Senku (v1)

Introducción

El Senku, o simplemente Solitario, es un pasatiempo de tablero. Se juega con un tablero con diferentes formas según el modelo o el lugar de origen. Al inicio del juego están todos los espacios ocupados, excepto por uno. El jugador debe mover una pieza por vez. Las piezas sólo pueden moverse capturando mediante un "salto" sobre otra, como en las damas. Sólo se puede capturar en horizontal o en vertical, nunca en diagonal. Así, al principio, sólo pocas tienen posibilidad de moverse, capturando una. El objetivo del juego es eliminar todas las piezas, dejando sólo una en el tablero.[1]

[1] https://es.wikipedia.org/wiki/Senku

https://commons.wikimedia.org/wiki/File:French_solitaire.jpg

Si queréis saber más sobre él, podéis consultar las páginas en la wikipedia:

- https://es.wikipedia.org/wiki/Senku
- https://en.wikipedia.org/wiki/Peg_solitaire

El objetivo de la práctica será construir un conjunto de clases que implementen el solitario:

- el sistema mostrará el tablero
- permitirá seleccionar la ficha que se quiere mover
    - solamente podrán seleccionarse fichas válidas
- permitirá seleccionar el hueco al que se quiere mover
    - solamente podrán seleccionarse huecos válidos
- el sistema indicará si el juego ha terminado, cuando ya no existen movimientos posibles

Proyecto que os proporcionamos

Teniendo en cuenta que la asignatura es de programación de primer curso, la parte de construcción de la solución consistente en dividir la solución en clases pequeñas que, combinadas, resuelvan el problema, es decir, lo que se conoce con el nombre de diseño orientado a objetos, la hemos realizado nosotros. Es por ello que tanto en el enunciado como en el proyecto que os proporcionamos, encontraréis:

- clases completamente implementadas
- **clases parcialmente implementadas** con las declaraciones de los métodos públicos (y algunos atributos) **a completar por vosotros** y una descripción de la funcionalidad que han de implementar dichos métodos
    - obviamente además de esos métodos públicos, podéis añadir miembros privados auxiliares (normalmente métodos)
- clases que comprueban el buen funcionamiento tanto de las clases que os proporcionamos implementadas como el de los métodos que debéis implementar
    - como siempre, pasar los test no garantiza que la función no contenga errores, pero no pasarlos garantiza que sí los contiene

Además la implementación **deberá realizarse en el orden en que las clases están descritas en el enunciado**, de manera que en cada paso se pueda usar el código implementado en los pasos anteriores.

De cara a construir las clases de prueba, en vez de hacer como en la primera práctica y construir un conjunto de métodos auxiliares para comprobar diversas cosas, hemos decidido utilizar la biblioteca de clases más comúnmente utilizada en java para hacer pruebas: **JUnit.** Como en esta práctica no deberéis crear nuevos tests, aunque no hay problema alguno en que lo hagáis, no hará falta disponer de un conocimiento exhaustivo de dicha biblioteca.

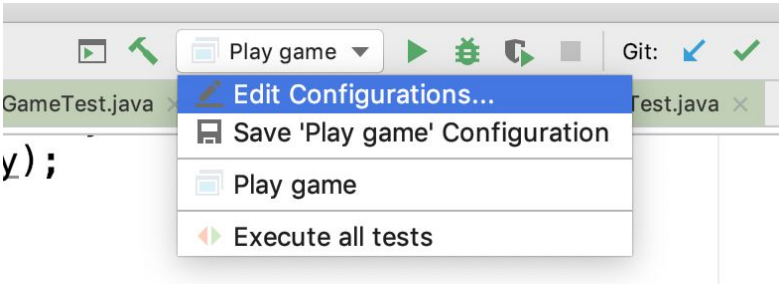
La primera vez que abráis el proyecto con IntelliJ IDEA, el entorno os indicará que hay problemas y, siguiendo los pasos que IntelliJ IDEA va indicando, se instalará JUnit en vuestro entorno. Todo esto también está explicado en la presentación sobre JUnit que se realizó en la sesión de laboratorio.

Ejecución del proyecto

El proyecto que os proporcionamos puede ejecutarse de dos formas diferentes:

- **Play game**: como una aplicación gráfica, en este caso para jugar al solitario
    - lo que solamente tendrá sentido cuando hayáis completado la solución
- **Execute all tests**: como una serie de tests que comprueban el funcionamiento de vuestra solución
    - a medida que vayáis completando la solución iréis viendo como el número de tests que pasan va aumentando

Para elegir qué tipo de ejecución queréis, usáis el desplegable existente en la barra de acciones del IntelliJ:



Descripción de las clases

La clase Position

Esta clase representa las posiciones, parejas de X e Y, dentro del tablero de juego. Dichas posiciones servirán para identificar cada una de las celdas que hay en él.

Esta clase es inmutable: una vez creada una instancia de posición, ésta ya no puede cambiar. Por ello, hemos definido sus dos variables de instancia con el calificador final, que indica que, una vez dichas variables han obtenido un valor, obligatoriamente en el constructor, éste es imposible de cambiar.

Su código es el siguiente:

```
public class Position {

    private final int x;
    private final int y;

    public Position(int x, int y) { ??? }

    public int getX() { ??? }

    public int getY() { ??? }

    public boolean colinear(Position other) { ??? }

    public int distance(Position other) { ??? }

    public Position middle(Position other) { ??? }

    // Needed for testing and debugging

    @Override
    public boolean equals(Object o) { ... }

    @Override
    public String toString() { ... }
}
```

Como veis, la clase tiene dos variables de instancia para las dos coordenadas que indican una posición. Los métodos que debéis implementar son:

- ➔ `public Position(int x, int y)`
  - ◆ construye la posición a partir de sus dos coordenadas
  - ◆ la posición (0,0) es la que está arriba a la izquierda y, por tanto, las x aumentan al movernos hacia la derecha, y las y al hacerlo hacia abajo

- ➔ `public int getX()`
  - ◆ devuelve la coordenada x (eje horizontal) de una posición

- ➔ `public int getY()`

- devuelve la coordenada y (eje vertical) de una posición

➔ `public boolean colinear(Position other)`
- indica si la posición receptora (this) y la posición que se recibe como parámetro (other) están sobre la misma línea horizontal o vertical

➔ `public int distance(Position other) { ??? }`
- calcula la distancia de Manhattan (o taxicab) entre la posición receptora (this) y la posición que se recibe como parámetro
- la distancia de Manhattan entre dos posiciones se define como la suma de los valores absolutos de las diferencias entre las coordenadas de ambas posiciones

➔ `public Position middle(Position other) { ??? }`
- devuelve la posición intermedia entre la posición receptora (this) y la que se recibe como parámetro
- la distancia media es la media (usando división entera) entre las coordenadas de ambas posiciones

Además de estos métodos, la clase implementa `equals` y `toString`.

- El primero de ellos es necesario para poder comparar si dos posiciones son iguales (y es utilizado en los tests por el método `assertEquals`). Por motivos técnicos que no vienen al caso, y que serán desvelados el año que viene en la asignatura *Estructuras de datos*, el parámetro que recibe es de clase Object, por lo que su implementación es algo más compleja que simplemente comparar las coordenadas de ambas posiciones.
- El segundo, lo hemos añadido por si, mientras resolvéis la práctica, queréis escribir el contenido de una posición (usando `print` o `println`).

La clase de prueba PositionTest

Esta clase contiene algunos tests que comprueban que el funcionamiento de la implementación es el esperado. Como ya se ha comentado algunas veces, que la implementación de una clase pase un conjunto de tests, no implica que la clase sea completamente correcta. En cambio, asumiendo que lo que comprueba el test es realmente el comportamiento esperado de una clase, si el test no pasa, la implementación es incorrecta.

Los tests que realiza la clase han sido implementados usando la librería JUnit 5 (que se ha explicado brevemente en una de las presentaciones que se han realizado en las sesiones de laboratorio). El código de la clase es el siguiente:

```
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;
```

```
class PositionTest {

    @Test
    void getter_and_constructor() {
        Position pos = new Position(4, 3);
        assertEquals(4, pos.getX());
        assertEquals(3, pos.getY());
    }

    @Test
    void colinear_horizontal() {
        Position pos1 = new Position(3, 8);
        Position pos2 = new Position(6, 8);
        assertTrue(pos1.colinear(pos2));
        assertTrue(pos2.colinear(pos1));
    }

    @Test
    void colinear_vertical() {
        Position pos1 = new Position(6, 4);
        Position pos2 = new Position(6, 8);
        assertTrue(pos1.colinear(pos2));
        assertTrue(pos2.colinear(pos1));
    }

    @Test
    void not_colinear() {
        Position pos1 = new Position(3, 8);
        Position pos2 = new Position(6, 4);
        assertFalse(pos1.colinear(pos2));
        assertFalse(pos2.colinear(pos1));
    }

    @Test
    void distance() {
        Position pos1 = new Position(3, -2);
        Position pos2 = new Position(6, 4);
        assertEquals(9, pos1.distance(pos2));
        assertEquals(9, pos2.distance(pos1));
    }

    @Test
    void middle() {
        Position pos1 = new Position(3, -2);
        Position pos2 = new Position(6, 4);
```

```
            assertEquals(new Position(4, 1), pos1.middle(pos2));
            assertEquals(new Position(4, 1), pos2.middle(pos1));
        }
    }
```

Los métodos de test realizan las siguientes comprobaciones:

- `getter_and_setter`
  - comprueba que los getters obtienen los valores que se han registrado en el constructor
- `colinear_horizontal`
  - comprueba que el método `colinear` devuelve cierto si las posiciones están en la misma línea horizontal
- `colinear_vertical`
  - comprueba que el método `colinear` devuelve cierto si las posiciones están en la misma línea vertical
- `not_colinear`
  - comprueba que el método `colinear` devuelve falso si las posiciones no están en la misma línea horizontal o verticalmente
- `distance`
  - comprueba que el método `distance` calcula correctamente la distancia entre dos posiciones
- `middle`
  - comprueba que el método `middle` calcula correctamente el punto medio entre dos posiciones
  - para comprobar el resultado es necesario que la clase Position tenga implementado el método equals (que os proporcionamos nosotros)

La clase Direction

Esta clase representa las posibles direcciones en las que podemos mover las fichas. Viene representada por las modificaciones en x e y que hacemos sobre las coordenadas.

Como puntos importantes, el constructor es privado, de manera que las únicas direcciones posibles son las que se pueden acceder como atributos públicos, estáticos y finales (además de a un array que las contiene todas). El método importante es `apply`, que transforma una posición en otra según las modificaciones guardadas en la instancia.

```
public class Direction {

    public static final Direction UP = new Direction(?, ?);
    public static final Direction RIGHT = new Direction(?, ?);
    public static final Direction DOWN = new Direction(?, ?);
    public static final Direction LEFT = new Direction(?, ?);
```

```
        public static final Direction[] ALL =
                    new Direction[]{UP, RIGHT, DOWN, LEFT};

        private final int dx;
        private final int dy;

        private Direction(int dx, int dy) { ??? }

        public Position apply(Position from) { ??? }
    }
```

En esta clase se han de implementar:

- ➔ Inicializaciones de cada una de las direcciones posibles
  - ◆ en el código debéis sustituir el `null` que aparece por la llamada al constructor con los valores adecuados de los parámetros.
- ➔ `private Direction(int dx, int dy)`
  - ◆ constructor privado que inicializa las variables de instancia
- ➔ `public Position apply(Position from)`
  - ◆ devuelve la posición consistente en habernos movido según la dirección del objeto receptor, desde la posición pasada como parámetro
  - ◆ para un mejor entendimiento del método apply es conveniente que estudiéis el código de la clase DirectionTest

Utilizando Java un poco más avanzado, lo lógico sería haber implementado la clase Direction como un tipo enumerado[2].

El array ALL permitirá acceder a todas las direcciones existentes. Será útil cuando, por ejemplo, busquemos si hay algún movimiento vàlido en alguna de las direcciones ya que podremos escribir código como:

```
for (int i = 0; i < Direction.ALL.length; i++) {
    Direction direction = Direction.ALL[i];
    ...
}
```

La clase Cell

Esta clase representa cada una de las celdas del tablero. Cada una de ellas puede ser:

- prohibida (`FORBIDDEN`), es decir, que no puede albergar ficha alguna
- tener una ficha y estar ocupada (`FILLED`)
- estar vacía (`EMPTY`)

[2] Podéis encontrar información sobre tipos enumerados en el tutorial oficial de Java (https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html).

Las celdas pueden ir pasando de ocupadas a libres y viceversa según los movimientos y capturas que se realicen a lo largo de la partida.

Como en el caso de `Direction`, como sabemos que solamente hay tres estados posibles, solamente crearemos tres instancias dentro de la clase y definiremos el constructor privado. Como en el caso anterior, en Java más avanzado, lo razonable sería haber modelado la clase como un tipo enumerado.

```
public class Cell {

    private static final char C_FORBIDDEN = '#';
    private static final char C_FILLED = 'o';
    private static final char C_EMPTY = '·';

    public static final Cell FORBIDDEN = new Cell(???);
    public static final Cell FILLED = new Cell(???);
    public static final Cell EMPTY = new Cell(???);

    private final char status;

    private Cell(char status) { ??? }

    public static Cell fromChar(char status) { ??? }

    public boolean isForbidden() { ??? }

    public boolean isFilled() { ??? }

    public boolean isEmpty() { ??? }

    @Override
    public String toString() {
        return String.valueOf(status);
    }
}
```

La clase utiliza un caràcter para distinguir los diferentes estados en los que puede estar (dicho carácter será el que se usará para leer y escribir el tablero). Como el constructor es privado, además de disponer de atributos públicos, estáticos y finales para acceder a cada uno de los posibles estados, disponemos del método estático `fromChar`, que permite crear una celda a partir del carácter.

- ➔ Inicializaciones de cada una de las direcciones posibles
  - ◆ en el código debéis sustituir el `null` que aparece por la llamada al constructor con los valores adecuados de los parámetros.

- ➔ `private Cell(char status)`
  - ◆ inicializa el atributo status con el parámetro que se le pasa
- ➔ `public static Cell fromChar(char status)`
  - ◆ devuelve la instancia de Cell correspondiente al caràcter: '#' para FORBIDDEN, 'o' para FILLED y '·' para EMPTY o null en caso de que el carácter pasado como parámetro no sea uno de esos tres
- ➔ `public boolean isForbidden()`
  - ◆ indica si el objeto receptor se corresponde con la celda prohibida
- ➔ `public boolean isFilled()`
  - ◆ indica si el objeto receptor se corresponde con la celda llena
- ➔ `public boolean isEmpty()`
  - ◆ indica si el objeto receptor se corresponde con la celda vacía

La clase Board

Esta clase representa el tablero de juego. Consta de atributos para sus dimensiones y una matriz para sus celdas. El tablero se inicializa a partir de un `String` que contiene, en cada línea, los caracteres correspondientes a cada una de sus celdas. **Para simplificar, supondremos que la cadena es correcta, es decir que se corresponde con un tablero de las dimensiones dadas.**

La diferencia principal entre esta clase y la clase `Game` es que la clase `Board` es ajena a las reglas del juego: permite modificaciones cualesquiera de las celdas.

```
public class Board {

    private final int width;
    private final int height;
    private final Cell[][] cells;

    public Board(int width, int height, String board) { ??? }

    public int getWidth() { ??? }

    public int getHeight() { ??? }

    public boolean isForbidden(Position pos) { ??? }

    public boolean isFilled(Position pos) { ??? }

    public boolean isEmpty(Position pos) { ??? }

    public void fillPosition(Position pos) { ??? }

    public void emptyPosition(Position pos) { ??? }
```

```
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        for(int y=0; y < height; y++) {
            for(int x=0; x < width; x++) {
                sb.append(cells[y][x].toString());
            }
            if (y != height-1) {
                sb.append("\n");
            }
        }
        return sb.toString();
    }
}
```

Los métodos a implementar en la clase son:

- ➔ `public Board(int width, int height, String board)`
  - ◆ construye una instancia de tablero con un tamaño de width celdas de anchura, height de altura y con las celdas según los caracteres de board, que consta de heigh líneas (acabadas en '\n') de width caracteres cada una.
- ➔ `public int getWidth()`
  - ◆ devuelve la anchura del tablero
- ➔ `public int getHeight()`
  - ◆ devuelve la altura del tablero
- ➔ `public boolean isForbidden(Position pos)`
  - ◆ indica si la posición dada está prohibida, ya sea porque está fuera del tablero o porque la celda que contiene está prohibida
- ➔ `public boolean isFilled(Position pos)`
  - ◆ indica si la posición dada está ocupada
  - ◆ las celdas fuera del tablero nunca están ocupadas
- ➔ `public boolean isEmpty(Position pos)`
  - ◆ indica si la posición dada está libre
  - ◆ las celdas fuera del tablero nunca están libres
- ➔ `public void fillPosition(Position pos)`
  - ◆ marca la celda correspondiente a la posición pos (que se supone que no es prohibida) como ocupada
- ➔ `public void emptyPosition(Position pos)`
  - ◆ marca la celda correspondiente a la posición pos (que se supone que no es prohibida) como libre

La clase Game

Esta clase representa la situación de la partida, que es básicamente el tablero. La diferencia con la clase `Board` es que ésta sí conoce las reglas del juego y, por tanto, es capaz de

indicar si una posición es seleccionable como inicio de un movimiento, si otra es seleccionable como fin del mismo y es capaz de realizar un movimiento.

```
public class Game {

  private final Board board;

  public Game(Board board) {???}

  public boolean isValidFrom(Position from) {???}

  // Assumes validFrom is a valid starting position
  public boolean isValidTo(Position validFrom, Position to) {???}

  // Assumes both positions are valid
  public Position move(Position validFrom, Position validTo) {???}

  public boolean hasValidMovesFrom() {???}

  public int countValidMovesFrom() {???}

  // Assumes validFrom is a valid starting position
  public int countValidMovesTo(Position validFrom) {???}

}
```

Los métodos a implementar son:

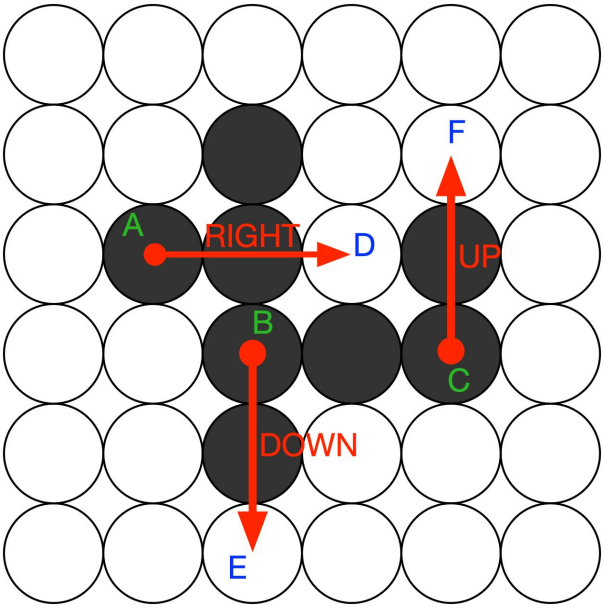
- ➔ `public Game(Board board)`
  - ◆ construye una partida y guarda una referencia al tablero sobre el que se jugará
- ➔ `public boolean isValidFrom(Position from)`
  - ◆ indica si la posición es un posible inicio de un movimiento
  - ◆ es decir, es una posición ocupada que, en una de las cuatro direcciones posibles, puede matar (saltando por encima), a una ficha adyacente
- ➔ `public boolean isValidTo(Position validFrom, Position to)`
  - ◆ indica si la posición to es una posición de llegada partiendo de la posición validFrom, que se supone válida
  - ◆ es decir, la posición no está vacía y es la casilla de destino al matar la pieza adyacente desde validFrom en la dirección adecuada
- ➔ `public Position move(Position validFrom, Position validTo)`
  - ◆ dadas las posiciones validFrom y validTo que representan el inicio y final de un movimiento, lo ejecutan modificando el estado del tablero, y devuelven la posición de la pieza intermedia (la que se ha matado)
- ➔ `public boolean hasValidMovesFrom()`
  - ◆ indica si en el tablero existe alguna posición válida como inicio de una jugada
- ➔ `public int countValidMovesFrom()`

- indica el número de movimientos posibles (posiciones seleccionables como origen)

➔ `public int countValidMovesTo(Position validFrom)`

- supone como precondición que `validFrom` es una posición válida como inicio de una jugada
- indica el número de movimientos posibles (posiciones seleccionables como destino)

Por ejemplo, en la siguiente figura:



las posiciones oscuras están ocupadas y las blancas están libres. La posición A es válida como posición inicial ya que la ficha que la ocupa podría saltar sobre la que tiene a la derecha para ocupar la posición libre D. A su vez, B también es una posición inicial válida, ya que la ficha que la ocupa podría saltar sobre la ficha que tiene debajo para ocupar la posición E. Lo mismo sucede con la posición C ya que podría saltar sobre la que tiene hacia arriba y ocupar la posición vacía F.

Si seleccionamos, por ejemplo, la posición A como inicial, la única posición válida como final es la D.

La clase Geometry

En esta clase se han agrupado todos los métodos que calculan las posiciones y dimensiones de los diferentes elementos de la interfaz gráfica.

Esta clase utiliza las clases `GPoint` y `GDimension` de la librería de la acm:

- `GPoint`: representar las coordenadas de un punto en la pantalla (con getters para la x y la y, que son floats)
- `GDimension`: representa las dimensiones de un elemento en la pantalla (con getters para height y width que son floats)

```
public class Geometry {

  private final int windowWidth;
  private final int windowHeight;
  private final int numCols;
  private final int numRows;
  private final double boardPadding;
  private final double cellPadding;

  public Geometry(
    int windowWidth,
    int windowHeight,
    int numCols,
    int numRows,
    double boardPadding,
    double cellPadding) { ??? }

  public int getRows() { ??? }

  public int getColumns() { ??? }

  public GDimension boardDimension() { ??? }

  public GPoint boardTopLeft() { ??? }

  public GDimension cellDimension() { ??? }

  public GPoint cellTopLeft(int x, int y) { ??? }

  public GDimension tokenDimension() { ??? }

  public GPoint tokenTopLeft(int x, int y) { ??? }

  public GPoint centerAt(Position position) { ??? }

  public Position xyToCell(double x, double y) { ... }
}
```

Los métodos a implementar son:

➔ `public Geometry(...)`

- ◆ construye una instancia de Geometry a partir de los siguientes valores:

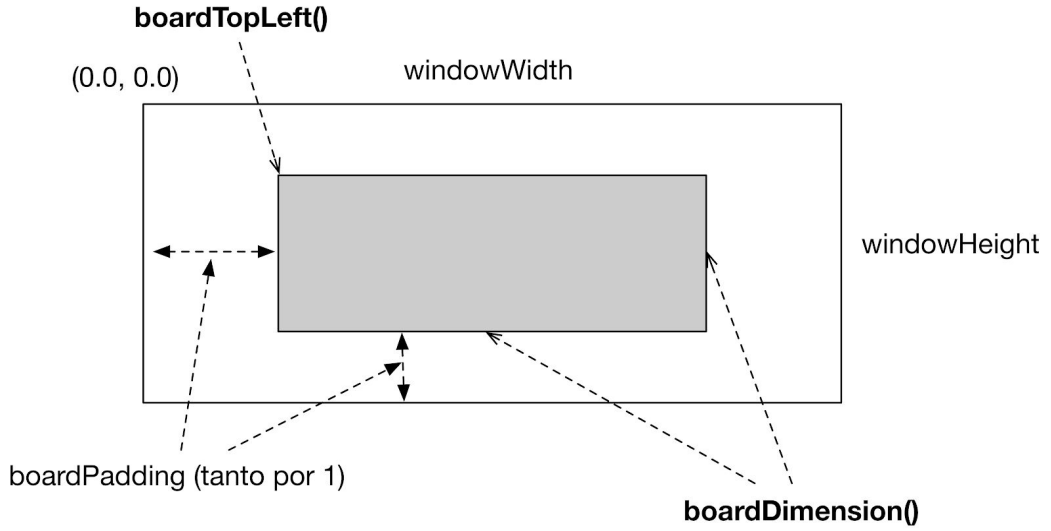
- `windowWidth`: anchura de la pantalla
- `windowHeight`: altura de la pantalla
- `numCols`: número de columnas del tablero
- `nunRows`: número de filas del tablero
- `boardPadding`: porción, en tanto por uno, del margen entre la parte interna del tablero y la pantalla
- `cellPadding`: porción, en tanto por uno, del margen entre cada una de las celdas del tablero y el tamaño de la ficha que la ocupa

➔ `public int getRows()`
- ◆ devuelve el número de filas

➔ `public int gelColumns()`
- ◆ devuelve el número de columnas

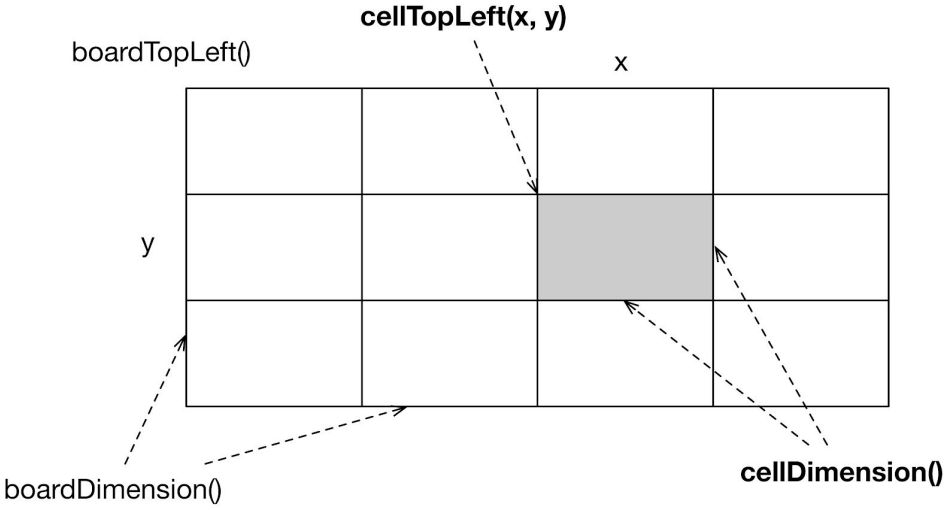
➔ `public GDimension boardDimension()`
- ◆ devuelve las dimensiones de la parte interna del tablero

➔ `public GPoint boardTopLeft()`
- ◆ devuelve las coordenadas del extremo superior izquierdo de la parte interna del tablero



➔ `public GDimension cellDimension()`
- ◆ devuelve las dimensiones de cada una de las celdas en las que se divide el tablero

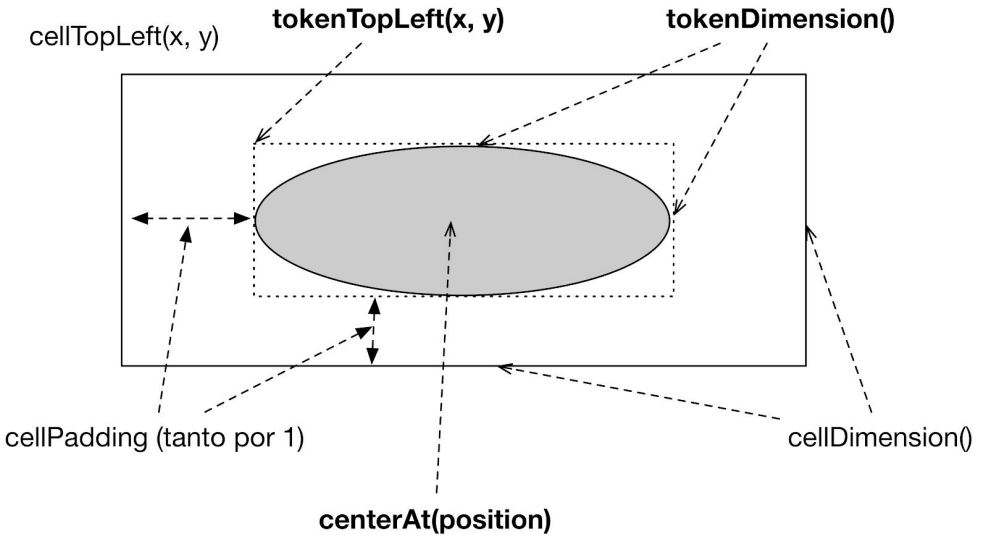
➔ `public GPoint cellTopLeft(int x, int y)`
- ◆ devuelve las coordenadas del extremo superior izquierdo de la celda que ocupa la columna x y fila y (comenzando desde 0).



➔ `public GDimension tokenDimension()`
- ◆ devuelve las dimensiones de los ovales que representan cada una de los ovales que representan las fichas

➔ `public GPoint tokenTopLeft(int x, int y)`
- ◆ devuelve las coordenadas del extremo superior izquierdo de cada una de los ovales que representan la ficha con columna x y fila y.

➔ `public GPoint centerAt(int x, int y)`
- ◆ devuelve las coordenadas del centro de la celda con columna x y fila y.



➔ `public Position xyToCell(double x, double y)`
- ◆ devuelve la posición (fila y columna) correspondiente a las coordenadas de la pantalla x e y (como son coordenadas de la pantalla se trata de valores expresados como double).
- ◆ No hace falta de que os preocupéis de que se trata de una posición válida para una ficha (p.e. en caso de que sean coordenadas existentes en el

borde, pueden aparecer posiciones con filas y columnas negativas o fuera de los límites).
- Os lo damos completamente implementado

La clase Palette

Esta clase os la damos completamente implementada. Su propósito es definir los colores que se usarán en la interfaz y algunas funciones que los manipulan.

La clase Display

Esta es la clase que implementa los métodos que construyen la interfaz gráfica y que utiliza la clase Geometry para los cálculos de posiciones y dimensiones y Palette para la gestión de los colores.

Los métodos:
- `initializeDisplay`
- `initializeRow`
- `initializePosition`
- `paintEmpty`
- `paintFilled`
- `createOval`

son los responsables de inicializar la interfaz gráfica en función del número de filas y columnas y de la descripción del tablero dada en un `String`. Como se indica en el código no se comprueba que dicha información sea coherente.

El tablero inicial contiene todas las posiciones (círculos) ocupadas por fichas excepto la posición central, que está vacía.

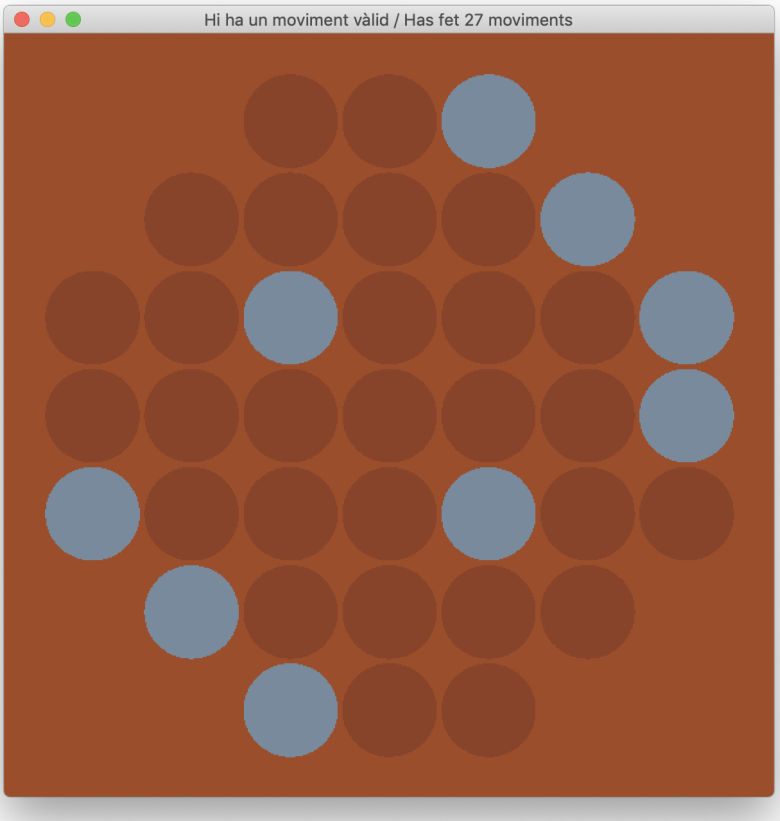
Los métodos:
- `setEmpty`
- `setFilled`
- `setColor`

se encargan de cambiar el estado de una posición (de llena a vacía y viceversa).

La parte más compleja de la clase es la que se dedica a seleccionar y deseleccionar posiciones. Para ello la clase dispone de dos variables de instancia:
- **highlighted**: elemento gráfico que, si el cursor está sobre él, se muestra de color resaltado pues es un origen origen/destino posible de un movimiento.
- **selected**: elemento gráfico (que representa una ficha) que se ha seleccionado como origen del siguiente movimiento.

El manejo de estos valores se realiza con los métodos:
- `highlight`

- `ignoreHighlight`
- `clearHighlight`
- `select`
- `ignoreSelection`
- `clearSelect`
- `setColor`



La clase Senku

Es la clase que representa el programa principal. Define constantes para los diferentes elementos de la aplicación:

- `APPLICATION_WIDTH`: anchura de la ventana del programa (en píxeles)
- `APPLICATION_HEIGHT`: altura de la ventana del programa (en píxeles)
- `OUTER`: tanto por uno del margen externo del tablero
- `INNER`: tanto por uno del margen en cada una de las celdas
- `ROWS`: número de filas que tiene el tablero
- `COLUMNS`: número de columnas que tiene el tablero
- `BOARD`: String que representa la configuración inicial del tablero (posiciones válidas y ocupadas)

El método run inicializa los diferentes elementos que se necesitan para ejecutar la partida. Como la interfaz necesitará acceder a las posiciones del ratón, ha de llamar al método `addMouseListeners`.

Para indicar un movimiento deberemos clicar sobre una ficha que pueda saltar sobre una de sus vecinas para eliminarla. Para reconocer si una ficha puede ser origen de un movimiento la interfaz resalta una ficha al pasar por encima si ésta puede ser un origen válido. Además la interfaz solamente nos permitirá seleccionar una posición si ésta puede ser origen de un movimiento. Una vez seleccionado un origen válido, que queda resaltado en la interfaz, debemos escoger un destino válido. Aquí la interfaz también nos ayudará al resaltar, cuando pasamos por encima, una posición si ésta puede ser el destino del movimiento. También, si volvemos a clicar sobre una posición seleccionada como origen, ésta se deselecciona.

Para poder realizar todo esto, la variable de instancia `selectedFrom` sirve para guardar la posición origen seleccionada para el movimiento.

Para gestionar el resaltado al pasar por encima de una posición que puede ser el inicio/final de un movimiento, se usan los métodos:

- `mouseMoved`
- `highlightOrClearFrom`
- `highlightOrClearTo`

Y para seleccionar las posiciones inicial y final de un movimiento, los métodos:

- `mouseClicked`
- `selectIfValid`
- `unselect`
- `move`

Finalmente, para gestionar el mensaje que aparece en la barra del título, se usa el método:

- `updateTitle`

Formato de la entrega

- Fichero **ZIP** con
  - el **directorio del proyecto IntelliJ**
  - el informe en **PDF**
- A entregar vía el **campus virtual**

Criterios de evaluación

NOTA = 70% programación + 30% informe

Programación

Como siempre no tan sólo valoraremos que la función calcule el resultado correctamente, sino que el código sea entendible (bien indentado, variables correctas, descomposición descendente usando funciones auxiliares, etc., etc.).

El orden de implementación de las clases será:

1. `Position` (1 punto)
2. `Direction` (1 punto)
3. `Cell` (1 punto)
4. `Board` (1 punto)
5. `Game` (2 punto)
6. `Geometry` (1 punto)

En el código que os pasamos, cada una de los métodos que debéis completar tiene en su implementación la instrucción:

```
throw new UnsupportedOperationException("Step 3");
```

El número, en este caso **3** se corresponde con cada uno de los elementos de la lista numerada anterior. Recordad que ese número también indica el orden en el que debéis ir implementando cada uno de los métodos.

Informe

De cara a realizar un buen informe que, como veis, constituye una parte muy sustancial de la nota final, es extremadamente importante que toméis notas mientras resolvéis la práctica. De hecho, como buena práctica, es importante que realicéis esbozos, diagramas, etc, en papel antes de poneros a programar.

El informe, de unas dos o tres páginas (sin contar la portada), deberá tener la siguiente estructura:

- Portada (nombre, DNI o similar, grupo)
- Descripción del desarrollo de la práctica
    - problemas que os habéis encontrado y cómo los habéis resuelto, tanto a nivel de:
        - comprensión del enunciado
        - programación de vuestra solución
        - uso del entorno de programación
    - descripción detallada para las **3 funciones que consideréis más complicadas** de cómo habéis diseñado la solución.
        - podéis añadir esquemas o diagramas para ilustrar vuestro enfoque (podéis escanearlos)
- Conclusiones:
    - ¿qué habéis aprendido resolviendo la práctica?
    - si volvierais a empezar la práctica, ¿qué haríais diferente tanto a nivel de código como de método de resolución?