (sin embargo solo permiten recorrer en un sentido). Por ejemplo parar recorrer una lista de celdas y mostrar sus valores de fila y columna en pantalla:

```
public void mostrarCeldas(List<Celda> celdas) {
	for (Celda celda : celdas) {
		System.out.println(celda.obtenerFila() + "-" + celda.obtenerColumna());
	}
}
```

Para utilizar el resto de métodos (vaciar, consultar el número de elementos, etc.) se recomienda consultar la documentación en línea, aunque el conjunto de métodos indicados debería ser suficiente para la resolución de la práctica.

Licencia

Autor: Raúl Marticorena

Área de Lenguajes y Sistemas Informáticos

Departamento de Ingeniería Informática

Escuela Politécnica Superior

UNIVERSIDAD DE BURGOS

2020