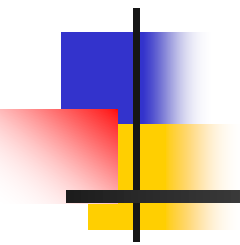
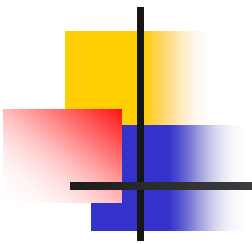


Práctica 1

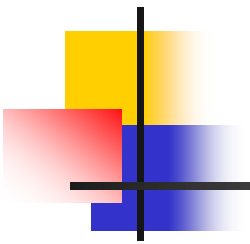


El enumerado `Direction`

- Se crea como un `enum`
- Contiene los elementos

  `UP, DOWN, LEFT, RIGHT`
- A los elementos se accede a través de la clase. Por ejemplo:

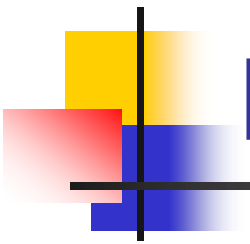
  `Direction.UP`



La clase `Position`

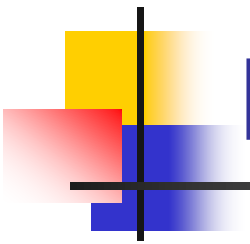
- Atributos: un entero para la fila y otro para la columna
- Métodos:
  - accesoras a los atributos
  - **`Position neighbour(dir, size)`** para conocer la posición vecina de una dada, en una dirección **`dir`** sobre un tablero de tamaño **`size`**

¿Se refiere a la posición vecina o a las posiciones ?



La clase `Cell`

- Atributos: al menos un entero para representar el valor contenido en la baldosa
- Métodos:
  - Accesoras, mutadoras
  - **`boolean doMerge(neighbour)`** fusiona, si es posible, una célula y su vecina, eliminando esta última
  - **`boolean isEmpty()`** averigua si una baldosa está vacía

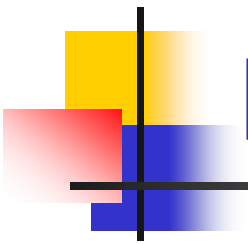


La clase `MoveResults`

- Atributos: un entero para guardar el **`Score`** y otro para el **`Highest`**; un booleano para saber si en la jugada recién hecha se ha movido alguna baldosa
- Métodos: accesoras

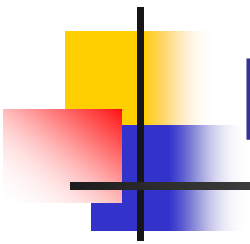
La clase Board

- Atributos: al menos una matriz **`Cell[][]`** y su tamaño
- Constructora: crea la matriz y después **crea una célula en cada una de sus componentes**
- Métodos:
  - **`MoveResults executeMove(dir)`**: realiza una jugada en la dirección **`dir`**, desplazando y fusionando. Es el método principal. [Piensa cómo se podrían simplificar los cuatro posibles movimientos y dejarlos solo en dos, incluso en uno. Estructura el código, definiendo métodos auxiliares privados]
  - Si quieres, define una clase (auxiliar) **`ArrayAsList`** para tratar una fila de baldosas como si fuera una lista y poder mover y fusionar sus elementos más fácilmente
  - **`String toString()`**: usando la clase **`MyStringUtils`**, genera la sucesión de caracteres que, impresa, muestra la situación del tablero



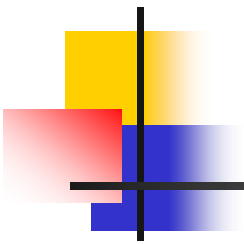
La clase Game

- Atributos: al menos, un objeto de la clase **`Board`**, un tamaño de tablero y un número inicial de baldosas (y, opcionalmente, una semilla) suministrados por **`main`**, más un `score` y un `highest`, y un booleano para indicar si el juego ha terminado
- Constructora: crea el objeto de la clase **`Board`** y le añade las baldosas iniciales
- Métodos:
  - **`void move(dir)`**: le dice al tablero que haga un movimiento en la dirección **`dir`**, actualiza los resultados tras hacer el movimiento y averigua si la partida ha acabado
  - **`void reset()`**: reinicializa el juego para empezar una nueva partida sobre un tablero con las características (tamaño y baldosas iniciales) del anterior



La clase `Controller`

- Atributos: al menos, un objeto de la clase **`Game`** y uno de la clase **`Scanner`** para poder leer las órdenes del usuario
- Métodos:
  - **`void run()`**: mientras el juego no se acabe, pide una orden al usuario, la analiza sintácticamente y la ejecuta. Para el análisis utiliza los métodos **`trim()`** y **`split()`** de la clase
- Solo se crea un objeto de cada una de las clases **`Controller`**, **`Game`** y **`Board`**



La clase Game2048

- Contiene el método **`main()`**
- Crea un controlador (que crea un juego, que crea un tablero) con el que invoca **`run()`**
- Lee el tamaño de un tablero con la instrucción

```
int dim = Integer.parseInt(args[0]);
```

- Análogamente lee el número de baldosas iniciales de **`args[1]`** y, opcionalmente, la semilla de **`args[2]`**

Comentarios finales

- Paquete único **`tp.pr1`** con posibilidad de hacer subpaquetes
- Para generar números aleatorios, se genera o se lee una semilla

```
long seed = new Random().nextInt(1000);
long seed = Long.parseLong(args[2]);
```

  y, a partir de entonces, se generan números aleatorios con:

```
Random rand = new Random(seed);
```

- Genera de la forma más aleatoria posible las baldosas iniciales, sus valores y la baldosa nueva tras cada jugada que suponga un movimiento o fusión de baldosas