

Trabajos Prácticos

Ejercicio 1: Wallpost.

Primera parte:

Se está construyendo una red social como Facebook o Twitter. Se debe 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, utilizar el recurso que se encuentra en el sitio de la cátedra (o que puede descargar desde [acá](#)). Para importar el proyecto, seguir los pasos explicados en el documento “Trabajando con proyectos Maven, importar un proyecto”.

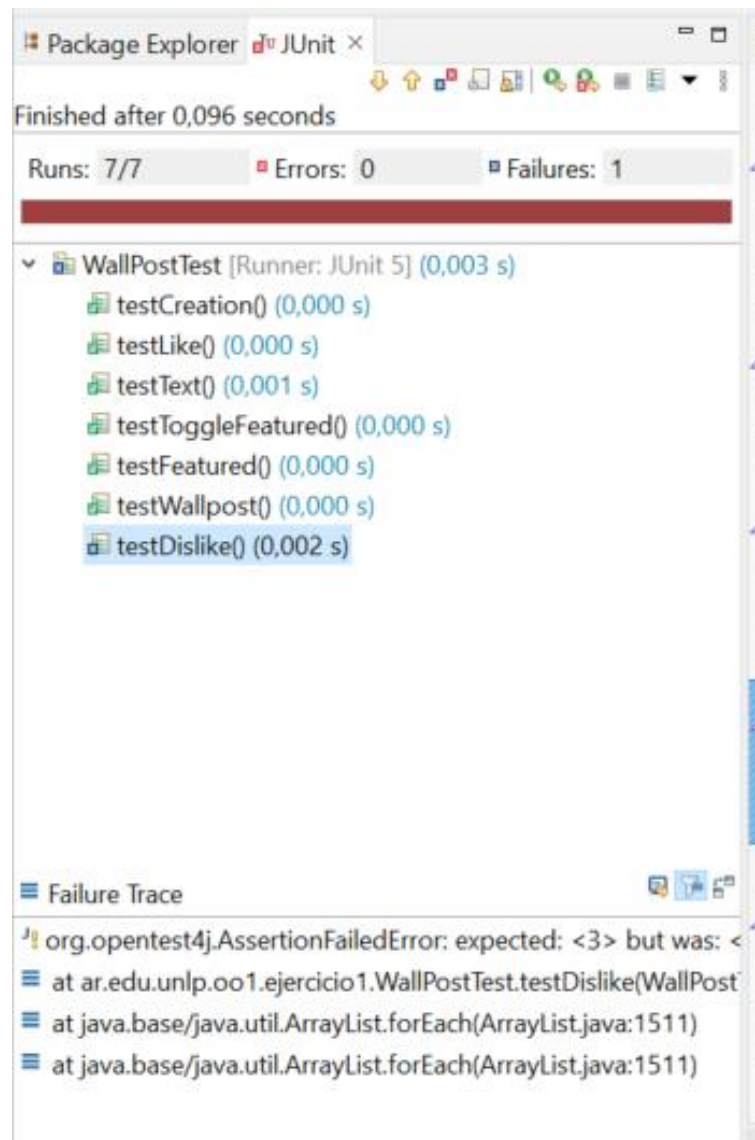
Una vez importado, dentro del mismo, se debe completar la clase WallPost de acuerdo a la siguiente especificación:

(Ver Trabajos Prácticos (P).pdf).

Segunda parte:

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

Para ejecutar los tests, simplemente, hacer click derecho sobre el proyecto y utilizar la opción Run As >> JUnit Test. Al ejecutarlo, se abrirá una ventana con el resultado de la evaluación de los tests. Sentirse libre de investigar la implementación de la clase de test. Ya se verá, 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 se selecciona, mostrará el mensaje de error correspondiente a ese test, que ayudará a encontrar la falla. Si se hace click sobre alguno de los test, se abrirá su implementación en el editor.

Tercera parte:

Una vez que la implementación pasa los tests de la primera parte, se 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.

Ver proyecto OO1_E1 de Java.

Ejercicio 2: Balanza Electrónica.

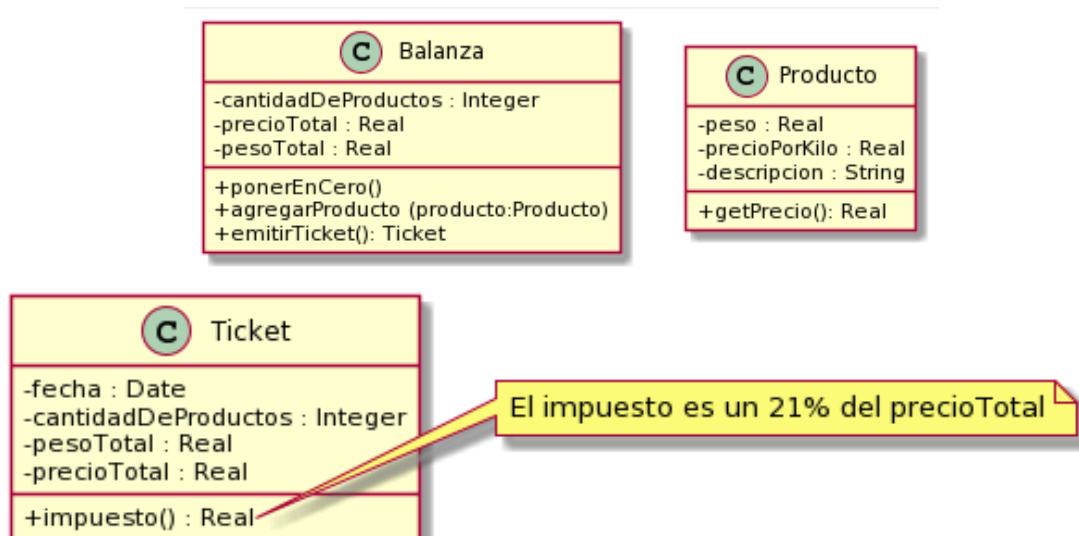
En el taller de programación, se programó una balanza electrónica. Se volverá a programar, con algún requerimiento adicional.

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.

(a) Implementación:

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



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

Todas las clases son subclases de *Object*.

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

(b) Probar implementación:

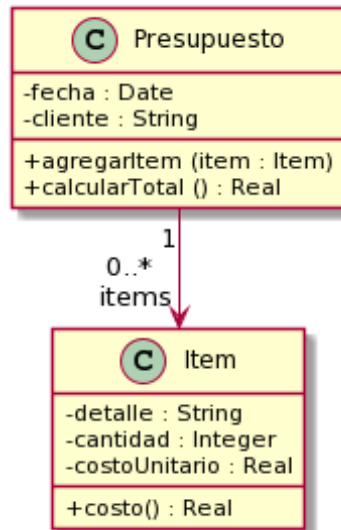
Para realizar este ejercicio, utilizar el recurso que se encuentra en el sitio de la cátedra o que puede descargarse desde este [link](#). En este caso, se trata de dos clases, `BalanzaTest` y `ProductoTest`, las cuales se deben agregar dentro del paquete `tests`. Hacer las modificaciones necesarias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

Ver proyecto OO1_E2 de Java.

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:



(a) Implementación:

Definir el proyecto Ejercicio 3 - Presupuesto y, dentro de él, implementar las clases que se observan en el diagrama. Ambas son subclases de `Object`.

(b) Discutir y reflexionar:

Prestar atención a los siguientes aspectos:

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

(c) Probar implementación:

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

Ver proyecto OO1_E3 de Java.

Ejercicio 4: Balanza Mejorada.

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

(a) *Mejorar la balanza para que recuerde los productos ingresados (los mantenga en una colección). Analizar de qué forma puede realizarse este nuevo requerimiento e implementar 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?*

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

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

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

Ver proyecto OO1_E4 de Java.

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 y que 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 fijos 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.

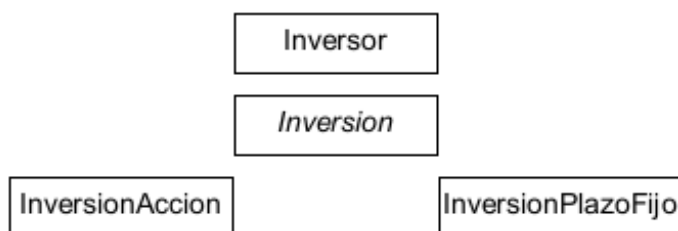
(a) *Realizar la lista de conceptos candidatos. Clasificar cada concepto dentro de las categorías vistas en la teoría.*

Lista de conceptos candidatos:

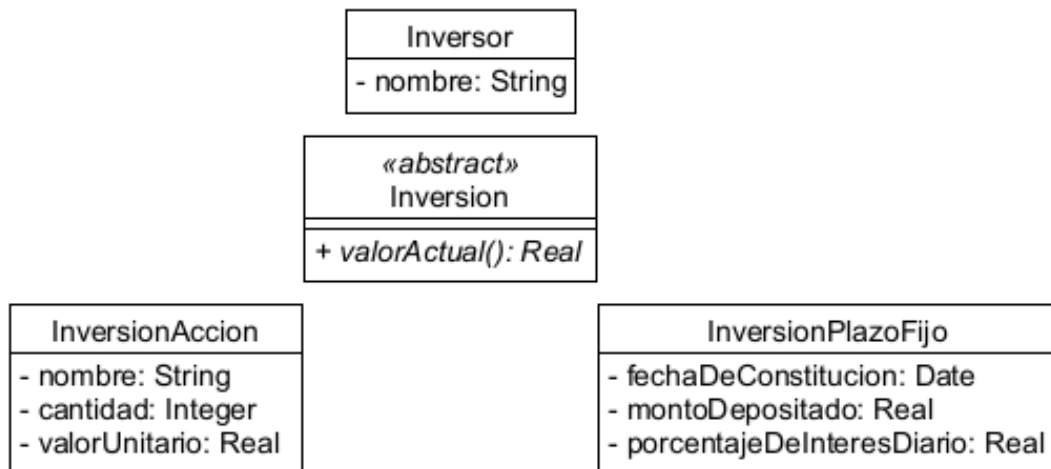
1. Objeto físico o tangible: Objeto relevante.
2. Especificación de una cosa: Características esenciales de objetos.
3. Lugar: Lugar relevante.
4. Transacción: Acontecimientos donde algo cambia en el sistema.
5. Roles de la gente: Personas o roles dentro del sistema.
6. Contenedor de cosas: Agrupa o contiene objetos.
7. Cosas de un contenedor: Son los objetos dentro del contenedor anterior.
8. Otros sistemas: Con los que se podría conectar, si existieran.
9. Hechos: Eventos o datos que pueden cambiar.
10. Reglas y políticas: Relacionadas con la lógica del negocio.
11. Registros financieros/laborales: Datos relevantes del ámbito económico.
12. Manuales/documentos: Manuales/documentos relevantes.

Categoría de clase conceptual	Ejemplos
Objeto físico o tangible	aplicación móvil sistema de inversiones
Especificación de una cosa	nombre del inversor nombre de la acción cantidad de acciones valor unitario de la acción fecha de constitución del plazo fijo monto depositado en el plazo fijo porcentaje de interés que genera el plazo fijo
Lugar	mercado de valores
Transacción	registro de una inversión (compra de acción, constitución de plazo fijo) cálculo del valor de las acciones cálculo del valor del plazo fijo cálculo del valor actual de la cartera
Roles de la gente	inversor
Contenedor de cosas	inversor
Cosas de un contenedor	inversión inversión en acciones inversión en plazo fijo
Otros sistemas	sistema financiero
Hechos	intereses ganados valor de la inversión actual suma del valor actual de todas las inversiones del inversor
Reglas y políticas	regla de cálculo del valor de las acciones regla de cálculo del valor del plazo fijo regla de cálculo del valor actual de la cartera
Registros financieros/laborales	registro de inversiones del inversor registro de movimientos o cambios en la cartera
Manuales, documentos	detalle de la inversión

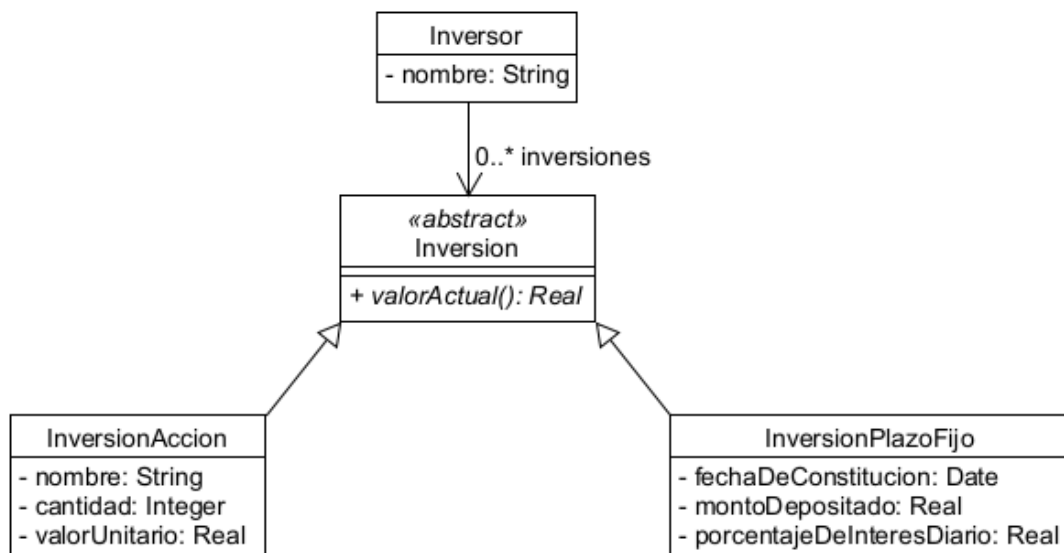
(b) Graficar el modelo de dominio usando UML.



(c) Actualizar el modelo de dominio incorporando los atributos a los conceptos.



(d) Agregar 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 sólo 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 sólo su último consumo registrado. Los datos que debe contener la factura son los siguientes:

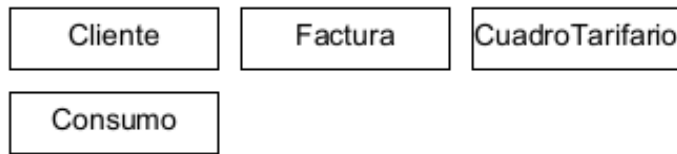
- *El usuario a quien se está cobrando.*
- *La fecha de emisión.*
- *La bonificación, si aplica.*
- *El monto final de la factura: Se calcula restando la bonificación al costo del consumo:*
 - *El costo del consumo se calcula multiplicando el consumo de energía activa por el precio del kWh proporcionado 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%.*

(a) *Realizar la lista de conceptos candidatos.*

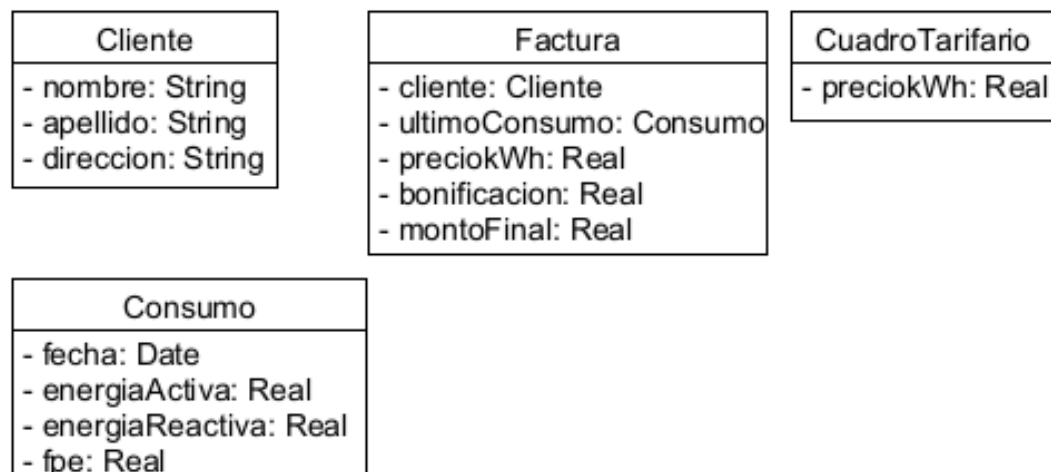
Lista de conceptos candidatos:

Categoría de clase conceptual	Ejemplos
Objeto físico o tangible	cliente domicilio
Especificación de una cosa	nombre, apellido y dirección del cliente fecha del consumo energía activa del consumo energía reactiva del consumo factor de potencia estimado (fpe) precio del kWh actual cliente de la factura último consumo registrado precio del kWh de la factura bonificación de la factura monto final de la factura
Lugar	domicilio del cliente
Transacción	actualización del cuadro tarifario emisión de una factura registro de un consumo cálculo del factor de potencia estimado cálculo del costo del consumo determinación de bonificación cálculo del monto final de la factura
Roles de la gente	cliente
Contenedor de cosas	cliente factura
Cosas de un contenedor	consumo del cliente consumo de la factura
Otros sistemas	-
Hechos	precio del kWh actual cliente de la factura último consumo registrado (fecha, energía activa, energía reactiva, fpe) precio del kWh de la factura bonificación de la factura monto final de la factura
Reglas y políticas	regla de que la factura se emite sólo con el último consumo registrado regla del cálculo del costo del consumo política de bonificación ($fpe > 0,8$)
Registros financieros/laborales	factura costo del consumo bonificación monto final
Manuales, documentos	cuadro tarifario factura

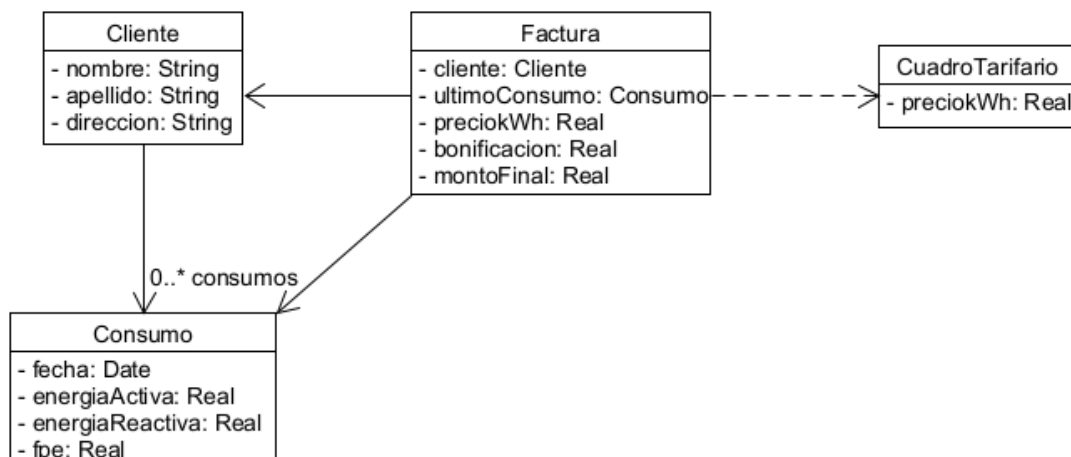
(b) Graficar el modelo de dominio usando UML.



(c) Actualizar el modelo de dominio incorporando los atributos a los conceptos.



(d) Agregar 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.

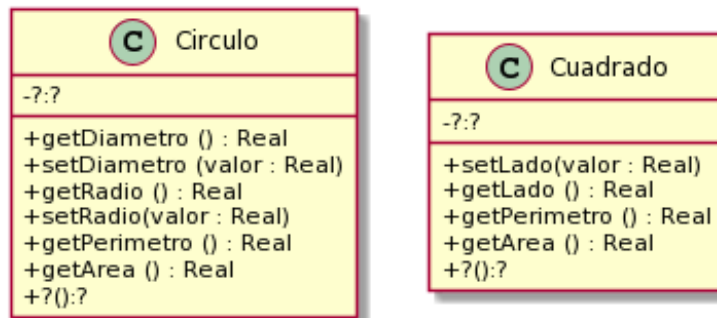


Ejercicio 7: Figuras y Cuerpos.

Figuras en 2D:

En Taller de Programación, se definieron clases para representar figuras geométricas. Se retomará 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: $\pi * \text{radio}^2$.
- π se obtiene enviando el mensaje `#pi` a la clase `Float` (`Float pi`) (ahora `Math.PI`).

(a) Implementación:

Definir un nuevo proyecto `figurasYCuerpos`. Implementar las clases `Circulo` y `Cuadrado`, siendo ambas subclases de `Object`. Decidir qué variables de instancia son necesarias. Se pueden agregar mensajes adicionales si se cree necesario.

(b) Discutir y reflexionar:

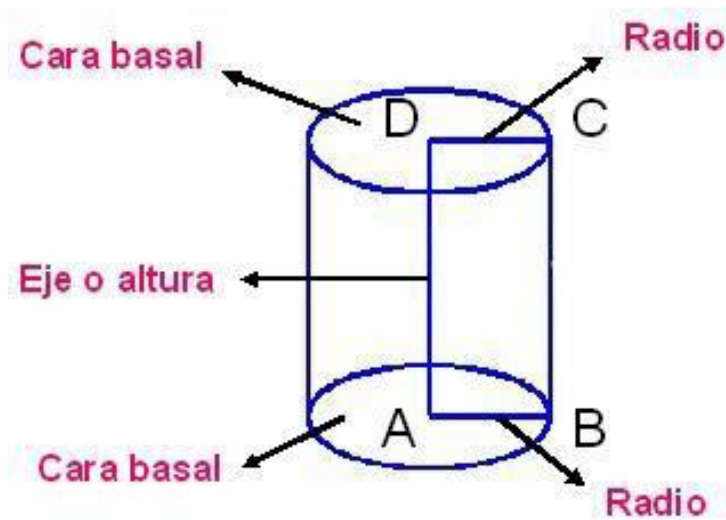
¿Qué variables de instancia se definieron? ¿Se puede hacer de otra manera? ¿Qué ventajas se encuentran en la forma en que se realizó?

Ver proyecto `OO1_E7` de Java.

Cuerpos en 3D:

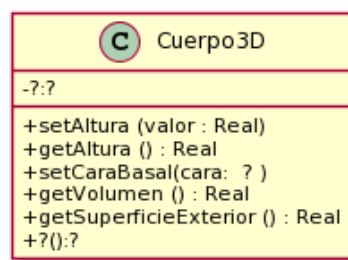
Ahora que se tiene `Círculos` y `Cuadrados`, se pueden usar para construir cuerpos (en 3D) y calcular su volumen y superficie o área exterior.

- Se va a pensar a un cilindro como “un cuerpo que tiene una figura 2D como cara basal y que tiene una altura (ver la siguiente imagen)”. Es decir, si, en el lugar de la figura 2D, se tuviera un círculo, se formaría el siguiente cuerpo 3D.



- Si se reemplaza la cara basal por un rectángulo, se tendrá 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 le llama *generatriz*. En [Wikipedia \(Cilindro\)](#), se puede aprender un poco más al respecto.

(a) Implementación:

Implementar la clase **Cuerpo 3D**, la cual es subclase de **Object**. Decidir qué variables de instancia son necesarias. También decidir si es necesario hacer cambios en las figuras 2D.

(b) Probar implementación:

Siguiendo los ejemplos de ejercicios anteriores, ejecutar las pruebas automatizadas provistas. En este caso, se trata de tres clases (CuerpoTest, TestCirculo y TestCuadrado) que se deben agregar dentro del paquete tests. Hacer las modificaciones necesarias para que el proyecto no tenga errores. Si algún test no pasa, consultar al ayudante.

(c) *Discutir y reflexionar:*

Discutir con el ayudante las elecciones de variables de instancia y métodos adicionales. ¿Es necesario todo lo que se definió?

Ver proyecto OO1_E7 de Java.

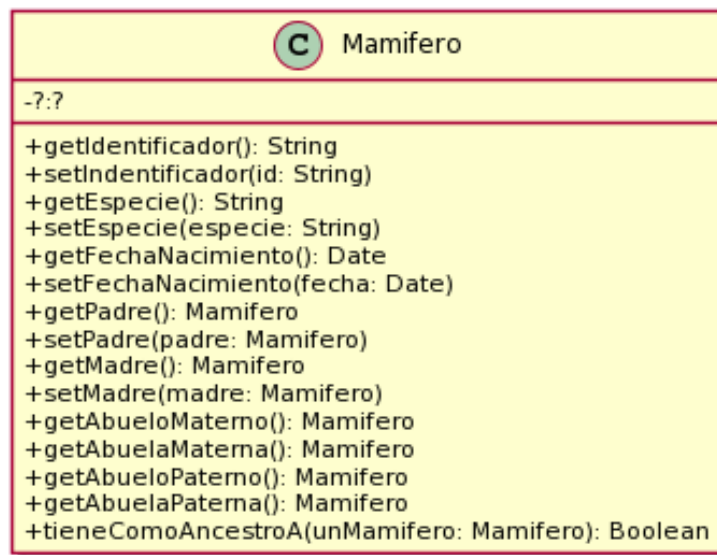
Ejercicio 8: Genealogía Salvaje.

En una reserva de vida salvaje (como la estación de cría ECAS, en el camino Centenario), los cuidadores quieren llevar registro detallado de los animales que cuidan y sus familias. Para ello, han pedido ayuda. Se debe:

(a) *Completar el diseño e implementar:*

Modelar una solución en objetos e implementar la clase Mamífero (como subclase de Object). El siguiente diagrama de clases (incompleto) nos da una idea de los mensajes que un mamífero entiende.

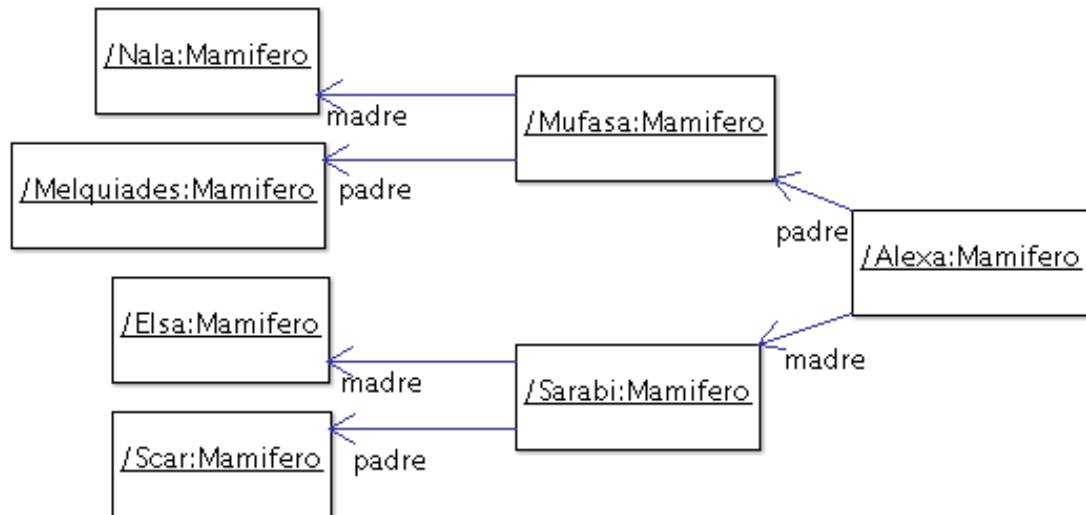
Proponer una solución para el método tieneComoAncestroA(...) y dejar la implementación para el final y discutir la solución con el ayudante.



Completar el diagrama de clases para reflejar los atributos y relaciones requeridas en la solución.

(b) *Pruebas automatizadas:*

Siguiendo los ejemplos de ejercicios anteriores, ejecutar las pruebas automatizadas provistas. En este caso, se trata de una clase, MamiferoTest, que se debe agregar dentro del paquete tests. En esta clase, se trabaja con la familia mostrada en la siguiente figura:



En el diagrama, se puede apreciar el nombre/identificador de cada uno de ellos (por ejemplo, Nala, Mufasa, Alexa, etc). Hacer las modificaciones necesarias para que el proyecto no tenga errores. Si algún test no pasa, consultar al ayudante.

Ver proyecto OO1_E8 de Java.

Ejercicio 9: Red de Alumbrado. (*)

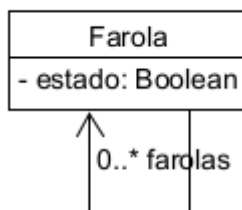
Imaginar una red de alumbrado donde cada farola está conectada a una o varias vecinas formando un grafo conexo. Cada una de las farolas tiene un interruptor. Es suficiente con encender o apagar una farola cualquiera para que se enciendan o apaguen todas las demás. Sin embargo, si se intenta apagar una farola apagada (o si se intenta encender una farola encendida), no habrá ningún efecto, ya que no se propagará esta acción hacia las vecinas.

La funcionalidad a proveer permite:

1. Crear farolas (inicialmente, están apagadas).
2. Conectar farolas a tantas vecinas como uno quiera (las conexiones son bidireccionales).
3. Encender una farola (y obtener el efecto antes descrito).
4. Apagar una farola (y obtener el efecto antes descrito).

(a) Modelar e implementar:

(i) Realizar el diagrama UML de clases de la solución al problema.



(ii) Implementar, en Java, la clase Farola, como subclase de Object, con los siguientes métodos:

Ver proyecto OO1_E9 de Java.

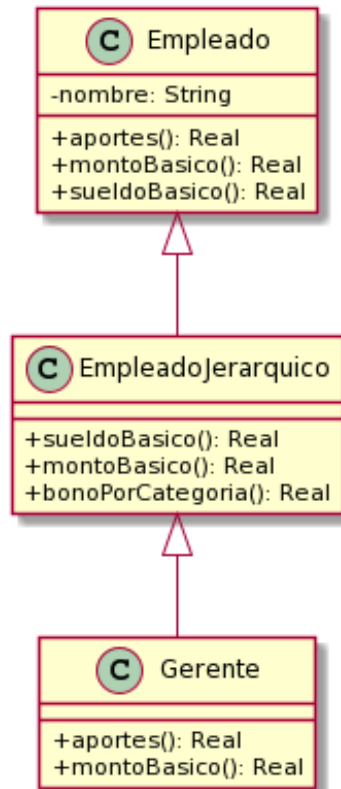
(b) Pruebas automatizadas:

Utilizar los tests provistos por la cátedra para probar las implementaciones del inciso (a).

Ver proyecto OO1_E9 de Java.

Ejercicio 10: Method Lookup con Empleados.

Sea la jerarquía de Empleado como muestra la figura de la izquierda, cuya implementación de referencia se incluye en la tabla.



Empleado	EmpleadoJerarquico	Gerente
<pre> public double montoBasico() { return 35000; } </pre>	<pre> public double sueldoBasico() { return super.sueldoBasico()+ this.bonoPorCategoria(); } </pre>	<pre> public double aportes() { return this.montoBasico() * 0.05d; } </pre>
<pre> public double aportes(){ return 13500; } </pre>	<pre> public double montoBasico() { return 45000; } </pre>	<pre> public double montoBasico() { return 57000; } </pre>
<pre> public double sueldoBasico() { return this.montoBasico() + this.aportes(); } </pre>	<pre> public double bonoPorCategoria() { return 8000; } </pre>	

Analizar cada uno de los siguientes fragmentos de código y resolver las tareas indicadas abajo:

```
Gerente alan = new Gerente("Alan Turing");  
double aportesDeAlan = alan.aportes();
```

```
Gerente alan = new Gerente("Alan Turing");  
double sueldoBasicoDeAlan = alan.sueldoBasico();
```

(a) *Listar todos los métodos, indicando nombre y clase, que son ejecutados como resultado del envío del último mensaje de cada fragmento de código (por ejemplo, (1) aportes de la clase Empleado, (2) ...).*

Primer fragmento de código:

- (1) *aportes* de la clase Gerente.
- (2) *montoBasico* de la clase Gerente.

Segundo fragmento de código:

- (1) *sueldoBasico* de la clase EmpleadoJerarquico.
- (2) *sueldoBasico* de la clase Empleado.
- (3) *montoBasico* de la clase Gerente.
- (4) *aportes* de la clase Gerente.
- (5) *bonoPorCategoria* de la clase EmpleadoJerarquico.

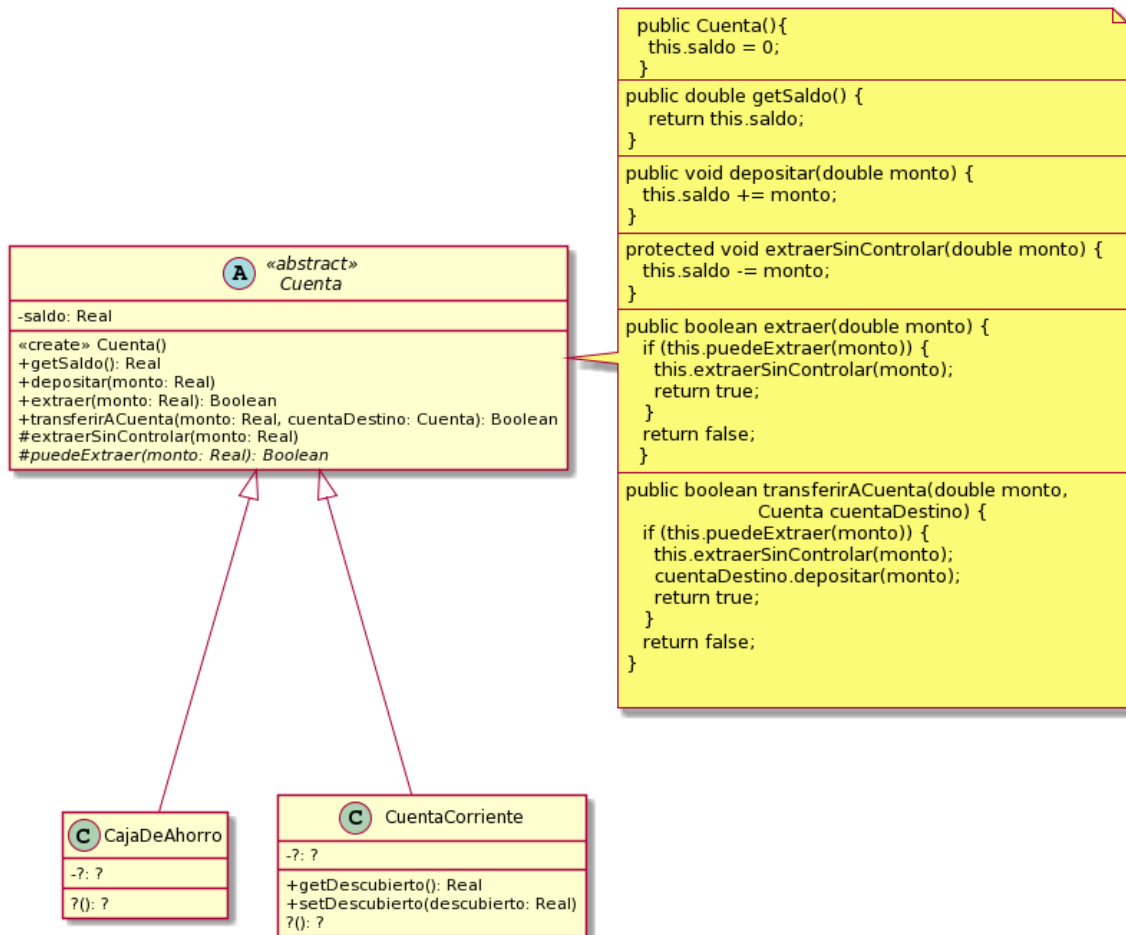
(b) *¿Qué valores tendrán las variables `aportesDeAlan` y `sueldoBasicoDeAlan` luego de ejecutar cada fragmento de código?*

Los valores que tendrán las variables *aportesDeAlan* y *sueldoBasicoDeAlan* luego de ejecutar cada fragmento de código serán 2.850 y 67.850, respectivamente.

Ver proyecto OO1_E10 de Java.

Ejercicio 11: Cuenta con Ganchos.

Observar, con detenimiento, el diseño que se muestra en el siguiente diagrama. La clase *cuenta* es abstracta. El método *puedeExtraer()* es abstracto. Las clases *CajaDeAhorro* y *CuentaCorriente* son concretas y están incompletas.



(a) Completar la implementación de las clases *CajaDeAhorro* y *CuentaCorriente* para que se puedan efectuar depósitos, extracciones y transferencias teniendo en cuenta los siguientes criterios:

- Las cajas de ahorro sólo pueden extraer y transferir cuando cuentan con fondos suficientes.
- Las extracciones, los depósitos y las transferencias desde cajas de ahorro tienen un costo adicional de 2% del monto en cuestión (tenerlo en cuenta antes de permitir una extracción o transferencia desde caja de ahorro).
- Las cuentas corrientes pueden extraer aun cuando el saldo de la cuenta sea insuficiente. Sin embargo, no deben superar cierto límite por debajo del saldo. Dicho límite se conoce como límite de descubierto (algo así como el máximo saldo negativo permitido). Ese límite es diferente para cada cuenta (lo negocia el cliente con la gente del banco).
- Cuando se abre una cuenta corriente, su límite descubierto es 0 (no olvidar definir el constructor por default).

(b) Reflexionar, charlar con el ayudante y responder a las siguientes preguntas:

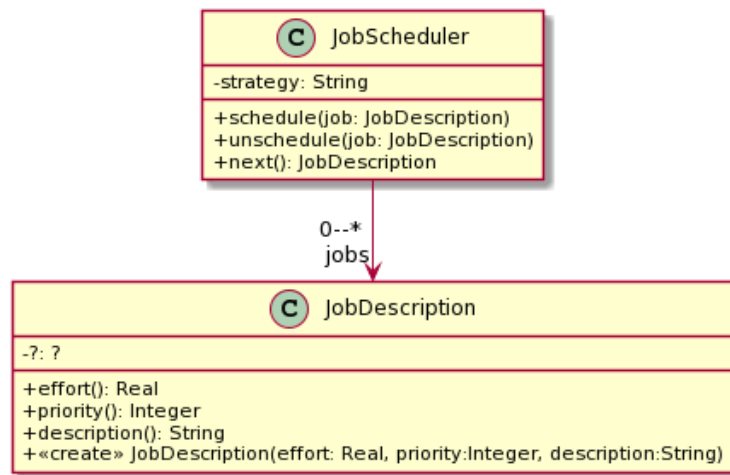
- ¿Por qué este ejercicio se llama “Cuenta con ganchos”?
- En las implementaciones de los métodos `extraer()` y `transferirACuenta()` que se ven en el diagrama, ¿quién es `this`? ¿Se puede decir de qué clase es `this`?
- ¿Por qué se decide que los métodos `puedeExtraer()` y `extraerSinControlar` tengan visibilidad “protegido”?
- ¿Se puede transferir de una caja de ahorro a una cuenta corriente y viceversa? ¿por qué? ¡Probarlo!
- ¿Cómo se declara, en Java, un método abstracto? ¿Es obligatorio implementarlo?
- ¿Qué dice el compilador de Java si una subclase no implementa un método abstracto que hereda?

(c) Escribir los tests de unidad que se crean necesarios para validar que la implementación funciona adecuadamente.

Ver proyecto OO1_E11 de Java.

Ejercicio 12: Job Scheduler.

El *JobScheduler* es un objeto cuya responsabilidad es determinar qué trabajo debe resolverse a continuación. El siguiente diseño ayuda a entender cómo funciona la implementación actual del *JobScheduler*.



- El mensaje `schedule(job: JobDescription)` recibe un *job* (trabajo) y lo agrega al final de la colección de trabajos pendientes.
- El mensaje `next()` determina cuál es el siguiente trabajo de la colección que debe ser atendido, lo retorna y lo quita de la colección.

En la implementación actual del método `next()`, el *JobScheduler* utiliza el valor de la variable `strategy` para determinar cómo elegir el siguiente trabajo.

Dicha implementación presenta dos serios problemas de diseño:

- Secuencia de `ifs` (o sentencia `switch/case`) para implementar alternativas de un mismo comportamiento.
- Código duplicado.

(a) Analizar el código existente:

Utilizar el [código y los tests](#) provistos por la cátedra y aplicar lo aprendido (en particular, en relación a herencia y polimorfismo) para eliminar los problemas mencionados. Sentirse libre de agregar nuevas clases como se considere necesario. También se puede cambiar la forma en la que los objetos se crean e inicializan. Asumir que una vez elegida una estrategia para un scheduler no puede cambiarse.

(b) Pruebas automatizadas:

Los cambios, probablemente, hagan que los tests dejen de funcionar. Corregirlos y mejorarlos como sea necesario.

Ver proyecto OO1_E13 de Java.

Ejercicio 13: ¡A Implementar Inversores!

Retomando el Ejercicio 5, donde se trabajó en el diseño y modelado UML de un sistema de inversiones, donde se definieron las clases, atributos y asociaciones necesarias para representar inversores y sus diferentes tipos de inversiones. Ahora, es el momento de llevar el diseño a la práctica: se va a implementar, en Java, lo diseñado y asegurar su calidad mediante pruebas automatizadas.

(a) Implementar:

- (i) Realizar el mapeo del modelo conceptual a un diagrama de clases de UML.*
- (ii) Implementar, en Java, lo necesario para que se pueda conocer el valor actual de cada inversión. Y también el monto total de las inversiones realizadas por un inversor.*

(b) Pruebas automatizadas:

- (i) Diseñar los casos de prueba teniendo en cuenta los conceptos de valores de borde y particiones equivalentes vistos en la teoría.*
- (ii) Implementar, utilizando JUnit, los tests automatizados diseñados en el inciso anterior.*

(c) Discutir con el ayudante:

- (i) Consultar con un ayudante los casos de prueba diseñados.*

Ver proyecto OO1_E13 de Java.

Ejercicio 14: Volumen y Superficie de Sólidos.

Ejercicio 15: Cliente de Correo. (*)

Ejercicio 16: Intervalo de Tiempo.

Ejercicio 17: Intervalo de Tiempo (¡Otra Vez!).

Ejercicio 18: *Filtered Set.*

Ejercicio 19: Alquiler de Propiedades. (*)

Ejercicio 20: Políticas de Cancelación - UML.

Ejercicio 21: Servicio de Envíos de Paquetes - UML.

Ejercicio 22: Sistema de Pedidos.

Ejercicio 23: *PoolCar* - UML.

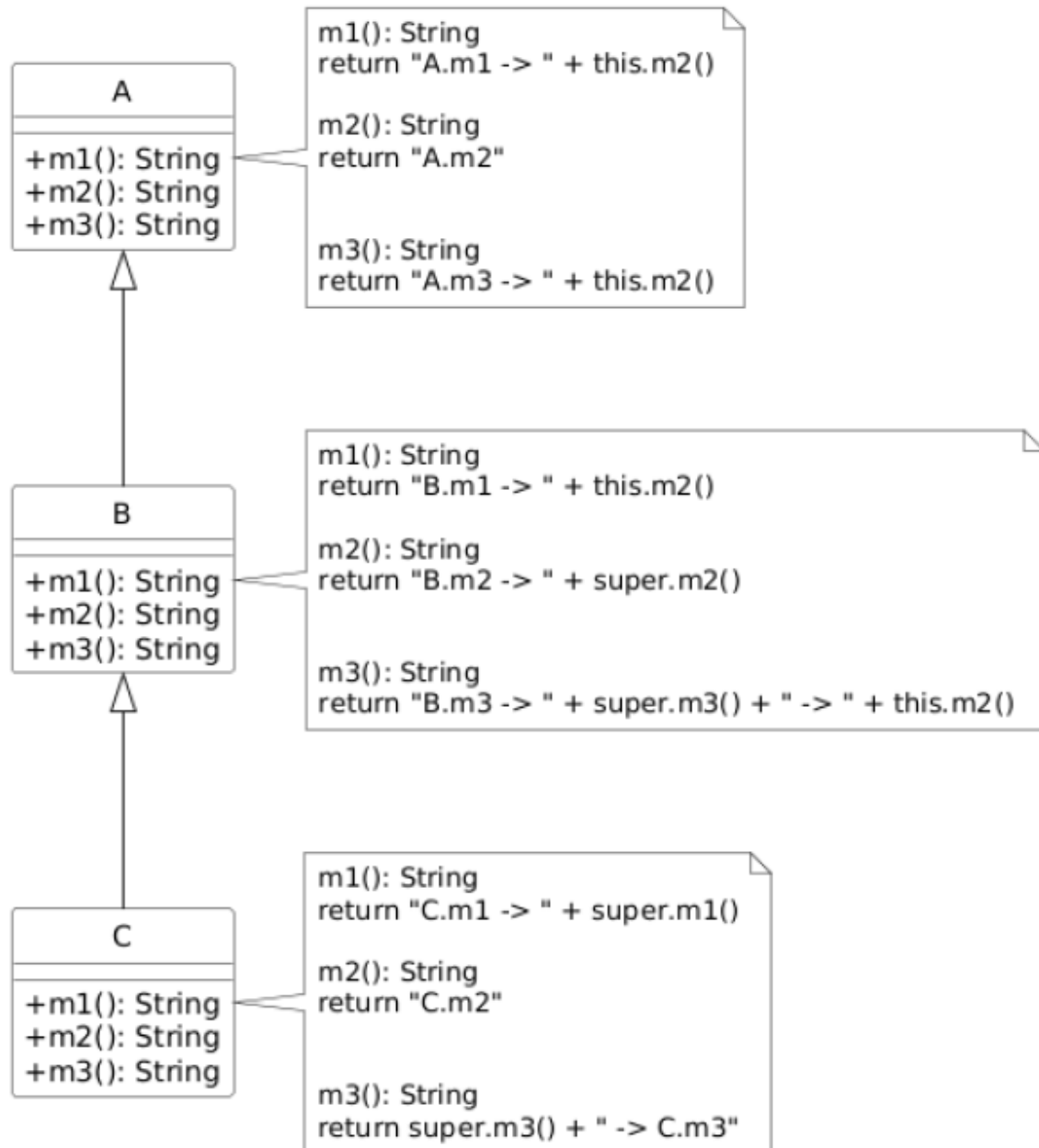
Ejercicio 24: *GreenHOOme*.

Ejercicio 25: *Bag*.

Ejercicio 26: Estadísticas del Cliente de Correo.

Ejercicio 27: Method Lookup.

Dado el siguiente modelo y los fragmentos de códigos mostrados, marcar la respuesta para cada una de las preguntas enunciadas. Sólo una opción es la correcta.



(a) Observando el diagrama, indicar qué texto retorna el siguiente fragmento de código:

```

C c= new C();
c.m1();
  
```

1. C.m1 -> B.m1 -> C.m2.
2. C.m1 -> A.m1 -> A.m2.
3. C.m1 -> B.m1 -> B.m2 -> A.m2.
4. C.m1 -> A.m1 -> C.m2.

(b) Observando el diagrama, indicar qué texto retorna el siguiente fragmento de código:

```
C c= new C();  
c.m2();
```

1. A.m2.
2. B.m2 -> A.m2.
3. C.m2.
4. B.m2 -> C.m2.

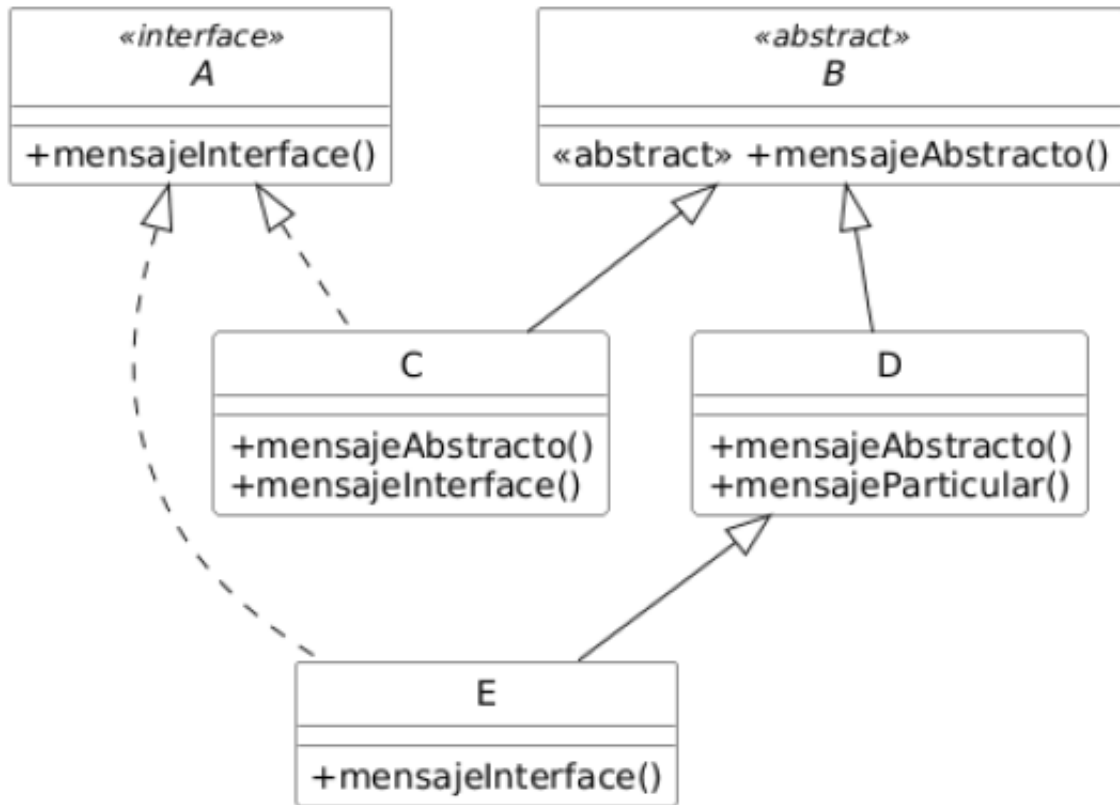
(c) Observando el diagrama, indicar qué texto retorna el siguiente fragmento de código:

```
C c= new C();  
c.m3();
```

1. B.m3 -> A.m3 -> C.m2 -> C.m2.
2. B.m3 -> A.m3 -> C.m2 -> C.m2 -> C.m3.
3. B.m3 -> B.m3 -> A.m2 -> A.m2 -> C.m3.
4. B.m3 -> A.m3 -> C.m2 -> C.m3.

Ejercicio 28: TipadOOs.

Dado el diagrama de clases UML proporcionado, completar todos los bloques de código reemplazando los signos de interrogación (???) con los tipos y métodos correctos. Escribir todas las combinaciones válidas posibles.

**(a)**

```
A objeto= new ???();
objeto.???
```

```
A objeto= new C();
objeto.mensajeAbstracto;
objeto.mensajeInterface;
```

```
A objeto= new E();
objeto.mensajeAbstracto;
objeto.mensajeParticular;
objeto.mensajeInterface;
```

(b)

```
B objeto= new ???();
objeto.???
```

```
B objeto= new C();  
objeto.mensajeAbstracto;  
objeto.mensajeInterface;
```

```
B objeto= new D();  
objeto.mensajeAbstracto;  
objeto.mensajeParticular;
```

```
B objeto= new E();  
objeto.mensajeAbstracto;  
objeto.mensajeParticular;  
objeto.mensajeInterface;
```

(c)

```
D objeto = new ???();  
objeto.???
```

```
D objeto= new D();  
objeto.mensajeAbstracto;  
objeto.mensajeParticular;
```

```
D objeto= new E();  
objeto.mensajeAbstracto;  
objeto.mensajeParticular;  
objeto.mensajeInterface;
```

(d)

```
C objeto= new C();  
objeto.???
```

```
C objeto= new C();  
objeto.mensajeAbstracto;  
objeto.mensajeInterface;
```

(e)

```
??? objeto= new C();  
objeto.mensajeAbstracto;
```

```
B objeto= new C();  
objeto.mensajeAbstracto;
```

```
C objeto= new C();  
objeto.mensajeAbstracto;
```

(f)

```
??? objeto= new C();  
objeto.mensajeInterface;
```

```
A objeto= new C();  
objeto.mensajeInterface;
```

```
C objeto= new C();  
objeto.mensajeInterface;
```

Ejercicio 29: Plataforma de *Streaming*.