Tarea 3: Programación orientada a objetos

Lenguajes de Programación

(No, la tarea no es programar Minecraft)

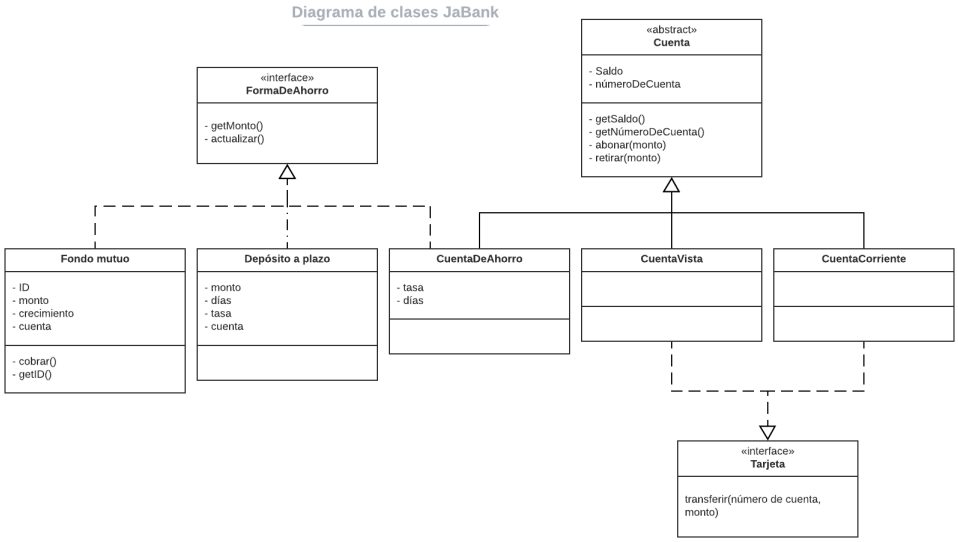
Equipo docente LP 2020-1

22 de junio de 2020

1. Contexto

JaBank, el banco para el que usted trabaja, está desarrollando un nuevo tipo de dispositivo para que mineros puedan manejar sus inversiones mientras trabajan varios días, semanas y meses bajo tierra, sin conexión a internet. Para esto están trabajando diversos equipos y a usted le han encargado el sistema que actúa en cuanto se restablece la conexión (al salir de la mina), que debe leer el registro de lo que ha hecho la persona e informar el resultado al Sistema Integrado de Gestión de Ahorros del banco (SIGA) que está programado en Java.
Por este motivo, su sistema debe estar en Java también, para ser compatible. Como la interfaz del SIGA está siendo desarrollada por otro equipo, de manera provisoria su sistema debe mostrar por consola la información. El departamento de desarrollo le ha indicado la arquitectura de su sistema, que debe implementar el diagrama de clases mostrado a continuación.
Además, a JaBank le preocupa que su competidor, el muy prestigioso banco *Universal Sack of Money* (USM) pueda *hackear* su sistema, por lo que este debe pasar por una seria prueba de seguridad.

2. Tarea



Como se ve en el diagrama, usted debe crear la clase abstracta **Cuenta** y la interfaz **FormaDeAhorro**. Heredando de Cuenta, debe implementar **CuentaVista**, **CuentaDeAhorro** y **CuentaCorriente**. Además, estas dos últimas deben implementar la interfaz **Tarjeta.**
Por otro lado, debe implementar **FondoMutuo** y **DepósitoAPlazo**, ambas clases que implementan **FormaDeAhorro**, al igual que **CuentaDeAhorro**.
Cada una de estas será descrita en detalle a continuación:

2.1. Cuenta

Esta clase abstracta debe incluir, al menos, los siguientes atributos:

- **saldo**: Que indique la cantidad de dinero que tiene disponible dicha cuenta.
- **númeroDeCuenta**: Que identifique a la cuenta. Toda cuenta debe tener uno y deben corresponder al orden en que son creadas. Es decir, la primera debe tener el número 1, la segunda el 2 y así.

Además, debe incluir al menos los siguientes métodos:

- **abonar(monto)**: Este método simplemente le suma cierto monto al `saldo` de la cuenta.
- **retirar(monto)**: Este método resta cierto monto al saldo de la cuenta **a menos** que el saldo no sea suficiente. En dicho caso, **levanta una excepción**.

- **getSaldo()**: Que retorne el saldo disponible en la cuenta.
- **getNúmeroDeCuenta()**: Que retorne el `númeroDeCuenta` que identifica a esta.
- Un constructor que reciba el saldo inicial de la cuenta.

2.2. CuentaDeAhorro

Las cuentas de ahorro se distinguen por ser una forma de ahorro y una cuenta al mismo tiempo, por lo que debe ser una clase hija de `Cuenta` e implementar la interfaz `FormaDeAhorro`.
Esta clase debe incluir, al menos, los siguientes atributos:

- **tasa**: Indica el crecimiento diario del monto guardado en esta cuenta, para ser utilizado por el método que corresponda. Normalmente la tasa es de 1,0001, pero a veces se crean cuentas con otra tasa. La tasa no puede ser modificada una vez creada la cuenta.
- **días**: Días para que esté disponible la opción de retirar dinero. En las cuentas de ahorro, después de retirar dinero, se deben esperar siete días antes de que vuelva a ser posible retirar dinero. Mientras sea mayor a cero, cada día se debe decrementar este número en uno.

2.3. Tarjeta

Esta interfaz tiene un solo método:

- **transferir(cuenta, monto)**: Este método llama a los métodos `retirar()` de la cuenta desde la que es llamado e invoca a `abonar()` en la cuenta que recibe como parámetro (es decir, el parámetro que recibe corresponde a la cuenta destino de la transferencia).

2.4. CuentaVista

Esta clase debe heredar de `Cuenta` e implementar la interfaz `Tarjeta`.

2.5. CuentaCorriente

Muy semejante a la `CuentaVista`, excepto porque las cuentas corrientes se caracterizan por tener una línea de crédito asociada. Es decir, el saldo de estas cuentas puede ser negativo.

2.6. FormaDeAhorro

Esta interfaz incluye los siguientes métodos:

- **actualizar()**: Actualiza el monto asociado a dicha forma de ahorro, multiplicándolo por su respectiva tasa de interés. Los detalles de la implementación dependen del tipo de ahorro.
- **getMonto()**: Retorna el monto asociado a dicha forma de ahorro.

2.7. FondoMutuo

Esta clase debe incluir, al menos, los siguientes atributos:

- **ID:** Identificador único de la inversión. Será indicado en la creación del objeto correspondiente. Considere que puede ser cualquier `String`.
- **monto**: Cantidad de dinero que tiene la persona ahorrada en ese fondo. Irá variando a medida que pasen los días (a través de las llamadas al método que corresponda).
- **cuenta**: Número de la cuenta desde la cual se pidió. El monto será depositado en esta cuenta una vez finalice el ahorro.
- **crecimiento**: Número que indica el crecimiento diario de la economía. Es igual para todos los fondos mutuos, y al pasar los días, sus montos se multiplican por este número. Además, es utilizado en los cálculos de los depósitos a plazo.

Esta clase debe incluir, al menos los siguientes métodos:

- **cobrar()**: Retira todo el dinero del fondo mutuo y lo abona a su cuenta asociada. Al retornar este método, el `monto` que queda debe ser cero.
- **getID()**: Retorna el identificador único del fondo mutuo en cuestión.
- **setCrecimiento(crecimiento)**: Modifica el valor del crecimiento diario de la economía. Puede suponer que el valor del crecimiento siempre estará fijado cuando sea necesario, es decir, que este método será invocado antes de actualizar los montos de los fondos mutuos.

2.8. DepósitoAPlazo

Los depósitos a plazo se caracterizan por tener un crecimiento que se fija en el momento en que se crean, y corresponde a un tercio del crecimiento de la economía (es decir, si la tasa de crecimiento de la economía fuera de 1,3, el depósito a plazo se crearía con una tasa de 1,1). Esta clase debe incluir, al menos, los siguientes atributos:

- **monto**: Cantidad de dinero que tiene la persona ahorrada en ese fondo. Irá variando a medida que pasen los días (a través de las llamadas al método que corresponda).

- **días**: Días que faltan para retirar el dinero. Los depósitos a plazo no se pueden retirar antes de la fecha indicada, y cuando esta llega, son automáticamente cobrados y abonados en la cuenta correspondiente. Mientras sea mayor a cero, cada día se debe decrementar este número en uno.
- **cuenta**: Cuenta desde la cual se pidió. El `monto` será depositado en esta cuenta una vez finalice el ahorro.

2.9. ReadFile

Esta es la clase principal, y es la que será invocada por consola. Esta contiene el método `main()` para leer archivos de entrada y tiene la estructura de cómo se irán utilizando las clases de más arriba. Usted puede modificarlo para que calce con su implementación, pero cualquier modificación al código entregado debe ser justificada apropiadamente en el `Readme`. De esta clase, deberá implementar el siguiente método:

- **transferir(monto, origen destino, comando)**: Este método recibe los números de la cuenta de origen y de destino de una transferencia y transfiere cierta cantidad de dinero (especificada por su primer argumento), invocando al método `transferir` de la cuenta de origen. El último argumento corresponde a la línea del archivo por la cual se desea hacer esta transferencia (es decir, el String que empieza con 'transferir' y contiene los otros datos). Para implementar este método puede ayudarse del método `getCuenta(númeroDeCuenta)`, que recibiendo un número de cuenta, retorna el objeto correspondiente. **Debe** explicar en `Readme` brevemente cómo y por qué funciona este método.

Deberá implementar otros atributos y métodos en las clases antes mencionadas. Decidir cuáles y cómo implementarlos forma parte de la tarea.

3. Ejemplos y formato de salida

La clase principal que va adjunta incluye métodos para leer el archivo y entregar los resultados por consola en el formato correspondiente. No es necesario modificar estos métodos, pero si lo hace, justifique su decisión en el `Readme`. El programa debe ser invocado de la siguiente forma:

```
java ReadFile <archivo>
```

donde `<archivo>` debe ser reemplazado por el nombre del archivo de entrada. Básicamente, el código que se le entrega itera sobre las líneas del archivo de entrada y va llamando a los distintos métodos de las clases y objetos correspondientes. En otras palabras, tiene esta estructura:

```
for comando in archivo:
    switch comando {
    case "abonar":
```

```
        cuenta = getcuenta(númeroDeCuenta)
        cuenta.abonar(monto)
    case ...
```

En cuanto a la salida, el programa debe, cuando el archivo de entrada lo indique, el monto total de dinero que tiene la persona, sumando todos sus instrumentos financieros. En caso de ocurrir alguna excepción, debe mostrarse un mensaje de error y el comando que lo causó (es decir, imprimir por pantalla la línea del archivo de entrada que no se puede ejecutar).
Se adjuntan algunos ejemplos de archivo y salida esperados para estos, pero puede generar sus propios casos de prueba usando el formato especificado en el anexo.

4. Evaluación

El objetivo de esta tarea es la utilización del paradigma orientado a objetos, por lo que la evaluación se centra en que demuestren comprenderlo y lo apliquen correctamente, por eso se evaluarán los siguiente ítems:

- (15 pts.) Java: Su código, incluyendo todas las clases pedidas, compila sin errores.
- (5 pts.) Demuestra comprender el concepto de clase abstracta e interfaz y lo implementa correctamente.
- (10 pts.) Aprovecha la estructura de herencias y utiliza correctamente el casting de objetos.
- (10 pts.) Maneja correctamente excepciones y las utiliza como parte de su código.
- (10 pts.) Encapsulamiento: Utiliza correctamente los modificadores de acceso, siendo lo más restrictivo posible, para mayor seguridad, y **no** funcionan los códigos *hacker* del USM.
- (30 pts.) Su código funciona para los casos de prueba.
- (10 pts.) Demuestra manejo del polimorfismo de métodos, implementándolo correctamente y llamando a dichos métodos adecuadamente.
- (10 pts.) Diferencia entre métodos y atributos de objeto y de clase, utilizándolos adecuadamente (es decir, usa el modificador `static` al menos una vez, justificadamente).
- (-20 pts.) Baja calidad del código (ver Condiciones de evaluación del código).
- Descuentos reglamentarios por atraso.

5. Consideraciones

- Debe incluir un `Makefile` que tenga un comando para compilar todas sus clases.
- Su programa debe ser ejecutable en los computadores del LabComp, con la versión de Java que tienen estos instalados (openjdk version '1.8.0_252'). Recuerde que puede acceder a estos vía SSH para probar su programa. Sin embargo, dado que se están considerando funcionalidades estándar de Java, es poco probable que su código presente conflictos de versiones.
- Consideraciones adicionales que surjan en el Moodle de la asignatura deben considerarse, siempre y cuando aparezcan dentro de un tiempo razonable antes de la entrega. **No se aceptarán preguntas de la tarea el día de su entrega.**

Condiciones de evaluación del código

Las siguientes condiciones son transversales al código y no cumplir con ellas puede llevar a descuentos en los ítemes correspondientes o en la nota general.

- **Ejecución correcta**: que funcionen los casos de prueba y no sea posible encontrar casos en que el programa entregue una respuesta equivocada.
- **Calidad del programa**: uso adecuado de funciones, uso de estructuras de control, uso de estructuras de datos, modularización correcta, código claro y simple.

  > "After more than 45 years in the field, I am still convinced that in computing, elegance is not a dispensable luxury but a quality that decides between success and failure." - E.W. Dijkstra

  Aunque la elegancia es difícil de definir, se puede decir que es una mezcla de simpleza y efectividad, haga una solución pequeña pero robusta, no se preocupe de resolver más de lo solicitado si eso complica su programa, reutilice código para las diferentes funcionalidades. Buscar resolver problemas que no existen y preoptimizar son la raíz de los males, el recurso más escaso es la capacidad mental del programador.
- **Complejidad computacional adecuada**: Que la solución ejecute en un tiempo aceptable (menos de 10[s]) para los casos de ejemplo.
- **Código ordenado**: nombres adecuados, indentación correcta, comentarios suficientes, ausencia de código comentado. Siga las convenciones del lenguaje.

Condiciones de entrega

- La tarea se realizará *individualmente* (esto es grupos de una persona), sin excepciones.
- Al azar, o ante sospecha de copia, los ayudantes interrogarán estudiantes sobre el código entregado, en tal caso, la nota obtenida en la interrogación reemplaza la nota de la revisión de la tarea. El faltar a la interrogación sin la justificación correspondiente significa nota cero.
- La entrega debe realizarse vía Moodle en un *tarball* en el área designada al efecto, bajo el formato `tarea-3-`*`rol`*`.tar.gz` (`rol` con dígito verificador y sin guión).
  Dicho *tarball* debe contener su código y un `README` indicando las instrucciones de compilación y ejecución cuando corresponda. Opcionalmente, información adicional que deba saber el evaluador.
- El plazo de entrega es el indicado en Moodle. Por cada día de atraso se descontarán 5 puntos y a partir del séptimo día de atraso no se recibirán más tareas y la nota se hará automáticamente cero.

Anexo

Formato .jaBank

Los archivos de entrada que recibirá el programa consisten en una serie de comandos, uno en cada línea. Cada línea empieza con el nombre del comando, seguido de los argumentos que este requiere, separados por espacios. Por ejemplo, si se desea crear una nueva cuenta vista con un saldo inicial de 1000 pesos, el archivo contendrá la siguiente línea:

```
crearCuentaVista 1000
```

A continuación encontrará una lista de todos los comandos posibles, seguidos de los parámetros que estos requieren

- `crearCuentaVista <int monto>`
- `crearCuentaCorriente <int monto>`
- `crearCuentaDeAhorro <int monto>`
- `crearCuentaDeAhorro <int monto><int tasa>`
- `crearFondoMutuo <int monto><int cuenta><String(sin espacios) ID>`
- `crearDepósitoAPlazo <int monto><int cuenta><int días>`
- `abonar <int monto><int cuenta>`

- `retirar <int monto><int cuenta>`
- `transferir <int monto><int origen><int destino>`
- `actualizarDía`
- `actualizarPIB <float nuevoPIB>`
- `verTotal`