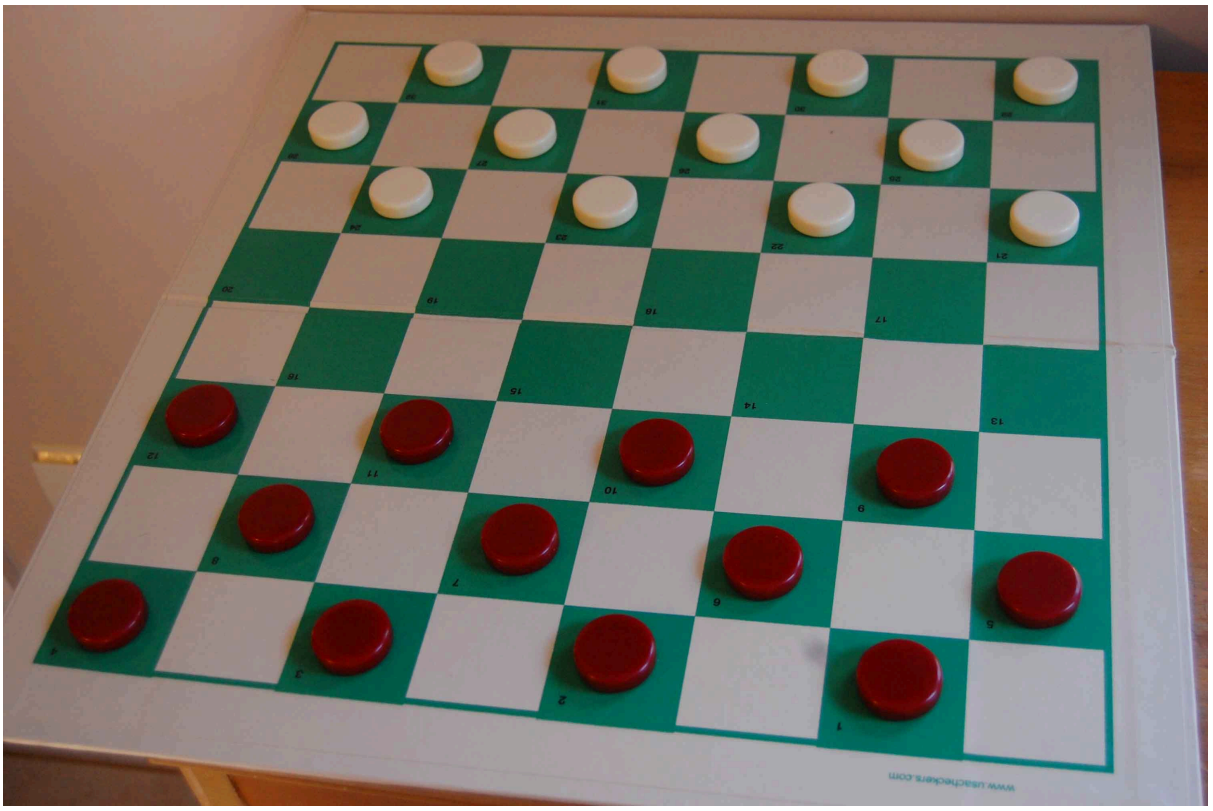
Checkers (v2)

Introducción

El objetivo de esta práctica es construir una aplicación para jugar, entre dos jugadores, a una **versión simplificada** del conocido juego de las **damas**. Las simplificaciones son las siguientes:

- en cada turno solamente se realiza un movimiento o una captura
    - es decir, las capturas no pueden encadenarse
- cuando una ficha llega a la última fila, no se convierte en dama, sino que la partida acaba, siendo ganador el jugador que lo ha logrado
- en ningún momento es obligatorio capturar



[Jud McCranie / CC BY (https://creativecommons.org/licenses/by/3.0)]

El objetivo de la práctica será construir un conjunto de clases que implementen esta versión simplificada del juego:

- 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

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 como 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 pero, si fuera necesario, variables)**
- 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 realizar comprobaciones, 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: simplemente con saber leer los tests será suficiente, ya que leerlos os ayudará a comprobar el funcionamiento de los métodos a implementar.

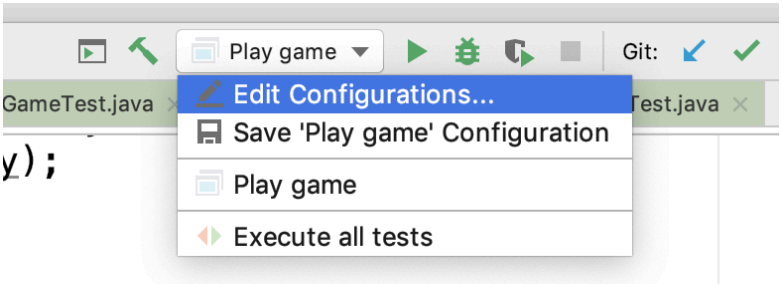
La primera vez que abráis el proyecto con IntelliJ IDEA, el entorno os indicará que hay problemas y, siguiendo los pasos que IntelliJ IDEA va indicando, se instalará JUnit en vuestro entorno. Todo esto también está explicado en la presentación sobre JUnit de la sesión de laboratorio (Introducción a JUnit5, en recursos/Laboratori/Presentacions).

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 a las damas
    - lo que solamente tendrá sentido cuando hayáis completado la solución ya que, si falta algún método por implementar, acabará lanzando un error.
- **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
    - obviamente los tests sobre partes no implementadas fallarán

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



Descripción de las clases

La clase Player

Esta clase representa a cada uno de los jugadores y os la damos completamente implementada.

```
public class Player {

    public static final Player WHITE = new Player("WHITE");
    public static final Player BLACK = new Player("BLACK");

    private final String name;

    private Player(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return name;
    }
}
```

Esta clase, que en “Java avanzado” se representaría como un tipo enumerado[1], tiene un constructor privado, de manera que sus únicas dos instancias posibles son las que se crean internamente y a las que se puede acceder ya que disponemos de dos atributos públicos estáticos.

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.

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

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 sameDiagonalAs(Position other) {???}

    public static int distance(Position pos1, Position pos2) {???}

    public static Position middle(Position pos1, Position pos2) {
     ???
    }

    // Needed for testing and debugging

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

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

De la misma manera que la clase Player, en Java avanzado, se representaría con un tipo enumerado, la clase Position, en java moderno, podría representarse por un record[2].

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

[2] Podéis encontrar más información sobre los records, añadidos en Java 16, en Record Classes.

- 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 sameDiagonalAs(Position other)`
- indica si la posición receptora (`this`) y la posición que se recibe como parámetro (`other`) están sobre la misma línea diagonal, ya sea en sentido / como en sentido \

➔ `public static int distance(Position pos1, Position pos2)`
- calcula la distancia de Manhattan (o taxicab) entre las posiciones `pos1` y `pos2`. 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 static Position middle(Position pos1, Position pos2)`
- devuelve la posición intermedia entre las posiciones `pos1` y `pos2`
- la posición media es aquella que tiene por coordenadas a las medias de las coordenadas de las posiciones. Es decir, para la coordenada `x`, será la media, **usando división entera**, entre `pos1.x` y `pos2.x`; para la coordenada `y` el cálculo será similar.

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 próximo curso 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 explica brevemente en una de las presentaciones 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 diagonal() {
        Position p00 = new Position(0, 0);
        Position p01 = new Position(0, 1);
        Position p22 = new Position(2, 2);
        Position p12 = new Position(1, 2);
        assertTrue(p00.sameDiagonalAs(p22));
        assertTrue(p22.sameDiagonalAs(p00));
        assertTrue(p01.sameDiagonalAs(p12));
        assertTrue(p12.sameDiagonalAs(p01));
        assertFalse(p00.sameDiagonalAs(p01));
        assertFalse(p01.sameDiagonalAs(p00));
        assertFalse(p12.sameDiagonalAs(p22));
        assertFalse(p22.sameDiagonalAs(p12));
    }

    @Test
    void contra_diagonal() {
        Position p10 = new Position(1, 0);
        Position p01 = new Position(0, 1);
        Position p22 = new Position(2, 2);
        Position p13 = new Position(1, 3);
        assertTrue(p10.sameDiagonalAs(p01));
        assertTrue(p01.sameDiagonalAs(p10));
        assertTrue(p22.sameDiagonalAs(p13));
        assertTrue(p13.sameDiagonalAs(p22));
        assertFalse(p10.sameDiagonalAs(p22));
        assertFalse(p22.sameDiagonalAs(p10));
```

```
        assertFalse(p13.sameDiagonalAs(p01));
        assertFalse(p01.sameDiagonalAs(p13));
    }

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

    @Test
    void middle() {
        Position pos1 = new Position(3, -2);
        Position pos2 = new Position(6, 4);
        assertEquals(new Position(4, 1),
                     Position.middle(pos1, pos2));
        assertEquals(new Position(4, 1),
                     Position.middle(pos2, pos1));
    }
}
```

Los métodos de test realizan las siguientes comprobaciones:

- `getter_and_constructor`
  - comprueba que los getters obtienen los valores que se han registrado en el constructor
- `diagonal`
  - comprueba que el método `sameDiagonalAs` devuelve cierto si las posiciones están en la misma diagonal en sentido \
- `contra_diagonal`
  - comprueba que el método `sameDiagonalAs` devuelve cierto si las posiciones están en la misma diagonal en sentido /
- `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; dos de ellos serán válidos para las blancas, NW y NE; y dos para las negras, SW y SE. 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 NW = new Direction(?, ?);
    public static final Direction NE = new Direction(?, ?);
    public static final Direction SW = new Direction(?, ?);
    public static final Direction SE = new Direction(?, ?);

    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.

La clase Cell

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

- prohibida (`FORBIDDEN`), es decir, que no puede albergar ficha alguna
- vacía (`EMPTY`), es decir, aunque pudiera ser ocupada por una ficha, no lo está en estos momentos
- blanca (`WHITE`), ocupada por una ficha blanca
- negra (`BLACK`), ocupada por una ficha negra

Las celdas pueden ir pasando de ocupadas (por una ficha blanca o negra) a vacías 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 cuatro estados posibles, solamente crearemos cuatro 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_EMPTY = ' ';
    private static final char C_WHITE = 'w';
    private static final char C_BLACK = 'b';

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

    private final char status;

    private Cell(char status) {???}

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

    public boolean isForbidden() {???}

    public boolean isEmpty() {???}

    public boolean isWhite() {???}

    public boolean isBlack() {???}

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

La clase utiliza un carácter para distinguir los diferentes estados en los que puede estar (dicho carácter será el que se usará para leer y escribir el tablero en los tests). 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`, `' '` para `EMPTY`, `'w'` para `WHITE`, `'b'` para `BLACK` y, finalmente, `null` en caso de que el carácter pasado como parámetro no sea uno de esos cuatro
- ➔ `public boolean isForbidden()`
  - ◆ indica si el objeto receptor se corresponde con la celda prohibida
- ➔ `public boolean isEmpty()`
  - ◆ indica si el objeto receptor se corresponde con la celda vacía
- ➔ `public boolean isWhite()`
  - ◆ indica si el objeto receptor se corresponde con la celda ocupada por una ficha blanca
- ➔ `public boolean isBlack()`
  - ◆ indica si el objeto receptor se corresponde con la celda ocupada por una ficha negra

La clase Board

Esta clase representa el tablero de juego. Consta de atributos para sus dimensiones y una matriz para sus celdas y, además, mantiene contadores sobre el número de fichas blancas y negras que contiene.

El tablero se inicializará 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.**[3]

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 sin que éstas respeten las reglas del juego.

[3] Recordad que, al no formar las excepciones parte del temario de esta asignatura, muchos aspectos sobre el tratamiento de errores son obviados. En Estructuras de Datos ya veréis cómo podríamos diseñar una clase Board que sí tuviera en cuenta este tipo de situaciones.

```
public class Board {

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

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

    public int getWidth() {???}

    public int getHeight() {???}

    public int getNumBlacks() {???}

    public int getNumWhites() {???}

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

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

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

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

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

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

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

    // For testing and debugging

    @Override
    public String 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 `height` 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 int getNumBlacks()`
  - ◆ devuelve el número de fichas negras que hay en el tablero
- ➔ `public int getNumWhites()`
  - ◆ devuelve el número de fichas blancas que hay en el 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 isBlack(Position pos)`
  - ◆ indica si la posición dada está ocupada por una ficha negra
  - ◆ las celdas fuera del tablero nunca están ocupadas
- ➔ `public boolean isWhite(Position pos)`
  - ◆ indica si la posición dada está ocupada por una ficha blanca
  - ◆ 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 setBlack(Position pos)`
  - ◆ ocupa con una ficha negra la celda correspondiente a la posición `pos` (que podéis suponer que no es prohibida)
- ➔ `public void setWhite(Position pos)`
  - ◆ ocupa con una ficha negra la celda correspondiente a la posición `pos` (que podéis suponer que no es prohibida)
- ➔ `public void setEmpty(Position pos)`
  - ◆ desocupa (marca como vacía) la celda correspondiente a la posición `pos` (que podéis suponer que no es prohibida)

La clase Move

Esta clase, que os damos ya implementada, se utilizará tanto en los tests como en la interfaz gráfica para indicar el movimiento que se ha realizado. Su código es:

```
public class Move {

    // Assumes it is a valid move !!!

    private final Position from;
    private final Position middle;
    private final Position to;

    public Move(Position from, Position middle, Position to){
        this.from = from;
        this.middle = middle;
        this.to = to;
```

```
    }

    public Position getFrom() {
        return from;
    }

    public Position getMiddle() {
        return middle;
    }

    public Position getTo() {
        return to;
    }

    public boolean isCapture() {
        return middle != null;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass())
               return false;
        Move move = (Move) o;
        return from.equals(move.from) &&
                Objects.equals(middle, move.middle) &&
                to.equals(move.to);
    }

    @Override
    public int hashCode() {
        return Objects.hash(from, middle, to);
    }

    @Override
    public String toString() {
        return "Move{" +
                "from=" + from +
                ", middle=" + middle +
                ", to=" + to +
                '}';
    }
}
```

Los atributos de la clase representan las posiciones involucradas en un movimiento:

- `from`: posición origen de la ficha que se ha movido

- `to`: posición destino de la ficha que se ha movido
- `middle`: en caso de ser una captura, posición de la ficha capturada; en caso de no ser una captura su valor será `null`.

La clase Game

Esta clase representa la situación de la partida, que es básicamente el tablero, el jugador al que le toca el turno de juego y un booleano indicando si el jugador ha ganado la partida.

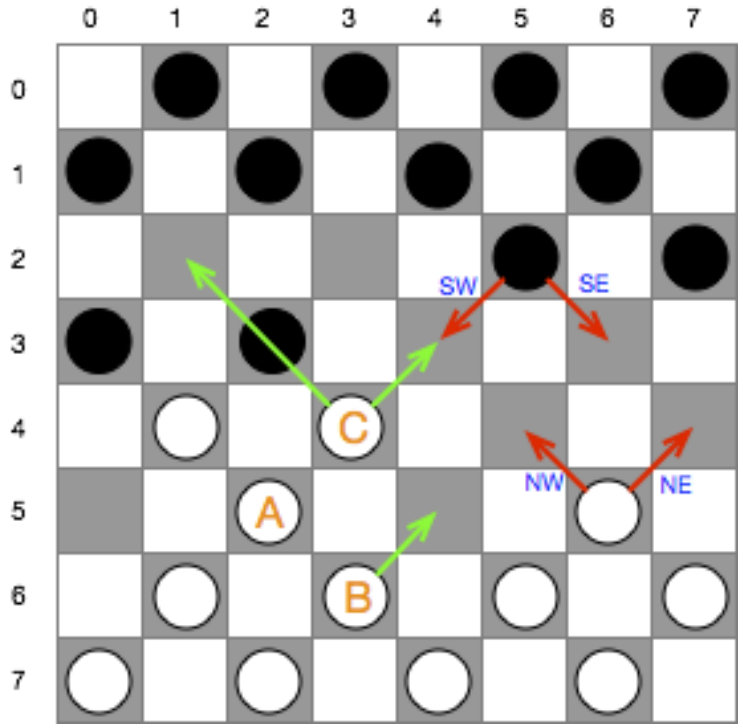
La diferencia con la clase `Board` es que `Game` sí conoce las reglas del juego y, por tanto, es capaz de indicar si una posición es seleccionable como inicio de un movimiento, si otra es seleccionable como fin del mismo y es capaz de realizar un movimiento.

```
public class Game {

    private static final Direction[] WHITE_DIRECTIONS =
{Direction.NW, Direction.NE};
    private static final Direction[] BLACK_DIRECTIONS =
{Direction.SW, Direction.SE};

    private final Board board;
    private Player currentPlayer;
    private boolean hasWon;

    public Game(Board board) {???}

    public Player getCurrentPlayer() {???}

    public boolean hasWon() {???}

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

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

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

    // Only for testing
    public void setPlayerForTest(Player player) {???}
}
```

Los métodos a implementar son:

- ➔ `public Game(Board board)`
  - ◆ construye una partida y guarda una referencia al tablero sobre el que se jugará
  - ◆ el turno es el del jugador de fichas blancas
  - ◆ para simplificar podemos suponer que ningún jugador ha ganado la partida
- ➔ `public Player getCurrentPlayer()`
  - ◆ devuelve el jugador que tiene el turno de juego
- ➔ `public boolean hasWon()`
  - ◆ devuelve true si el jugador actual ha ganado la partida, false en caso contrario
- ➔ `public boolean isValidFrom(Position from)`
  - ◆ indica si la posición es un posible inicio de un movimiento para el jugador actual
  - ◆ es decir, es una posición ocupada que, en una de las dos direcciones posibles, puede avanzar o capturar
- ➔ `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 está vacía y es la casilla de destino al avanzar o al capturar desde la posición `validFrom`
- ➔ `public Move 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
  - ◆ ambas posiciones se suponen válidas
  - ◆ devuelve un objeto `Move` que indica las posiciones involucradas en el movimiento realizado.

Para entender mejor los conceptos involucrados en la clase, considerad el siguiente diagrama:



Como en el juego de las damas las piezas avanzan siempre en diagonal, ya sea para hacer movimientos simples como capturas, las direcciones válidas posibles de un movimiento de las piezas blancas son **NW** y **NE** (valor del array `WHITE_DIRECTIONS`). De forma similar, las únicas direcciones potencialmente válidas para una pieza negra son **SW** y **SW**, que es justamente el valor del array `BLACK_DIRECTIONS`.

En la posición mostrada en el diagrama es el turno de las blancas. En este turno, por ejemplo, la ficha mostrada como **A** no tiene ningún movimiento posible; la pieza **B** solamente puede moverse en dirección NE haciendo un movimiento simple; y la pieza **C** puede hacer un movimiento de captura en dirección NW y, también, un movimiento simple en la dirección NE.

Es por ello que el método `isValidFrom` devolverá `false` para la posición de la pieza A; y `true` para las posiciones de las piezas B y C.

Si nos centramos en la pieza B, `isValidTo` devolvería `true` solamente para la posición `to` correspondiente a la x=4 y=5; y `false` para cualquier otra posición.

Si, por ejemplo realizamos el movimiento de captura de la pieza C, el movimiento (`Move`) que se retornaría contendría las posiciones:

- `from`: posición de C, es decir, x=3 y=4
- `middle`: posición de la pieza negra capturada, x=2 y=3
- `to`: posición destino del movimiento, es decir, x=1 y=2

Condiciones de finalización de la partida

Un jugador se declara como ganador cuando:

- ha colocado una ficha en la última fila (fila 0 para el jugador de blancas, y fila 7 para el de negras)
- después de realizar su movimiento, el otro jugador no tiene ningún movimiento posible (este caso incluye cuando en dicho movimiento le han comido la última ficha que tenía)

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`:

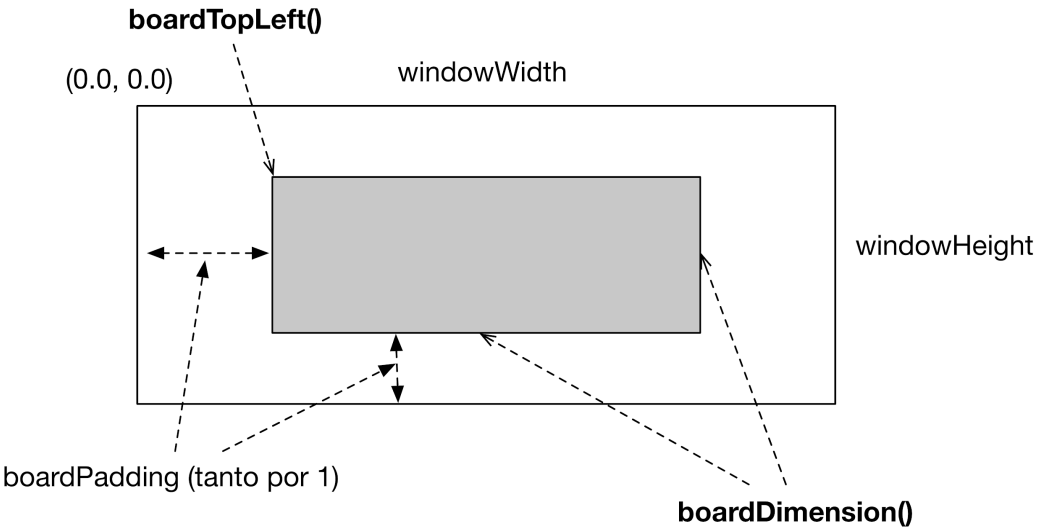
- `GPoint`: representa las coordenadas de un punto en la pantalla (con getters para la `x` y la `y`, que son `doubles`)
- `GDimension`: representa las dimensiones de un elemento en la pantalla (con getters para `height` y `width` que son `doubles`)

```
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) { ... }
  }
```

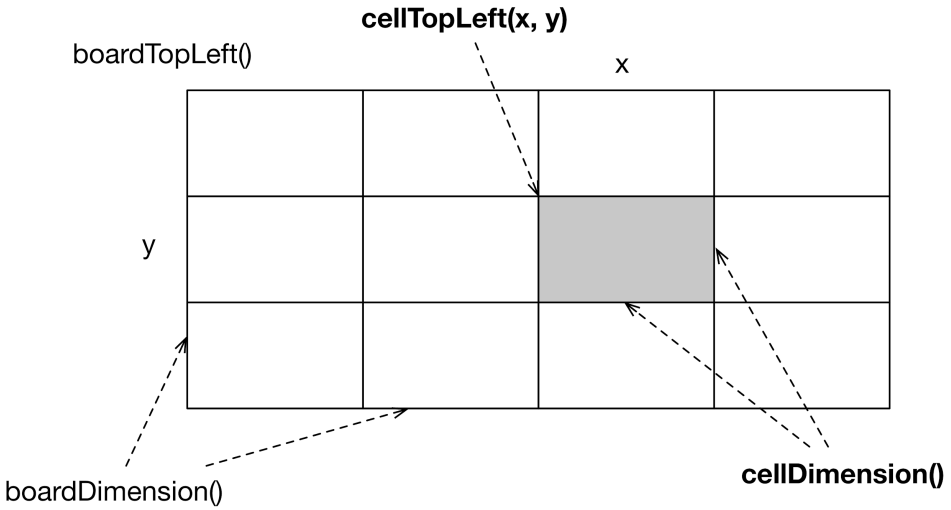
Los métodos a implementar son:

- ➔ `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

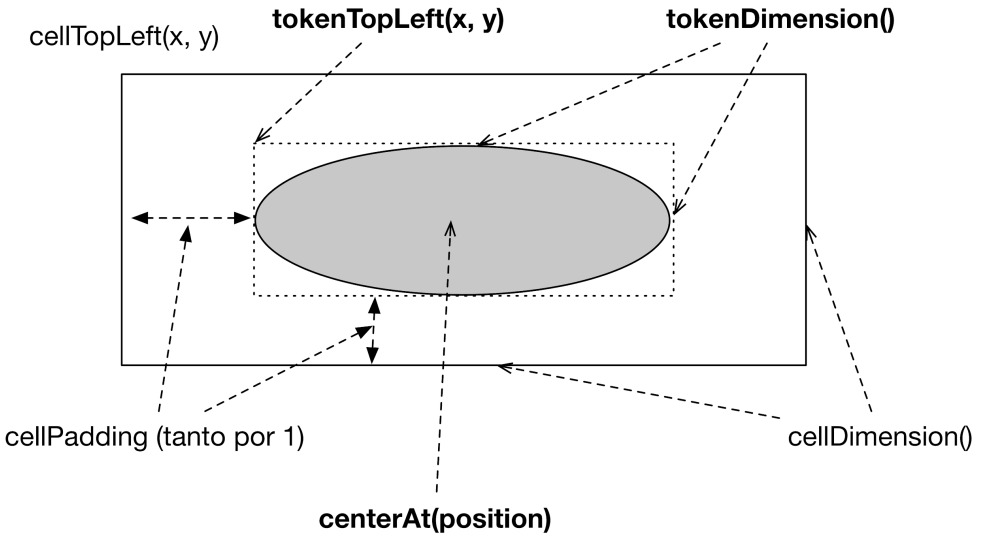
➔ `public GPoint cellTopLeft(int x, int y)`
- ◆ devuelve las coordenadas del extremo superior izquierdo de la celda que ocupa la columna x y fila y (comenzando desde 0).



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

➔ `public GPoint tokenTopLeft(int x, int y)`
- ◆ devuelve las coordenadas del extremo superior izquierdo de cada uno 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 que os preocupéis de que se trate 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`
- `paintForbidden`
- `paintPlayable`
- `paintWhite`
- `paintBlack`
- `createRect`

- `createOval`

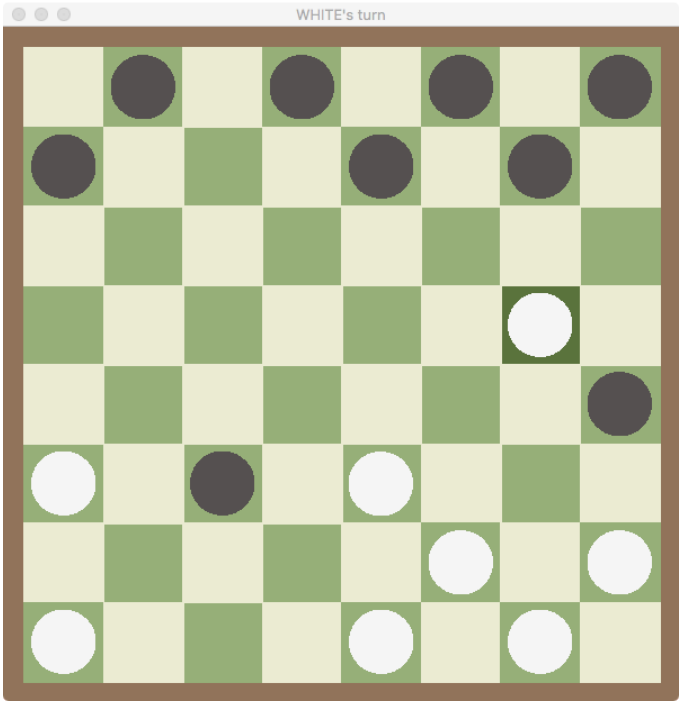
son los responsables de inicializar la interfaz gráfica en función del número de filas y columnas y de la descripción del tablero dada en un `String`. Como se indica en el código, no se comprueba que dicha información sea coherente.

Los métodos:

- `highlight`
- `unHighlight`
- `select`
- `unSelect`

son los responsables de los efectos visuales para resaltar las posiciones válidas en una situación válida y para resaltar la posición seleccionada (como origen del movimiento).

El método `clear` borra la ficha que hay en una determinada posición (la ficha que se ha capturado) y, finalmente, el método `move` es el que coordina los efectos visuales asociados a la realización de un movimiento.



La clase Checkers

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 moverse. Para facilitar visualmente 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 cuando 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 existen las variables de instancia:

- `selectedFrom`, posición seleccionada como origen de un movimiento (o `null` si no hay ninguna)
- `highlighted`, posición resaltada al pasar por encima de ella si ésta puede ser el origen (o destino, en caso de haber seleccionado una posición) válido

Para gestionar el resaltado al pasar por encima de una posición que puede ser el inicio/final de un movimiento, se usa el método `mouseMoved`.

Para seleccionar las posiciones inicial y final de un movimiento, se usa el método `mouseClicked`; y el método `showMove` actualiza el display para mostrar el movimiento seleccionado.

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. `Geometry` (1 punto)
6. `Game` (2 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 deberá tener la siguiente estructura:

- Portada (nombre, DNI o equivalente, 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 los **3 métodos que consideréis más complicados** de vuestra solución.
        - obviamente si se descomponen en métodos auxiliares, estos han de ser descritos también.
        - 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?
        - o quizás, en algún momento habéis rehecho partes ya finalizadas conforme vais avanzando en la práctica y mejorando vuestro nivel de programación en Java; comentad estos cambios.