

## MÓDULO 2. PROGRAMACIÓN ORIENTADA A OBJETOS CON JAVA

### Relación de Problemas Nº 1

#### Proyecto prJarras (básico)

El objetivo de este ejercicio es crear una clase `Jarra` que utilizaremos para “simular” algunas de las acciones que podemos realizar con una jarra.

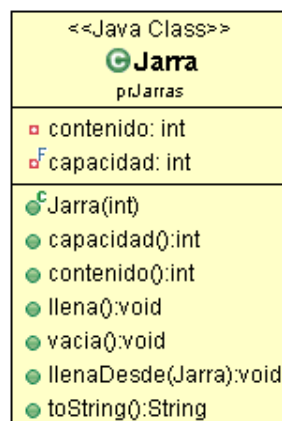
Nuestras jarras van a poder contener cierta cantidad de agua. Así, cada jarra tiene una determinada capacidad (en litros) que será la misma durante la vida de la jarra (dada en el constructor). En un momento determinado, una jarra dispondrá de una cantidad de agua que podrá variar en el tiempo. Las acciones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

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.



## Proyecto prEstadística (básico)

Crear una clase `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 ( $sumaX/numElementos$ ), `double varianza()` que devuelve la varianza ( $sumaX2/numElementos - media()^2$ ). Y `public double desviacionTipica()` que devuelve la raíz de la varianza.

Un ejemplo de uso de la clase anterior es

```
import java.util.Random;

import prEstadistica.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("Desviación típica = "+est.desviacionTipica());
    }
}
```

donde 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.

## Proyecto prCPI (básico)

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 `CPI` 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  
`CPI(x=..., y = ..., z = ..., t = ...)`

Un programa que usa la calculadora para hallar el valor de la expresión anterior sería:

```
package prCPI;

public class Main {
    public static void main(String [] args) {
        // 3 * (6 - 4) + 5;
        // NPI 3 6 4 - * 5 +
        CPI cpi = new CPI();
        cpi.entra(3);
        cpi.entra(6);
        cpi.entra(4);
        cpi.resta();
        cpi.multiplica();
        cpi.entra(5);
        cpi.suma();
        System.out.println("Resultado " + cpi.getResultado ());
    }
}
```

Crear otro programa para calcular la expresión  $3*(6-2)+(2+7)/5$

## Proyecto prRelojArena (composición)

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

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 tiempoRestante()` 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)`.

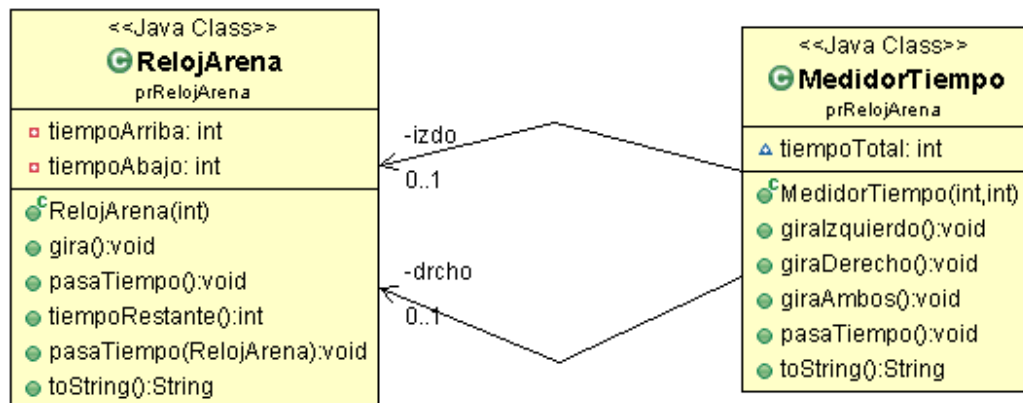
Pongamos un ejemplo de uso de los relojes. Supongamos que tenemos dos relojes de arena de 7 y 5 minutos y vamos a medir 9 minutos.

```
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 giraIzquierda()` 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 incrementa 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.
- 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.



## Proyecto prRectas (composición, excepciones)

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

a) La clase `Punto` se tomará de las transparencias. En adelante, un punto de coordenadas  $x$  e  $y$  se expresará como  $P(x, y)$ .

b) En cuanto a la clase `Vector`, 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 primero creará un vector conociendo sus dos componentes; el segundo lo hará conociendo el punto extremo; 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 con componentes  $x$  e  $y$  se expresará como  $V(x, y)$ .

Un vector *ortogonal* (perpendicular) al vector  $V(x, y)$  es el vector  $V(-y, x)$  (Éste está girado 90 grados en sentido contrario a las agujas del reloj con respecto al anterior). Dos vectores  $V(vx, vy)$  y  $V(ux, uy)$  son *paralelos* si verifican  $vx * uy == vy * ux$ . Esto quiere decir que los vectores 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`.

c) Para construir la clase `Recta` 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 primero que genere la recta conociendo un punto por donde pasa y un vector director; y el segundo que genere la recta conociendo dos puntos por donde pasa. Una recta se expresará como  $R(\text{vector}, \text{punto})$ .

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. Las rectas  $R(V(vx, vy), P(px, py))$  y  $R(V(ux, uy), P(qx, qy))$  se cortan en el punto  $P(ox, oy)$  de ordenada y abscisa:

$$\begin{aligned} ox &= (d1 * ux - vx * d2) / d \\ oy &= - (vy * d2 - uy * d1) / d \end{aligned}$$

siendo

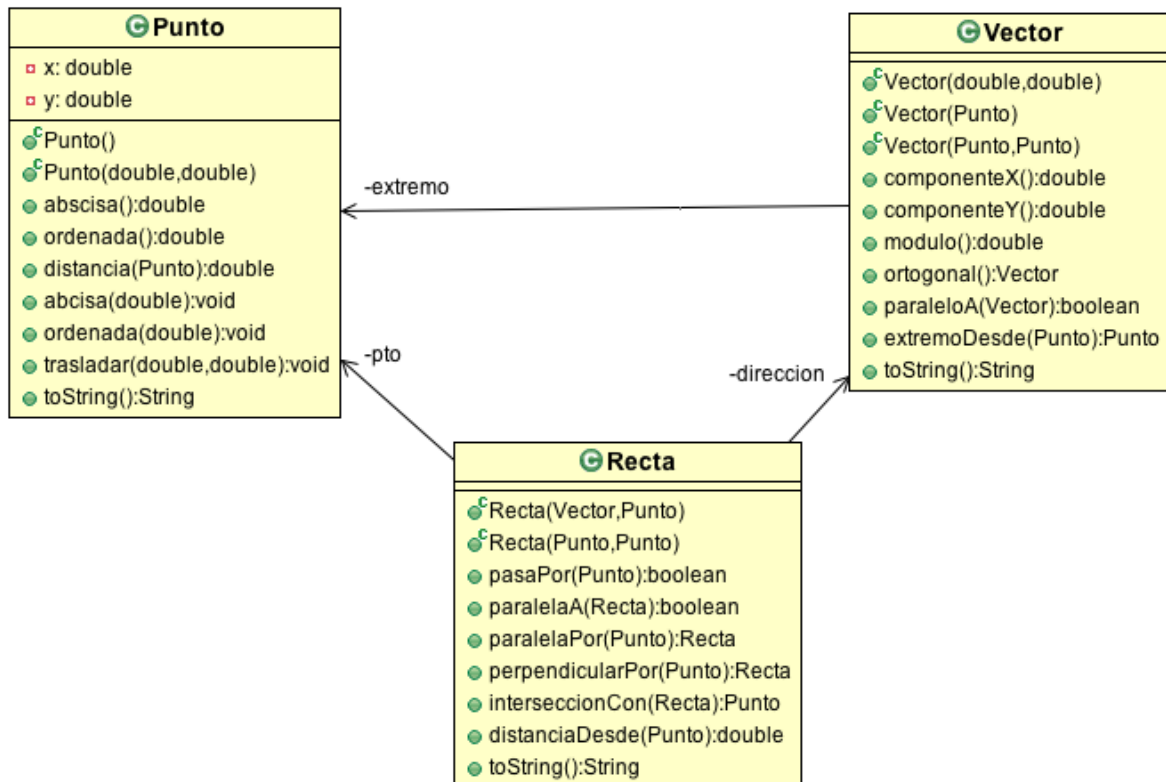
$$\begin{aligned} d &= uy * vx - vy * ux \\ d1 &= vx * py - vy * px \\ d2 &= ux * qy - uy * qx \end{aligned}$$

Como vemos, sólo está definido si  $d$  no es cero.

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  $p$ .

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



d) El siguiente programa calcula el área de un triángulo conociendo los tres puntos del plano que lo delimitan:

```

import prRectas.Punto;
import prRectas.Recta;

public class EjRectas {
    public static void main(String [] args) {
        // Calcular el área del triángulo formado por tres puntos
        Punto p1 = new Punto(0,0);
        Punto p2 = new Punto(4,0);
        Punto p3 = new Punto(2,3);
        // Se calcula la distancia entre p1 y p2
        double base = p1.distancia(p2);
        // Se calcula la recta que pasa por p1 y p2
        Recta r1 = new Recta(p1,p2);
        // se calcula la distancia entre p3 y r1
        double altura = r1.distanciaDesde(p3);
        // El área es base*altura/2
        double area = base*altura/2;
        System.out.printf ("Puntos %s %s %s\n",p1,p2,p3);
        System.out.println ("area = " + area);
        // La perpendicular a r1 por p3
        Recta r2 = r1.perpendicularPor(p3);
        System.out.println("La perpendicular a " + r1 + " por " + p3 + " es: " + r2);
        // La interseccion de r2 con la recta paralela a r1 y que pasa por p3
        System.out.println(r2.interseccionCon(r1.paralelaPor(p3)));
    }
}
  
```



## 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 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.

Aunque puede haber otras alternativas, en los lenguajes en los que hay disponible un mecanismo de tratamiento de excepciones es recomendable su uso, aunque sea de la más sencilla de sus formas. Queremos que se utilice incluso antes de haberlo estudiado en profundidad.

Antes de estudiar el mecanismo de excepciones vamos a utilizarlo de una manera muy simple que pasamos a explicar:

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. Más adelante veremos cómo tratar estas situaciones de una forma adecuada.

Por tanto, en los ejercicios de esta práctica, cualquier situación anómala deberá provocar el lanzamiento de una excepción del tipo `RuntimeException`.