

# Cuadernillo Semestral de Actividades

**Actualizado: 4 de septiembre de 2025**

El presente cuadernillo posee un compilado con todos los ejercicios que se usarán durante el semestre en la asignatura. Los ejercicios están organizados en forma secuencial, siguiendo los contenidos que se van viendo en la materia.

Cada semana les indicaremos cuáles son los ejercicios en los que deberían enfocarse para estar al día y algunos de ellos serán discutidos en la explicación de práctica.

## **Recomendación importante:**

Los contenidos de la materia se incorporan y fijan mejor cuando uno intenta aplicarlos - no alcanza con ver un ejercicio resuelto por alguien más. Para sacar el máximo provecho de los ejercicios, es importante que asistan a las consultas de práctica habiendo intentado resolverlos (tanto como les sea posible). De esa manera podrán hacer consultas más enfocadas y el docente podrá darles mejor feedback.

## Ejercicio 1: WallPost

### Primera parte

Se está construyendo una red social como Facebook o Twitter. Debemos definir una clase `WallPost` con los siguientes atributos: un texto que se desea publicar, cantidad de likes ("me gusta") y una marca que indica si es destacado o no. La clase es subclase de `Object`.

Para realizar este ejercicio, utilice el recurso que se encuentra en el sitio de la cátedra (o que puede descargar desde [acá](#)). Para importar el proyecto, siga los pasos explicados en el documento "*Trabajando con proyectos Maven, importar un proyecto*".

Una vez importado, dentro del mismo, debe completar la clase `WallPost` de acuerdo a la siguiente especificación:

```
/*
 * Permite construir una instancia del WallPost.
 * Luego de la invocación, debe tener como texto: "Undefined post",
 * no debe estar marcado como destacado y la cantidad de "Me gusta" debe ser 0.
 */
public WallPost()
```

```

/*
 * Retorna el texto descriptivo de la publicación
 */
public String getText()

/*
 * Asigna el texto descriptivo de la publicación
 */
public void setText (String descriptionText)

/*
 * Retorna la cantidad de "me gusta"
 */
public int getLikes()

/*
 * Incrementa la cantidad de likes en uno.
 */
public void like()

/*
 * Decrementa la cantidad de likes en uno. Si ya es 0, no hace nada.
 */
public void dislike()

/*
 * Retorna true si el post está marcado como destacado, false en caso contrario
 */
public boolean isFeatured()

/*
 * Cambia el post del estado destacado a no destacado y viceversa.
 */
public void toggleFeatured()

```

## Segunda parte

Utilice los tests provistos por la cátedra para comprobar que su implementación de Wallpost es correcta. Estos se encuentran en el mismo proyecto, en la carpeta test, clase WallPostTest.

Para ejecutar los tests simplemente haga click derecho sobre el proyecto y utilice la opción Run As >> JUnit Test. Al ejecutarlo, se abrirá una ventana con el resultado de la evaluación de los tests. Siéntase libre de investigar la implementación de la clase de test. Ya veremos en detalle cómo implementarlas.



En el informe, Runs indica la cantidad de test que se ejecutaron. En Errors se indica la cantidad que dieron error y en Failures se indica la cantidad que tuvieron alguna falla, es decir, los resultados no son los esperados. Abajo, se muestra el Failure Trace del test que falló. Si lo selecciona, mostrará el mensaje de error correspondiente a ese test, que le ayudará a encontrar la falla. Si hace click sobre alguno de los test, se abrirá su implementación en el editor.

### Tercera parte

Una vez que su implementación pasa los tests de la primera parte puede utilizar la ventana que se muestra a continuación, la cual permite inspeccionar y manipular el post (definir su texto, hacer like / dislike y marcarlo como destacado).



Para visualizar la ventana, sobre el proyecto, usar la opción del menú contextual Run As >> Java Application. La ventana permite cambiar el texto del post, incrementar la cantidad de likes, etc. El botón Print to Console imprimirá los datos del post en la consola.

## Ejercicio 2: Balanza Electrónica

En el taller de programación ud programó una balanza electrónica. Volveremos a programarla, con algún requerimiento adicional.

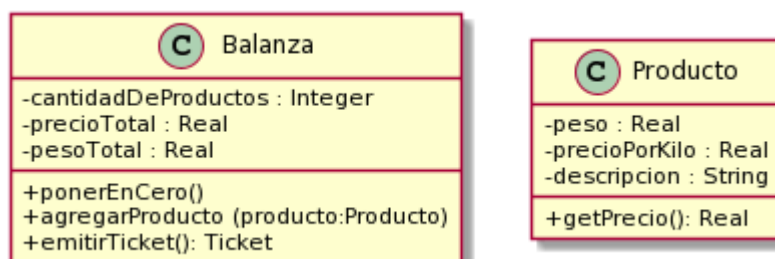
En términos generales, la Balanza electrónica recibe productos (uno a uno), y calcula dos totales: peso total y precio total. Además, la balanza puede poner en cero todos sus valores.

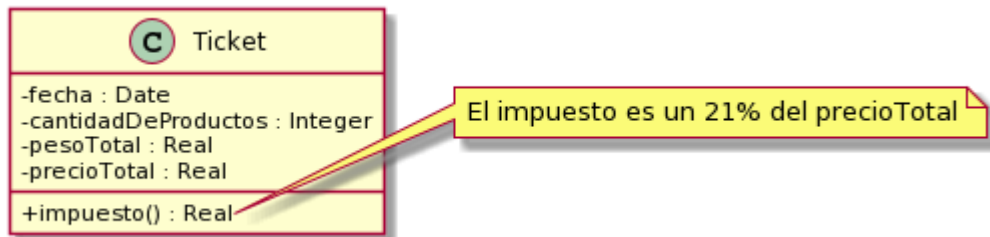
La balanza no guarda los productos. Luego emite un ticket que indica el número de productos considerados, peso total, precio total.

### Tareas:

#### a) Implemente:

Cree un nuevo proyecto Maven llamado `balanzaElectronica`, siguiendo los pasos del documento “*Trabajando con proyectos Maven, crear un proyecto Maven nuevo*”. En el paquete correspondiente, programe las clases que se muestran a continuación.





Observe que no se documentan en el diagrama los mensajes que nos permiten obtener y establecer los atributos de los objetos (accessors). Aunque no los incluimos, verá que los tests fallan si no los implementa. Consulte con el ayudante para identificar, a partir de los tests que fallan, cuales son los accessors necesarios (pista: todos menos los setters de balanza).

Todas las clases son subclasses de Object.

Nota: Para las fechas, utilizaremos la clase `java.time.LocalDate`. Para crear la fecha actual, puede utilizar `LocalDate.now()`. También es posible crear fechas distintas a la actual. Puede investigar más sobre esta clase en <https://docs.oracle.com/en/java/javase/11/docs/api/index.html>

#### b) Probando su implementación:

Para realizar este ejercicio, utilice el recurso que se encuentra en el sitio de la cátedra o que puede descargar desde este [link](#). En este caso, se trata de dos clases, `BalanzaTest` y `ProductoTest`, las cuales debe agregar dentro del paquete tests. Haga las modificaciones necesarias para que el proyecto no tenga errores.

Si todo salió bien, su implementación debería pasar las pruebas que definen las clases agregadas en el paso anterior. El propósito de estas clases es ejercitar una instancia de la clase `Balanza` y verificar que se comporta correctamente.

## Ejercicio 3: Presupuestos

Un presupuesto se utiliza para detallar los precios de un conjunto de productos que se desean adquirir. Se realiza para una fecha específica y es solicitado por un cliente, proporcionando una visión de los costos asociados.

El siguiente diagrama muestra un diseño para este dominio.



#### Tareas:

##### a) Implemente:

Defina el proyecto Ejercicio 3 - Presupuesto y dentro de él implemente las clases que se observan en el diagrama. Ambas son subclases de Object.

##### b) Discuta y reflexione

Preste atención a los siguientes aspectos:

- ¿Cuáles son las variables de instancia de cada clase?
- ¿Qué variables inicializa? ¿De qué formas se puede realizar esta inicialización?
- ¿Qué ventajas y desventajas encuentra en cada una de ellas?

##### c) Probando su código:

Utilice los [tests provistos](#) para confirmar que su implementación ofrece la funcionalidad esperada. En este caso, se trata de dos clases: ItemTest y PresupuestoTest, que debe agregar dentro del paquete tests. Haga las modificaciones necesarias para que el proyecto no tenga errores. Siéntase libre de explorar las clases de test para intentar entender qué es lo que hacen.

## Ejercicio 4: Balanza mejorada

Realizando el ejercicio de los presupuestos, aprendimos que un objeto puede tener una colección de otros objetos. Con esto en mente, ahora queremos mejorar la balanza implementada en el ejercicio 2.

### Tarea 1

Mejorar la balanza para que recuerde los productos ingresados (los mantenga en una colección). Analice de qué forma puede realizarse este nuevo requerimiento e implemente el mensaje

```
public List<Producto> getProductos()
```

que retorna todos los productos ingresados a la balanza (en la compra actual, es decir, desde la última vez que se la puso a cero).

¿Qué cambio produce este nuevo requerimiento en la implementación del mensaje *ponerEnCero()* ?

¿Es necesario, ahora, almacenar los totales en la balanza? ¿Se pueden obtener estos valores de otra forma?

## Tarea 2

Con esta nueva funcionalidad, podemos enriquecer al Ticket, haciendo que él también conozca a los productos (a futuro podríamos imprimir el detalle). Ticket también debería entender el mensaje *public List<Producto> getProductos()* .

- ¿Qué cambios cree necesarios en Ticket para que pueda conocer a los productos?
- ¿Estos cambios modifican las responsabilidades ya asignadas de realizar cálculo del precio total?. ¿El ticket adquiere nuevas responsabilidades que antes no tenía?

## Tarea 3

Después de hacer estos cambios, ¿siguen pasando los tests? ¿Está bien que sea así?

## Ejercicio 5: Inversores

Estamos desarrollando una aplicación móvil para que un inversor pueda conocer el estado de sus inversiones. El sistema permite manejar dos tipos de inversiones: Inversión en acciones e inversión en plazo fijo. En todo momento, se desea poder conocer el valor actual de cada inversión y de las inversiones realizadas por el inversor.

Para las inversiones en acciones el valor actual se calcula multiplicando el valor unitario de una acción por la cantidad de acciones que se posee. De las acciones se conoce el nombre que las identifica en el mercado de valores, un inversor puede invertir en diferentes acciones con diferentes valores unitarios. Por su parte, para los plazos fijos, el valor actual consiste en el cálculo del valor inicial de constitución del plazo fijo sumando los intereses diarios desde la fecha de constitución hasta hoy.

De las inversiones en acciones es importante poder conocer su nombre, la cantidad de acciones en las que se invertirá y el valor unitario de cada acción. Por su parte, los plazos fijos se constituyen en una fecha, es importante conocer el monto depositado y cuál es el porcentaje de interés que genera.

Por último, el valor de inversión actual de un inversor es la suma de los valores actuales de todas las inversiones que posee. Un inversor puede agregar y sacar inversiones de su cartera de inversiones cuando lo desee. Las inversiones pueden ser tanto en acciones como en plazo fijos y pueden estar mezcladas.

**Tareas:**

1. Realice la lista de conceptos candidatos. Clasifique cada concepto dentro de las categorías vistas en la teoría.
2. Grafique el modelo de dominio usando UML.
3. Actualice el modelo de dominio incorporando los atributos a los conceptos
4. Agregue asociaciones entre conceptos, indicando para cada una de ellas la categoría a la que pertenece, de acuerdo a lo explicado en la teoría, y demás atributos, según sea necesario.

## Ejercicio 6: Distribuidora Eléctrica

Una distribuidora eléctrica desea gestionar los consumos de sus usuarios para la emisión de facturas de cobro.

De cada usuario se conoce su nombre y domicilio. Se considera que cada usuario sólo puede tener un único domicilio en donde se registran los consumos.

Los consumos de los usuarios se dividen en dos componentes:

- **Consumo de energía activa:** tiene un costo asociado para el usuario. Se mide en kWh (kilowatt/hora).
- **Consumo de energía reactiva:** no genera ningún costo para el usuario, es decir, se utiliza solamente para determinar si hay alguna bonificación. Se mide en kVARh (kilo voltio-amperio reactivo hora).

Se cuenta con un **cuadro tarifario** que establece el precio del kWh para calcular el costo del consumo de energía activa. Este cuadro tarifario puede ser ajustado periódicamente según sea necesario (por ejemplo, para reflejar cambios en los costos).

Para emitir la factura de un cliente se tiene en cuenta **solo su último consumo registrado**.

Los datos que debe contener la factura son los siguientes:

- El usuario a quien se está cobrando.
- La fecha de emisión.
- La bonificación, si aplica.
- El monto final de la factura: se calcula restando la bonificación al costo del consumo:
  - El costo del consumo se calcula multiplicando el consumo de energía activa por el **precio del kWh** proporcionado por el cuadro tarifario.
  - Se calcula su **factor de potencia** para determinar si hay alguna bonificación aplicable. Si el factor de potencia estimado (fpe) del último consumo del usuario es mayor a 0.8, el usuario recibe una bonificación del 10%.

**Tareas:**

1. Realice la lista de conceptos candidatos. Clasifique cada concepto dentro de las categorías vistas en la teoría.



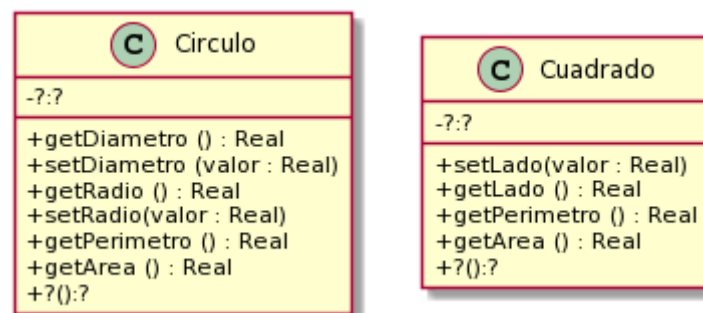
2. Grafique el modelo de dominio usando UML.
3. Actualice el modelo de dominio incorporando los atributos a los conceptos
4. Agregue asociaciones entre conceptos, indicando para cada una de ellas la categoría a la que pertenece, de acuerdo a lo explicado en la teoría, y demás atributos, según sea necesario.

## Ejercicio 7: Figuras y cuerpos

### Figuras en 2D

En Taller de Programación definió clases para representar figuras geométricas. Retomaremos ese ejercicio para trabajar con Cuadrados y Círculos.

El siguiente diagrama de clases documenta los mensajes que estos objetos deben entender.



Fórmulas y mensajes útiles:

- Diámetro del círculo:  $\text{radio} * 2$
- Perímetro del círculo:  $\pi * \text{diámetro}$
- Área del círculo:  $\pi * \text{radio}^2$
- $\pi$  se obtiene enviando el mensaje #pi a la clase Float (Float pi) (ahora Math.PI)

### Tareas:

#### a) Implementación:

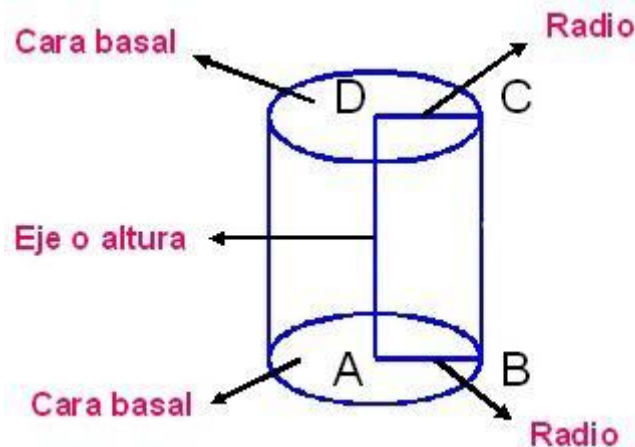
Defina un nuevo proyecto `figurasYCuerpos`. Implemente las clases `Círculo` y `Cuadrado`, siendo ambas subclases de `Object`. Decida usted qué variables de instancia son necesarias. Puede agregar mensajes adicionales si lo cree necesario.

#### b) Discuta y reflexione

¿Qué variables de instancia definió? ¿Pudo hacerlo de otra manera? ¿Qué ventajas encuentra en la forma en que lo realizó?

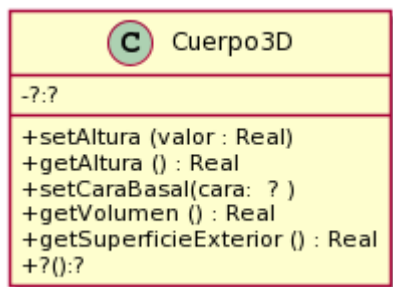
### Cuerpos en 3D

Ahora que tenemos `Círculos` y `Cuadrados`, podemos usarlos para construir cuerpos (en 3D) y calcular su volumen y superficie o área exterior. Vamos a pensar a un cilindro como "un cuerpo que tiene una figura 2D como cara basal y que tiene una altura (vea la siguiente imagen)". Si en el lugar de la figura2D tuviera un círculo, se formaría el siguiente cuerpo 3D.



Si reemplazamos la cara basal por un rectángulo, tendremos un prisma (una caja de zapatos).

El siguiente diagrama de clases documenta los mensajes que entiende un cuerpo3D.



Fórmulas útiles:

- El área o superficie exterior de un cuerpo es:  
 $2 * \text{área-cara-basal} + \text{perímetro-cara-basal} * \text{altura-del-cuerpo}$
- El volumen de un cuerpo es:  $\text{área-cara-basal} * \text{altura}$

Más info interesante: A la figura que da forma al cuerpo (el círculo o el cuadrado en nuestro caso) se le llama directriz. Y a la recta en la que se mueve se llama generatriz. En [wikipedia \(Cilindro\)](https://es.wikipedia.org/wiki/Cilindro)<sup>1</sup> se puede aprender un poco más al respecto.

**Tareas:**

#### a) Implementación

Implemente la clase Cuerpo 3D, la cuál es subclase de Object. Decida usted qué variables de instancia son necesarias. También decida si es necesario hacer cambios en las figuras 2D.

#### b) Pruebas automatizadas

Siguiendo los ejemplos de ejercicios anteriores, ejecute [las pruebas automatizadas provistas](#). En este caso, se trata de tres clases (CuerpoTest, TestCirculo y TestCuadrado) que debe agregar dentro del paquete tests. Haga las modificaciones

<sup>1</sup> <https://es.wikipedia.org/wiki/Cilindro>

necesarias para que el proyecto no tenga errores. Si algún test no pasa, consulte al ayudante.

**c) Discuta y reflexione**

Discuta con el ayudante sus elecciones de variables de instancia y métodos adicionales. ¿Es necesario todo lo que definió?