MÓDULO INICIAL. PROGRAMACIÓN ORIENTADA A OBJETOS CON JAVA

Relación de Problemas Nº 1

Proyecto Jarras (clases, composición)(*mandatory*)

El objetivo de este ejercicio es crear una clase `Jarra` que utilizaremos para "simular" algunas de las acciones que podemos realizar con una jarra. Se creará en el paquete `jarras`.

Nuestras jarras van a poder contener cierta cantidad de agua. Así, cada jarra tiene una determinada capacidad (en litros) que será la misma durante la vida de la jarra (dada en el constructor). En un momento determinado, una jarra dispondrá de una cantidad de agua que podrá variar en el tiempo. Las acciones que podremos realizar sobre una jarra son:

- Llenar la jarra por completo desde un grifo.
- Vaciarla enteramente.
- Llenarla con el agua que contiene otra jarra (bien hasta que la jarra receptora quede colmada o hasta que la jarra que volcamos se vacíe por completo).

Por ejemplo: Disponemos de dos jarras A y B de capacidades 7 y 4 litros respectivamente. Podemos llenar la jarra A (no podemos echar menos del total de la jarra porque no sabríamos a ciencia cierta cuánta agua tendría). Luego volcar A sobre B (no cabe todo por lo que en A quedan 3 litros y B está llena). Ahora vaciar B. Después volver a volcar A sobre B. En esta situación, A está vacía y B tiene 3 litros.

- Hay que construir la clase `Jarra` con los métodos necesarios para realizar las operaciones que acabamos de describir. Además de dichas operaciones necesitamos métodos para consultar tanto la cantidad de agua que tiene una jarra como su capacidad. Definir el método `public String toString()` que devuelva un `String` que represente los datos de la jarra (oonsultar el diagrama UML para ver los nombres de los métodos).
- Para probar nuestra nueva clase vamos a construir una aplicación que cree dos jarras, una con capacidad para 5 litros y otra para 7. Una vez creadas hemos de realizar las operaciones necesarias para dejar en una de las jarras exactamente un litro de agua.

- Crear la clase **`Mesa`** que dispondrá de dos jarras (**`jarraIz`** y **`jarraDr`**). Define el tipo enumerado `Posicion` con dos posibles valores, `Izquierda` y `Derecha` como clase anidada a `Mesa` con visibilidad pública. Se pide:
    a. Un constructor que cree una mesa con dos jarras y otro que tendrá dos argumentos que serán las capacidades iniciales de las jarras izquierda y derecha.
    b. `capacidad(Posicion):int`
    Devuelve la *capacidad* de la jarra especificada como parámetro.

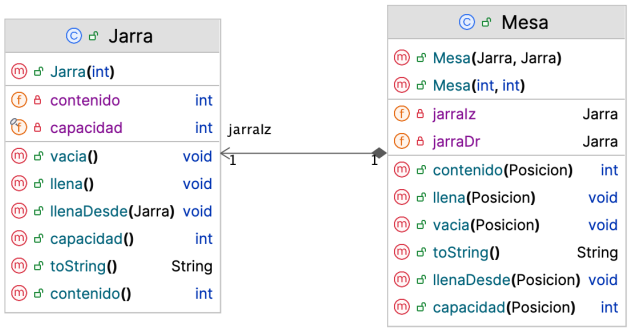
    `c. contenido(Posicion):int`
    Devuelve el *contenido* actual de la jarra especificada.

    `d. llena(Posicion):void`
    Llena el *contenido* de la jarra especificada.

    e. `vacia(Posicion):void`
    Vacía el *contenido* de la jarra especificada.

    `f. llenaDesde(Posicion):void`
    Llena el contenido de la jarra receptora con el contenido de la jarra emisora, especificada como parámetro, bien hasta que la jarra receptora quede llena o hasta que la jarra emisora se vacíe por completo.

    `g. toString():String  // @Redefinición`
    Devuelve un `String` con la representación textual del objeto mesa en el formato `M(J(cap,cnt),J(cap,cnt))`.



- Crear una aplicación que cree una mesa con valores iniciales de las jarras de 7 y 5 litros y realice las operaciones necesarias para que en una de las jarras quede 1 litro.

Proyecto Estadistica (clases) (labs)

Crear una clase `Estadistica` (en el paquete `estadistica`) que simplifique el trabajo de calcular medias y desviaciones típicas de una serie de valores. La clase incluirá tres variables de instancia, una para mantener el número de elementos de la serie (`numElementos`), otra para su suma (`sumaX`) y otra para la suma de los cuadrados (`sumaX2`).

La clase dispone de dos métodos para agregar datos a la serie, `public void agrega(double d)` que agrega el dato `d` a la serie (incrementa `numElementos` en uno, incrementa `sumaX` en `d` e incrementa `sumaX2` en d^2) y `public void agrega(double d, int n)` que agrega `n` veces el dato `d` a la serie (incrementa `numElementos` en `n`, incrementa `sumaX` en `n*d` e incrementa `sumaX2` en $n*d^2$).

Para consultar los valores estadísticos disponemos de tres métodos, `public double media()` que devuelve la media de los valores (`sumaX/numElementos`), `public double varianza()` que devuelve la varianza (`sumX2/numElementos – media()`^{2}). Y `public double desviacionTipica()` que devuelve la raíz de la varianza.

La clase `EjemploUso` muestra cómo se usa esta clase. Se calcula la media y desviación típica de una serie de 100000 valores que se distribuyen según una Normal(0,1). Es de esperar pues que la media esté cercana a 0 y la desviación típica a 1.

```
import java.util.Random;
import estadistica.Estadistica;

public class EjemploUso {
   public static void main(String [] args) {
      Estadistica est = new Estadistica();
      Random ran = new Random();
      for (int i = 0; i < 100000 ; i++) {
         est.agrega(ran.nextGaussian());
      }
      System.out.println("Media = "+est.media());
      System.out.println("Desviacion tipica = "+est.desviacionTipica());
   }
}
```

Proyecto NPIV1 (clases) (labs)

Se pretende crear un simulador de calculadora que opera con la Notación Polaca Inversa (NPI). Esta notación se caracteriza por no usar paréntesis para describir expresiones aritméticas. Así, la expresión

```
3*(6-4)+5
```

se escribe en NPI de la siguiente forma:

```
3 6 4 - * 5 +
```

La forma de operar de estas calculadoras es la siguiente.

Cada calculadora dispone de cuatro registros llamados x, y, z, t. Al calcular una expresión en NPI se realizan las siguientes operaciones:

3	x = 3, y = 0, z = 0, t = 0	// t = z, z = y, y = x, x = 3
6	x = 6, y = 3, z = 0, t = 0	// t = z, z = y, y = x, x = 6
4	x = 4, y = 6, z = 3, t = 0	// t = z, z = y, y = x, x = 4
-	x = 2, y = 3, z = 0, t = 0	// x = y – x, y = z, z = t
*	x = 6, y = 0, z = 0, t = 0	// x = y * x, y = z, z = t
5	x = 5, y = 6, z = 0, t = 0	// t = z, z = y, y = x, x = 5
+	x = 11, y =0, z = 0, t = 0	// x = y + x, y = z, z = t

y en la variable x obtenemos el resultado de la expresión.

Crear la clase `NPI` (en el paquete `npi`) que mantenga cuatro variables `x`, `y`, `z`, `t` con el siguiente comportamiento:

- El constructor por defecto.
- El método `public void entra(double valor)` que simule la entrada de un valor.
- El método `public void suma()` que simule la entrada de una suma.
- El método `public void resta()` que simule la entrada de una resta
- El método `public void multiplica()` que simule la entrada de una multiplicación.
- El método `public void divide()` que simula la entrada de una división.
- El método `public double getResultado()` que devuelve el valor de la variable x.
- Definir una representación para los objetos de esta clase de la forma

  NPI(x =…, y = …, z = …, t = …)

Se proporciona la aplicación `Main` que calcula el valor de la expresión

```
3 * (6 - 2) + 5
```

que Polaca Inversa es

```
3 6 2 - * 5 +
```

Crear otro programa para calcular la expresión

```
3 * (6 - 2) + (2 + 7) / 5
```

Proyecto RelojArena (composición)

En esta práctica vamos a simular el comportamiento de un reloj de arena. Crearemos la clase `RelojArena` y `MedidorTiempo` en el paquete `reloj`.

Un reloj de arena se crea con una cantidad determinada de arena. La medida de la cantidad de arena se hace por tiempo. Por ejemplo, podemos crear un reloj con arena para medir 7 minutos.

En un instante dado, un reloj puede tener 3 minutos en la parte superior y 4 minutos transcurridos (en la parte inferior). Es imposible saber el tiempo que le queda a un reloj de arena hasta que la parte superior se vacíe. Solo podemos medir el tiempo transcurrido cuando toda la arena se encuentre en la parte inferior.

El estado de un reloj lo vamos a caracterizar por dos enteros, los minutos que hay en la parte superior y los minutos que hay en la parte inferior.

Las operaciones que vamos a disponer en el reloj son:

Un constructor que crea el reloj con una cantidad de minutos. En el constructor, todos los minutos están en la parte inferior.

- El método `public void gira()` que intercambia los minutos de las partes superior e inferior.
- El método `public void pasatiempo()` que hace que todos los minutos pasen a la parte inferior.
- El método `public int getTiempoRestante()` que nos dice el tiempo que le queda a este reloj para que toda la arena este en la parte inferior.
- El método `public void pasatiempo(RelojArena reloj)`. Este es el método más interesante pues permite medir tiempos. Simula que pasa el tiempo del reloj que se pasa como argumento. Así, si al receptor le quedan 7 minutos y al reloj argumento le quedan 3 minutos, el resultado del método es que el receptor le quedan 4 minutos y el reloj argumento no le queda nada. Pero si al receptor le quedan 4 minutos y al reloj argumento le quedan 6 minutos, entonces, correrán los 6 minutos y a los dos relojes no le quedarán nada de tiempo.
- Un método para representar un reloj en la forma `R(Arriba/Abajo)`.

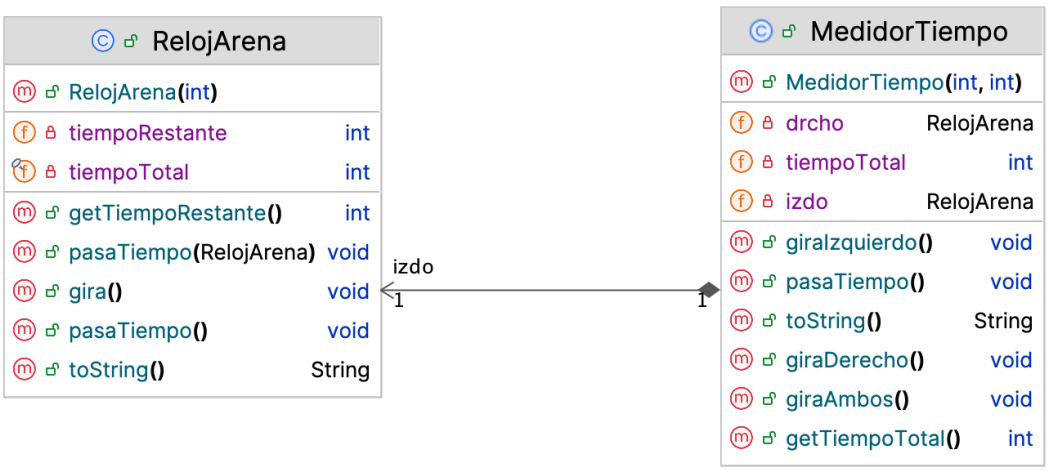
Pongamos un ejemplo de uso de los relojes. Supongamos que tenemos dos relojes de arena de 7 y 5 minutos y vamos a medir 9 minutos. La clase `Main` proporciona este ejemplo.

```
RelojArena r1 = new RelojArena(7);
RelojArena r2 = new RelojArena(5);
r1.gira();            // r1=R(7/0)
r2.gira();            // r2=R(5/0)
r1.pasaTiempo(r2);    // r1=R(2/5) r2=R(0/5). Tiempo = 5
r2.gira();            // r2=R(5/0)
r2.pasaTiempo(r1);    // r2=R(3/2) r1=(0/7). Tiempo = 5+2
r2.gira();            // r2=R(2/3)
r2.pasaTiempo();      // r2=R(0/5) Tiempo = 5+2+2
```

Como lo normal es manejar los relojes de dos en dos para poder medir tiempos, vamos a crear la clase `MedidorTiempo` cuyo estado viene caracterizado por dos relojes de arena y un tiempo total. El comportamiento de la clase es el siguiente:

- En el constructor se le pasa el tiempo con el que se crearán cada uno de los relojes de arena. El tiempo total inicial será 0. A un reloj le llamamos `izdo` y al otro `drcho`.
- El método `public void giraIzquierdo()` que gira el reloj de la izquierda. Este método llamará al método `pasatiempo()` descrito más abajo.
- El método `public void giraDerecho()` que gira el reloj derecho. Este método llamará al método `pasatiempo()` descrito más abajo.
- El método `public void giraAmbos()` que gira ambos relojes. Este método llamará al método `pasatiempo()` descrito más abajo.
- El método `public void pasaTiempo()` que se comporta de la siguiente manera:
    - Si uno de los relojes tiene toda la arena en la parte inferior, se hace pasar el tiempo del otro reloj y se incremente el tiempo total.
    - Si ninguno está vacío, se toma el que menor tiempo le reste. Si es el derecho se hace `izdo.pasaTiempo(drcho)` y se incrementa el tiempo total en el tiempo transcurrido. En caso contrario se hace `drcho.pasaTiempo(izdo)` y también se incrementa en el tiempo transcurrido.
- El método `public int getTiempoTotal()` que devuelve el tiempo transcurrido desde que se creo el medidor.
- Un método para visualizar un medidor de tiempos de manera que se vean los dos relojes y el tiempo total.

Crear un programa principal y un Medidor de tiempos que mida 15 minutos a partir de dos relojes que miden 7 y 5 minutos.



Proyecto Rectas (record, composición, excepciones) (labs)

En esta práctica implementaremos clases que manipulan puntos, vectores y rectas del plano (serán las clases `Punto`, `Vector` y `Recta` en el paquete `rectas`).

a) La clase `Punto` se implementará como un *record*. Su constructor principal tomará como argumentos la abscisa y ordenada del punto. Crea también un segundo constructor sin argumentos que crea un punto en el origen de coordenadas. Crea además un método, `double distancia(Punto p)` que calcula la distancia entre el receptor y el punto `p`.

b) En cuanto a la clase `Vector`, también se implementará como un *record*. Suponemos que almacena el representante del vector con origen en el origen de coordenadas, por lo que, bastará con conocer (y almacenar) su extremo. Esta clase tendrá tres *constructores*: el principal creará un vector conociendo el punto extremo; otro constructor creará un vector conocidas sus dos componentes; y el último lo creará conociendo un punto origen y un punto extremo (en este caso se realizarán los cálculos necesarios para almacenar únicamente el extremo del vector equivalente con origen en el origen de coordenadas).

Un vector *ortogonal* (perpendicular) al vector `(x,y)` es el vector `(-y,x)` (Éste está girado 90 grados en sentido contrario a las agujas del reloj con respecto al anterior). Dos vectores `(vx,vy)` y `(ux,uy)` son *paralelos* si verifican `vx*uy == vy*ux`. (La igualdad entre valores doubles es muy complicada por lo que podemos suponer que son iguales si difieren en menos de un *epsilon* (define una constante privada EPSILON=0.000001)). Dos vectores paralelos tienen la misma dirección (aunque pueden tener diferente sentido). El método `public Punto extremoDesde(Punto org)` devuelve el punto donde quedaría el extremo del vector si el origen se colocara en `org`. El método `public double modulo()` devuelve el proyecto del recetor. El método `public Vector dirección()` devuelve un vector unitario con la misma dirección y sentido que el receptor (lanza una excepción `RuntimeException` si el proyecto es cero. *Ver nota sobre situaciones excepcionales*). El diagrama muestra el resto de los métodos a implementar en este record. Su significado se deduce fácilmente.

c) Para construir la clase `Recta` que también se implementará como un *record*. Se tendrá en cuenta que una recta queda determinada por un vector que marque su dirección (*vector director*) y un punto por donde pase. Para esta clase se proporcionarán dos constructores: el principal que genere la recta conociendo vector director y un punto por donde pasa; y otro que genere la recta conociendo dos puntos por donde pasa.

d) Dos rectas son *paralelas* si sus vectores de dirección son paralelos. Una recta *pasa por* un punto `p` si el vector formado por `p` y un punto de la recta es paralelo al vector director de la recta.

e) La ecuación implícita de una recta con vector director `(vx,vy)` y que pasa por el punto `(px,py)` es `ax+by+c=0` y se obtiene desarrollando la ecuación continua de la misma $\frac{x-px}{vx} = \frac{y-py}{vy}$. Define el método `implicita()` que devuelve un record `Implicita` con los coeficientes `a`, `b` y `c` de la recta en forma implícita. Este record estará anidado a la clase `Recta`.

Define el método privado de clase

`double determinante(double a11, double a12, double a21, double a22)`

que dada una matriz $\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$, calcule su determinante $\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11} * a_{22} - a_{12} * a_{21}$.

Para calcular la intersección de dos rectas, la primera con vector director `(vx,vy)` y que pasa por el punto `P(px,py)` y la segunda con vector director `(ux,uy)` y que pasa por el punto `(qx,qy)` se sigue la regla de Cramer:

Calculamos la forma implícita de las dos rectas. Sean `ax+by+c=0` y `a'x+b'y+c'=0`.

Las coordenadas del punto de corte se determinan de la siguiente manera:

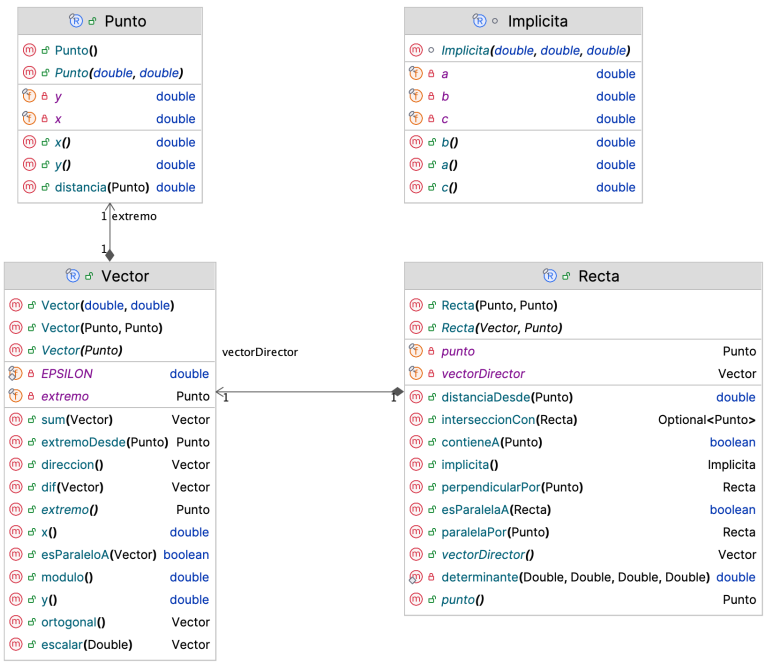
$$x = \frac{\begin{vmatrix} -c & b \\ -c' & b' \end{vmatrix}}{\begin{vmatrix} a & b \\ a' & b' \end{vmatrix}} \qquad y = \frac{\begin{vmatrix} a & -c \\ a' & -c' \end{vmatrix}}{\begin{vmatrix} a & b \\ a' & b' \end{vmatrix}}$$

Para que exista el punto de corte, se debe cumplir que el denominador no sea 0.

Define el método `Optional<Punto> interseccion(Recta r)` que calcula el punto de intersección de dos rectas. Si no existe ese punto se devuelve un opcional vacío. (*Ver la nota siguiente sobre situaciones excepcionales*)

f) El método `public Recta paralelaPor(Punto p)` devuelve una recta paralela a la actual que pase por el punto `p` que se pasa como parámetro, es decir una recta cuyo vector director sea el mismo que el de la recta actual y que pase por `p`. El método `public Recta perpendicularPor(Punto p)` devuelve una recta perpendicular a la actual que pase por el punto `p` que se pasa como parámetro, esto es, una recta cuyo vector director sea perpendicular al actual y que pase por `p`. El método `public double distanciaDesde(Punto)`, ha de devolver la distancia entre la recta y el punto que se pasa como parámetro. Para ello se habrá de crear una recta perpendicular a la actual que pase por `p`, calcular el punto de intersección de ambas rectas, y devolver la distancia desde este punto a `p`. Ver el diagrama para completar los métodos de esta clase.

El siguiente diagrama muestra las clases y métodos que hay que definir y sus relaciones:



Crea la aplicación `EjRectas` que calcule el área de un triángulo conociendo los tres puntos del plano que lo delimitan y luego la intersección de dos rectas.

Nota sobre el tratamiento de situaciones excepcionales.

Debemos ser sistemáticos a la hora de tratar situaciones excepcionales, y siempre debemos evitar efectos laterales. Entendemos por efecto lateral cualquier acción que no tiene nada que ver con la funcionalidad de un método; por ejemplo, un método para calcular el punto de intersección de dos rectas no debe imprimir nada en la pantalla, aunque las rectas sean paralelas, o modificar el valor de una variable de instancia, por ejemplo, su vector director. Debemos tener en cuenta que no sabemos en qué contexto se va a utilizar después esta clase.
Puede haber dos formas alternativas para tratar estas situaciones.

Tratamiento de excepciones
Una forma es el tratamiento de excepciones. Antes de estudiar el mecanismo de excepciones vamos a describirlo por si lo utilizamos:
Una excepción se utiliza cuando se produce una situación anómala. Java dispone de mecanismos para:

- "informar" de que se ha producido una situación anómala.
- "tratar" dicha situación.

Mientras no sepamos cómo tratarlas, nos conformaremos simplemente con informar de la situación anómala. Para ello debemos "lanzar" una excepción. Aunque existen distintos tipos de excepciones, en una primera aproximación, supondremos que una excepción es un objeto de la clase `RuntimeException` que el sistema trata de una forma especial.

- La clase `RuntimeException` dispone de un constructor con un argumento `String` con el que se describe el problema ocurrido.
- Cuando se produce una situación anómala, debemos crear un objeto de la clase `RuntimeException` (con `new`) y lanzarlo utilizando la instrucción `throw`.

Así, en un método donde se produce una situación excepcional, como, por ejemplo, en `interseccionCon` de la clase `Recta`, lo primero que hacemos es controlar la situación anómala lanzando la excepción si ésta se produce. Por ejemplo:

```
public Punto interseccionCon(Recta r) {
  if (this.paralelaA(r)) {
        throw new RuntimeException("Rectas paralelas");
  }
  // Aquí estamos seguros de que no son paralelas
  // ...
}
```

Como vemos, si las rectas son paralelas (situación anómala) se lanza una nueva excepción que informará de que las rectas son paralelas. Si se llega a ejecutar el `throw` (se lanza la excepción), se interrumpe la ejecución del programa en ese punto.

Igual ocurre en el caso de calcular el vector dirección de un vector dado. Si el proyecto es 0 se debe lanzar una excepción.

Usar `Optional<T>`

Otra forma de manejar situaciones excepcionales es usar la clase `Optional<T>`. Es una clase genérica (de ahí lo de `<T>`). Esto quiere decir que cuando la vamos a usar debemos indicar cual es la clase que sustituirá a la `T`. Por ejemplo en el método `interseccionCon(Recta)` podemos devolver un `Optional<Punto>`. Así, el resultado puede que sea un `Punto` o puede que no. Es como si devolviéramos una caja que puede estar vacía o contener un `Punto`.
La clase `Optional` tiene dos métodos de fábrica (de clase), `empty()` que devuelve la caja vacía y `of(Punto)` que devuelve la caja con un punto.

```
public Optional<Punto> interseccionCon(Recta r) {
  if (this.paralelaA(r)) {
       return Optional.empty();
  }
  // Aquí estamos seguros de que no son paralelas
  // ...
  return Optional.of(...);
}
```

Luego, el método que llama al método `interseccionCon` puede usar los métodos de instancia de `Optional`:

`boolean ifPresent()` para saber si hay algo en la caja o
`T get()` para extraer el contenido de la caja.

En este ejercicio hemos optado por las dos opciones.

La aplicación Tesoro (composición, paquetes)

Un mapa de un tesoro tenía las siguientes indicaciones.

En la playa de la isla Margarita hay tres palmeras, una con una marca amarilla, otra con una marca azul y una tercera con una marca rosa. Las tres palmeras permiten localizar un tesoro escondido en la arena. Para ello, deben seguirse las siguientes instrucciones:

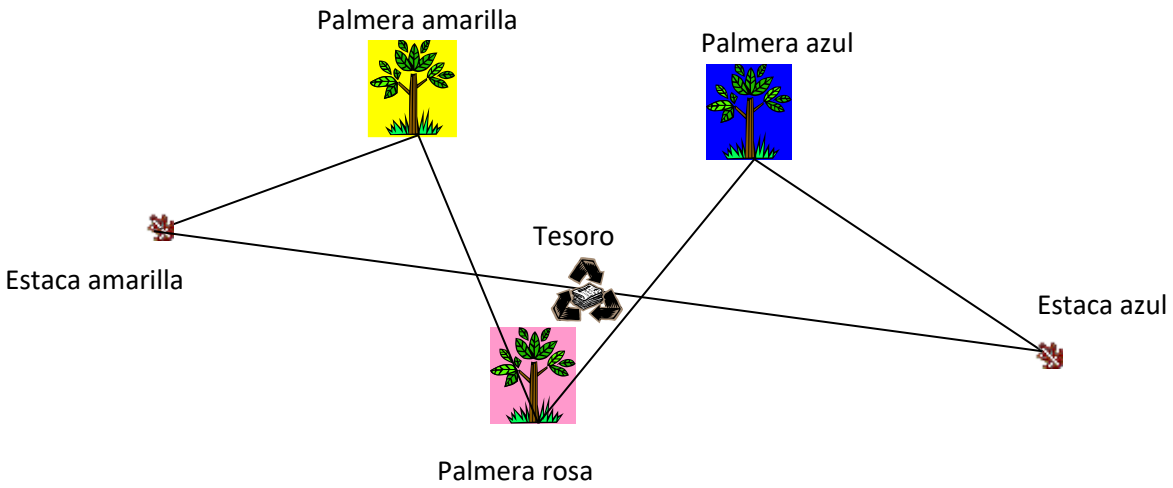
- *Desde la palmera rosa avanzar en línea recta hasta la amarilla contando el número de pasos. Una vez en la palmera amarilla, girar 90 grados en sentido contrario a las agujas del reloj y avanzar en línea recta el mismo número de pasos antes contado. Clavar una estaca (la llamaremos estaca amarilla) en el lugar alcanzado.*
- *Volver a la palmera rosa y repetir el procedimiento caminando ahora hacia la palmera azul pero girando los 90 grados en sentido de las agujas del reloj. Clavar también una estaca (la llamaremos estaca azul) en el lugar alcanzado.*
- *El tesoro se encuentra en la mitad del camino entre la estaca amarilla y la azul.*

- Crea una clase Tesoro a la que le añadiremos un método de clase

  ```
  Punto posicionTesoro(Punto pAm, Punto pAz, Punto pRo)
  ```

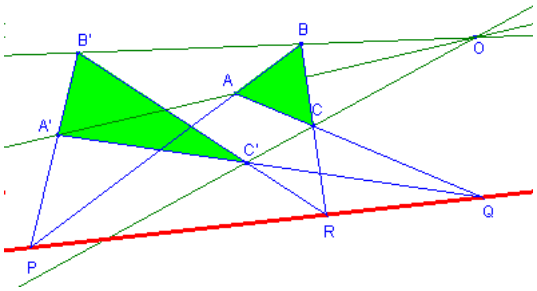
  Este método devuelve la posición del tesoro conocida las posiciones de las tres palmeras.
- Crea en esta misma clase una aplicación a la que le proporcionemos seis argumentos, los dos primeros generan la posición en el plano de coordenadas de la palmera amarilla, los dos siguientes la posición de la palmera azul y los dos últimos la de la palmera rosa. El programa debe localizar y mostrar la posición en la que se encuentra el tesoro.
- El pirata griego Nizor Rademates fue a la isla margarita con el mapa del tesoro y localizó la posición de las palmeras con marca amarilla y azul pero no encontró ni rastro de la palmera rosa. Maldiciendo su mala suerte abandonó la isla sin el tesoro.

  Para comprobar que si el pirata hubiera sabido algo de matemáticas se hubiera llevado el tesoro, crea una clase `Pirata` y en ella una aplicación que trabaja sobre un cuadrado del plano de -300 a 300 en el eje x y en el eje y. Coloca donde quieras la palmera amarilla y la azul en ese cuadrado. Ahora, realiza 100 simulaciones; en cada una, coloca aleatoriamente la palmera rosa en el cuadrado y calcula donde quedaría el tesoro; muestra las tres palmeras y la posición del tesoro en una misma línea (usar el formato `%25s` para mostrar cada punto).



La aplicación Desargues

Una prueba más completa para el paquete rectas sería comprobar que es cierto el teorema de Desargues:



Dado un punto O cualquiera, se trazan tres rectas que pasen por O. En estas rectas se eligen tres puntos A, B y C de manera que formen un triángulo. Se vuelven a tomar otros tres puntos A', B' y C' sobre cada recta que formen otro triangulo de manera que los triángulos no tengan lados paralelos.

Sea P el punto de intersección de la recta que pasa por A y B y la recta que pasa por A' y B'.
Sea Q el punto de intersección de la recta que pasa por A y C y la recta que pasa por A' y C'.
Sea R el punto de intersección de la recta que pasa por B y C y la recta que pasa por B' y C'.

Entonces, los puntos P, Q y R siempre están alineados.

Crea una aplicación, `Desargues`, que pruebe este teorema, conocidos los puntos O, A, B, C, A', B' y C'. Tomar puntos cualesquiera, pero deben construirse de manera que verifiquen la hipótesis del teorema. En caso contrario indicarlo con un mensaje.

Proyecto UniversoV1 (otro proyecto, composición, arrays, paquetes)

Se pretende crear una aplicación que maneje partículas en el universo. Las partículas estarán localizadas en una posición y tendrán una masa, radio y velocidad. A partir de la fuerza de atracción con otras estrellas la fuerza que actúa sobre la estrella variará y por tanto su velocidad. El universo estará compuesto de muchas partículas que se atraerán y generarán el movimiento de estas. Las clases `Particula` y `Universo` se crearán en el paquete universo.

Para esta aplicación necesitaremos las clases `Punto` y `Vector` del proyecto Rectas. Debemos publicar el proyecto `Rectas` e incluirla su referencia en el `pom.xml` de este proyecto.

Clase Particula

- La clase `Particula` tiene una posición (`Punto`), una velocidad (`Vector`), una masa (constante de tipo `double`) y un radio (constante de tipo `doble`).
- Define un constructor donde se le pase la información de todas las variables de instancia en el orden en el que las hemos definido.
- Define los métodos `Punto posicion()` que devuelva la posición y `double radio()` que devuelva el radio.
- Define el método `Vector fuerzaDesde(Particula part)` que calcule la fuerza de atracción que ejerce la partícula `part` sobre la receptora. Para ello, usa la constante de gravitación universal (`G = 6.67E-11`). La fuerza que ejerce la partícula p_2 sobre p_1 se calcula por medio de:

  $$F_{21} = G\frac{m_1 m_2}{\|r_2 - r_1\|^2}\hat{u}_{12}$$

  Donde $\hat{u}_{12}$ se refiere al vector unitario que va desde p_1 a p_2, r_1 y r_2 son los radios de p_1 y p_2 y m_1 y m_2 son las masas de p_1 y p_2.
- Define el método `void mueve(Vector vz, double dt)` que simule que se aplica una fuerza `vz` a la partícula durante un tiempo `dt`. Como sabemos por Física (en mayúsculas están vectores y puntos)

  $$Fuerza = masa \times Aceleracion$$

  $$VelFinal = VelInicial + Aceleracion \times tiempo$$

  $$PosFinal = PosInicial + Velocidad \times tiempo$$

  - Conociendo la masa y la fuerza (`vz`), podemos calcular el vector aceleración.
  - Con la velocidad actual, la aceleración calculada y el incremento de tiempo (`dt`) podemos calcular la nueva velocidad de la partícula.
  - Con la posición actual, la nueva velocidad de la partícula y el incremento de tiempo (`dt`) podemos calcular la nueva posición de la partícula.
- Define el método `String toString()` para visualizar todos los atributos de una partícula.

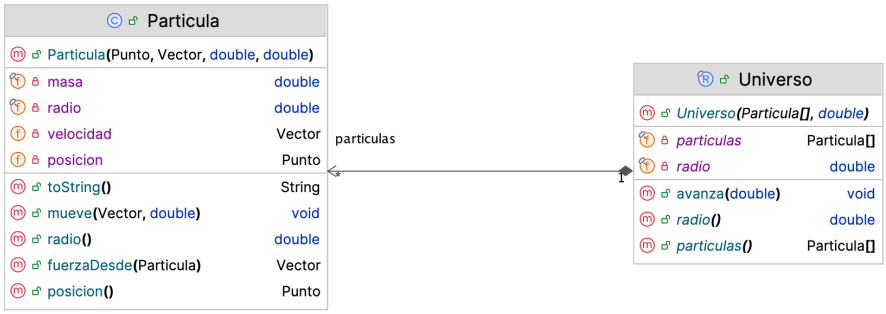
Clase Universo

Un universo contiene un array de partículas y un radio que será el espacio visible del universo.

- Define un constructor en el que se pase el array de partículas y el radio del universo.
- Define un método `void avanza(double dt)` que simule como se modifican las posiciones y velocidades de las partículas transcurrido un tiempo dt.

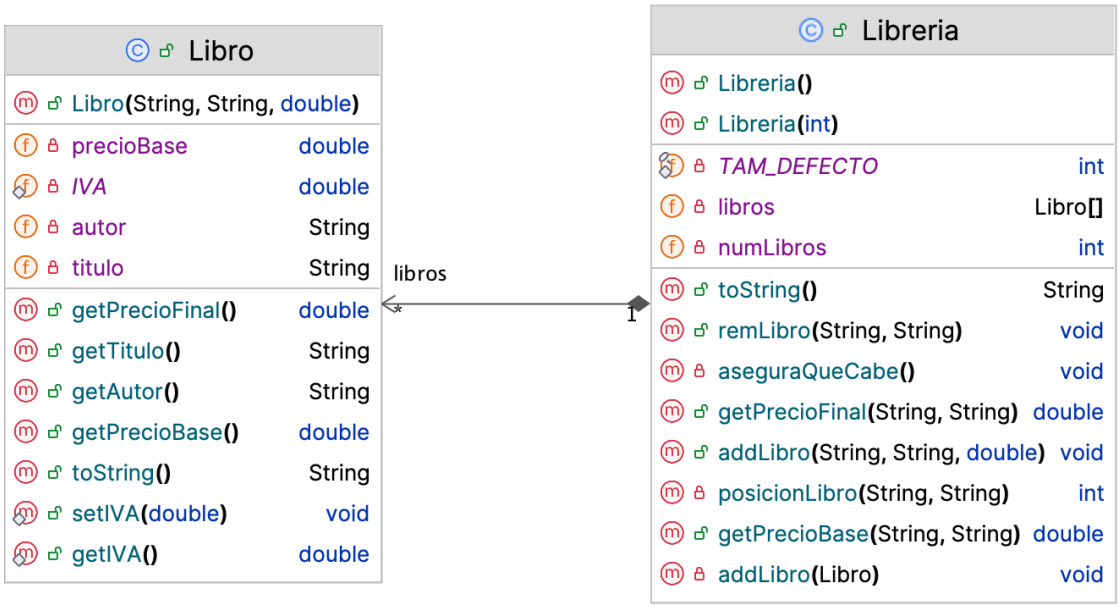
Este método calcula, para cada partícula, la fuerza de atracción total del resto de partículas. Para ello, crea un array de vectores de la misma dimensión que el de partículas y va sumando a cada posición i de ese array la fuerza de atracción de cada partícula j con i siendo j distinto de i. Finalmente, mueve cada partícula con la fuerza que actúa sobre ella y el tiempo que se aplica.

Se proporciona un paquete `gui` que incluye la clase `PanelUniverso` que permite visualizar un universo (incluirlo en el proyecto). También se proporciona la clase `Main` que contiene cuatro ejemplos que pueden probarse de forma independiente. Deben ejecutarse quitando el comentario en el ejemplo en el método `main`. Falta crear el paquete universo y en él las clases `Particula` y `Universo`.



Proyecto LibreriaV0 (composición, static, arrays, paquetes)

Se pretende crear clases que mantengan información sobre libros. Para ello, se crearán las clases `Libro` (del paquete `libreria`) y la clase `Libreria` (en el mismo paquete).



Nota: se pueden añadir a las siguientes clases los métodos **privados** que se consideren necesarios.

La clase `Libro`

La clase `Libro` (del paquete `libreria`) contiene información sobre un determinado libro, tal como el nombre del autor, el título, y el precio base. Además, también posee información sobre el porcentaje de IVA que se aplica para calcular su precio final. Nótese que el porcentaje de IVA a aplicar es el mismo y es compartido por todos los libros, siendo su valor inicial el 10 %.

- `Libro(String,String,double)`

  Construye un objeto `Libro`. Recibe como parámetros, en el siguiente orden, el nombre del autor, el título, y el precio base del libro.

- `getAutor():String`
- `getTitulo():String`
- `getPrecioBase():double`

  Devuelven los valores correspondientes almacenados en el objeto.

- `getPrecioFinal():double`

  Devuelve el precio final del libro, incluyendo el IVA, según la siguiente ecuación.

$$PF = PB + PB \cdot IVA / 100$$

- `toString(): String // [@Redefinición]`

  Devuelve la representación textual del objeto, según el formato del siguiente ejemplo:

(Isaac Asimov; La Fundación; 7.30; 10%; 8.03)

- `getIVA():double // [@MétodoDeClase]`

  Devuelve el porcentaje del IVA asociado a la clase `Libro`.

- `setIVA(double):void // [@MétodoDeClase]`

  Actualiza el valor del porcentaje del IVA asociado a la clase `Libro` al valor recibido como parámetro.

La clase `Libreria`

La clase `Libreria` (del paquete `libreria`) almacena múltiples instancias de la clase `Libro` en un array, así como el número total de libros que contiene almacenados. Además, también contiene una constante de clase que especifica la capacidad inicial por defecto del array (16).

Nota 1: las comparaciones que se realicen tanto del nombre del autor como del título del libro se deberán realizar sin considerar el caso de las letras que lo componen.

Nota 2: se recomienda la definición de métodos privados que simplifiquen y permitan modularizar la solución de métodos complejos.

- `Libreria()`

  Construye un objeto `Librería` vacío (sin libros) con un array con una capacidad inicial de tamaño 16.

- `Libreria(int)`

  Construye un objeto `Librería` vacío (sin libros) con un array con una capacidad inicial del tamaño recibido como parámetro,

- `addLibro(String,String,double): void`

  Crea un nuevo objeto `Libro` con el nombre del autor, el título, y el precio base recibidos como parámetros. Si ya existe un libro de ese mismo autor, con el mismo título, entonces se reemplaza el libro anterior por el nuevo. En otro caso, añade el nuevo libro a la librería, considerando que si el array está lleno se debe duplicar su capacidad. Así mismo, se debe actualizar adecuadamente el valor de la cuenta del número de libros.

- `remLibro(String,String): void`

  Si existe el libro correspondiente al autor y título recibidos como parámetros, entonces elimina el libro de la librería, manteniendo el mismo orden relativo de los libros almacenados. Así mismo, se debe actualizar adecuadamente el valor de la cuenta del número de libros. Si no existe no se hace nada.

- `getPrecioBase(String,String): double`

  Devuelve el precio base del libro correspondiente al autor y título recibidos como parámetros. Si el libro no existe en la librería, entonces devuelve cero.

- `getPrecioFinal(String,String): double`

Devuelve el precio final del libro correspondiente al autor y título especificados. Si el libro no existe en la librería, entonces devuelve cero.

▸ `toString(): String // [@Redefinición]`

Devuelve la representación textual del objeto, según el formato del siguiente ejemplo: (sin considerar los saltos de línea)

```
[(George Orwell; 1984; 6.20; 10%; 6.82),
 (Philip K. Dick; ¿Sueñan los androides con ovejas eléctricas?; 3.50; 10%;
 3.85), (Isaac Asimov; Fundación e Imperio; 9.40; 10%; 10.34),
 (Ray Bradbury; Fahrenheit 451; 7.40; 10%; 8.14),
 (Alex Huxley; Un Mundo Feliz; 6.50; 10%; 7.15),
 (Isaac Asimov; La Fundación; 7.30; 10%; 8.03),
 (William Gibson; Neuromante; 8.30; 10%; 9.13),
 (Isaac Asimov; Segunda Fundación; 8.10; 10%; 9.81),
 (Isaac Newton; Arithmetica Universalis; 10.50; 10%; 11.55)]
```

Desarrolle una aplicación `PruebaLibreria` (en el paquete anónimo) que permita realizar una prueba de las clases anteriores. Así, deberá añadir a la librería los siguientes libros:

```
("george orwell", "1984", 8.20)
("Philip K. Dick", "¿Sueñan los androides con ovejas eléctricas?", 3.50)
("Isaac Asimov", "Fundación e Imperio", 9.40)
("Ray Bradbury","Fahrenheit 451",7.40)
("Alex Huxley", "Un Mundo Feliz",6.50)
("Isaac Asimov", "La Fundación",7.30),
("William Gibson", "Neuromante", 8.30)
("Isaac Asimov","SegundaFundación",8.10)
("Isaac Newton", "arithmetica universalis", 7.50)
("George Orwell", "1984", 6.20)
("Isaac Newton", "Arithmetica Universalis", 10.50)
```

De tal forma que al mostrar la representación textual de la librería mostrará (sin considerar los saltos de línea):

```
[(George Orwell; 1984; 6.20; 10%; 6.82),
(Philip K. Dick; ¿Sueñan los androides con ovejas eléctricas?; 3.50; 10%; 3.85),
(Isaac Asimov; Fundación e Imperio; 9.40; 10%; 10.34),
(Ray Bradbury; Fahrenheit 451; 7.40; 10%; 8.14),
(Alex Huxley; Un Mundo Feliz; 6.50; 10%; 7.15),
(Isaac Asimov; La Fundación; 7.30; 10%; 8.03),
(William Gibson; Neuromante; 8.30; 10%; 9.13),
(Isaac Asimov; Segunda Fundación; 8.10; 10%; 9.81),
(Isaac Newton; Arithmetica Universalis; 10.50; 10%; 11.55)]
```

A continuación, se eliminarán los siguientes libros:

```
("George Orwell", "1984")
("Alex Huxley", "Un Mundo Feliz")
("Isaac Newton", "Arithmetica Universalis")
```

De tal forma que al mostrar la representación textual de la librería mostrará (sin considerar los saltos de línea):

```
[(Philip K. Dick; ¿Sueñan los androides con ovejas eléctricas?; 3.50; 10%;3.85),
(Isaac Asimov; Fundación e Imperio; 9.40; 10%; 10.34),
(Ray Bradbury; Fahrenheit 451; 7.40; 10%; 8.14),
(Isaac Asimov; La Fundación; 7.30; 10%; 8.03),
(William Gibson; Neuromante; 8.30; 10%; 9.13),
(Isaac Asimov; Segunda Fundación; 8.10; 10%; 9.81)]
```

Finalmente se mostrará el precio final de los siguientes libros:

```
getPrecioFinal("George Orwell", "1984"): 0
getPrecioFinal("Philip K. Dick", "¿Sueñan los androides con ovejas eléctricas?"): 3.85
getPrecioFinal("isaac asimov", "fundación e imperio"): 10.34
getPrecioFinal("Ray Bradbury", "Fahrenheit 451"): 8.14
getPrecioFinal("Alex Huxley", "Un Mundo Feliz"): 0
getPrecioFinal("Isaac Asimov", "La Fundación"): 8.03
getPrecioFinal("william gibson", "neuromante"): 9.13
getPrecioFinal("Isaac Asimov", "Segunda Fundación"): 9.81
getPrecioFinal("Isaac Newton", "Arithmetica Universalis"): 0
```